Object Code Verification

by

Matthew Wahab

Thesis
Submitted to the University of Warwick
for the degree of
Doctor of Philosophy

Department of Computer Science

December 1998

Abstract

Object code is a program of a processor language and can be directly executed on a machine. Program verification constructs a formal proof that a program correctly implements its specification. Verifying object code therefore ensures that the program which is to be executed on a machine is correct. However, the nature of processor languages makes it difficult to specify and reason about object code programs in a formal system of logic. Furthermore, a proof of the correctness of an object code program will often be too large to construct manually because of the size of object code programs. The presence of pointers and computed jumps in object code programs constrains the use of automated tools to simplify object code verification.

This thesis develops an abstract language which is expressive enough to describe any sequential object code program. The abstract language supports the definition of program logics in which to specify and verify object code programs. This allows the object code programs of any processor language to be verified in a single system of logic. The abstract language is expressive enough that a single command is enough to describe the behaviour of any processor instruction. An object code program can therefore be translated to the abstract language by replacing each instruction with the equivalent command of the abstract language. This ensures that the use of the abstract language does not increase the difficulty of verifying an object code program.

The verification of an object code program can be simplified by constructing an abstraction of the program and showing that the abstraction correctly implements the program specification. Methods for abstracting programs of the abstract language are developed which consider only the text of a program. These methods are based on describing a finite sequence of commands as a single, equivalent, command of the abstract language. This is used to define transformations which abstract a program by replacing groups of program commands with a single command. The abstraction of a program formed in this way can be verified in the same system of logic as the original program. Because the transformations consider only the program text, they are suitable for efficient mechanisation in an automated proof tool. By reducing the number of commands which must be considered, these methods can reduce the manual work needed to verify a program.

The use of an abstract language allows object code programs to be specified and verified in a system of logic while the use of abstraction to simplify programs makes verification practical. As examples, object code programs for two different processors are modelled, abstracted and verified in terms of the abstract language. Features of processor languages and of object code programs which affect verification and abstraction are also summarised.

Acknowledgements

I am grateful to Professor Mathai Joseph for suggesting the topic of this thesis and for his support and encouragement during the course of the work. Professor Joseph's guidance in the principles and issues surrounding program verification was invaluable and the thesis has benefited by his comments on the contents and the text. I am also grateful to Dr. Sara Kalvala for discussions on the contents of the thesis and on the use of automated theorem provers, which help clarify many of the issues involved. Dr Kalvala also read through drafts of the thesis which was greatly improved by her comments.

Declaration

This thesis is the result of work carried out in accordance with the regulations of the University of Warwick. This thesis is the result of my own work and no part of it has previously been submitted for any qualification at any university.

Contents

List of Figures

Glossary of Symbols and Keywords

Expressions

Values	Basic values	37
Labels	Basic labels	37
Vars	Basic variables	37
Regs	Register identifiers	37
Names	Set of all variable names	37
name	Name constructor	37
pc	Program counter of $\mathcal{L}$	37
$\mathcal{B}$	Interpretation of values as Booleans	37
true, **false**	Boolean values of $\mathcal{L}$	37
State	The type of states	38
undef	Undefined elements	38
$\mathcal{F}$	Function identifiers	39
$\mathcal{I}_f$	Interpretation of function identifiers	39
arity	Arity of a function identifier	39
$\mathcal{F}_v$	Value functions	39
$\mathcal{F}_l$	Label functions	39
$\mathcal{F}_n$	Name functions	39
equal	Equality operator of $\mathcal{L}$	39
$\mathcal{E}$	Value expressions	42
$\mathcal{E}_l$	Label expressions	42
$\mathcal{E}_n$	Name expressions	42
$\mathcal{E}_b$	Boolean expressions of $\mathcal{L}$	42
$\mathcal{I}_e$	Interpretation of an expression as a value	43
$\mathcal{I}_l$	Interpretation of a label expression as a label	43
$\mathcal{I}_n$	Interpretation of a name expression as a name	43
$\mathcal{I}_b$	Interpretation of an expression as a Boolean	43
$\equiv_{(\mathcal{I},s)}$	Equivalence in state s under interpretation $\mathcal{I}$	47
$\equiv_s$	Equivalence in state s under assumed interpretation	47

$\equiv_{\mathcal{I}}$	Strong equivalence under interpretation $\mathcal{I}$ 47
$\equiv$	Strong equivalence under assumed interpretation 47
Alist	Assignment lists 51
nil	Empty assignment list 51
$\mathit{cons}((x, e), \mathit{al})$	Assignment list constructor 51
$(x, e) \cdot \mathit{al}$	Abbreviation of $\mathit{cons}((x, e), \mathit{al})$ 51
$\mathit{combine}(\mathit{al}, \mathit{bl})$	Combination of assignment lists 51
$\mathit{al} \oplus \mathit{bl}$	Abbreviation of $\mathit{combine}(\mathit{al}, \mathit{bl})$ 51
combine?	Recogniser for combined assignment lists 51
simple?	Recogniser for simple assignment lists 52
Slist	Simple assignment lists 52
initial	Prefix of an assignment list 52
$x \in_s \mathit{al}$	Membership of assignment list in state s 53
find	Expression associated with name in an assignment list 53
occs?, *Assoc*	Names and values in assignment list 62
correct?	Correct assignment lists 62
$\mathit{update}(s, \mathit{al})$	Update state s with assignment list al 54
subst	Substitution expression 54
$e \triangleleft \mathit{al}$	Substitution of al in value, label or name expression e 54, 55
$\mathit{bl} \triangleleft \mathit{al}$	Substitution of al in assignment list bl 56

Commands

$\mathcal{C}_0$	Set of all commands 59
if _ **then** _ **else** _	Conditional command 59
$:= (\mathit{al}, l)$	Assignment command 59
$l : c$	Labelled command 59
$\mathit{label}(c)$	Label of command c 59
regular?	Regular commands 60
$\mathcal{C}$	Commands of the language $\mathcal{L}$ 60
$\mathcal{I}_c(c)(s, t)$	Interpretation of command c in states s, t 64
goto, **abort**	Derived commands 65
enabled	Selection of command in a state 66
halt?	Failure of command 66
_; _	Sequential composition 68
	of labelled and conditional commands 69
	of assignment commands 69

Programs

program?	Defining predicate for programs of $\mathcal{L}$ 83

$\mathcal{P}$	Programs of $\mathcal{L}$	83
at	Access a command of a program	83
$p + c$	Addition of command c to program p	84
$p_1 \uplus p_s$	Combination of programs p_1 and p_2	84
Behaviour	The type of program behaviours	86
$\mathcal{I}_p(p)(\sigma)$	Interpretation of program p in behaviour σ	86
halt?	Failure of a program in a state	87
final?	Final states of a program	87
$s \overset{c}{\leadsto} t$	For command c, abbreviation of $\mathcal{I}_c(c)(s, t)$	88
$s \overset{p}{\leadsto} t$	State s leads to state t through program p	88
$c_1 \longmapsto c_2$	Command c_1 directly reaches command c_2	95
$c_1 \overset{p}{\longmapsto} c_2$	Command c_1 reaches command c_2 through p	97
path?$(p, c_1 c_2)$	There is a path from c_1 to c_2 through p	266
loopfree?(p)	Program p contains no loops	103
trace	Trace through a program	109
tset	Trace set of a program	109
mtrace	Maximal trace through a program	110
rmtrace	Restricted maximal trace through a program	110
$p_1 \sqsubseteq p_2$	Refinement between programs p_1 and p_2	90
$p_1 \leq_{I,al} p_2$	Program p_2 data refines program p_1	185

Regions

region?	Defining predicate for regions	99
$\mathcal{R}$	Set of all regions	99
region	Region constructor	100
unit	Unit region constructor	100
unit?	Unit region recogniser	100
label(r)	Label of region r	101
body(r)	Body of region r	101
head(r)	Head of region r	101
$c \in r$	Membership of a region	101
$r_1 \subseteq r_2, r_1 \subset r_2$	Subset relations between regions	101
$r_1 < r_2$	r_1 is a maximal sub-region of r_2	192
$s \overset{r}{\leadsto} t$	State s leads to state t through region r	101
$c_1 \overset{r}{\longmapsto} c_2$	Command c_1 reaches c_2 through region r	101
rest(r)	Sub-regions of region r	102
loopfree?(r)	Region r contains no loops	103
single?(r)	Region r is a single loop	103

enabled$(r)(s)$	Region r is enabled in state s	105
final?$(r)(s)$	State s is final for region r	105
halt?$(r)(s)$	Region r fails in state s	105
$\mathcal{I}_r(r)(s,t)$	Interpretation of region r in states s,t	106
$r_1 \sqsubseteq r_2$	Refinement between regions r_1, r_2	107
$c; A$	Generalised composition of command c with set A	113
T_1	Path transformation of regions	114
lphead?	Commands beginning a loop	116
lpheads	Set of commands beginning a loop	116
cuts	Cut-points of a region	116
lpbody	Body of a loop	117
loops	Sub-regions beginning with a cut-point	117
gtbody	Abstractions of paths in a region	118
T_2	General transformation of regions	118

Assertion Language

$\mathcal{A}$	Set of all assertions	74
$\neg, \wedge, \vee, \Rightarrow$	In general logical operators, also defined for $\mathcal{A}$	74
$\forall(\lambda v : F(v))$	Universal quantification in $\mathcal{A}$	74
$P \triangleleft al$	Substitution in assertion of $\mathcal{A}$	74
$\bigvee A$	Generalised disjunction over set of assertions A	192
$\vdash P$	Validity of assertion P	74
$\mathbf{wp}(c, Q)$	Weakest precondition of command c	75
$[P]p[Q]$	Specification of program p	122
$\mathbf{wp}_0(r, Q)$	Weakest precondition of a path in region r	192
$\mathbf{wp}_1(r, Q)$	Weakest liberal precondition of region r	192
$\mathbf{wp}(r, Q)$	Weakest precondition of region r	192

Expressions in Examples

Values, *Vars*, *Labels*	Values, variables and labels in examples	41
Regs	Set of register identifiers, in general	41
	for Motorola 68000 processor	141
	for PowerPC processor	157
ref	General name function	40
	also generic memory access	133
loc	General label function	40
not, **and**, **or**	Negation, conjunction and disjunction of expressions in $\mathcal{E}_b$	45
$=_a$	Synonym for equality of $\mathcal{L}$	39
$<_a, \geq_a$	Comparisons between expressions	41,45
$+_a, -_a, \times_a$	General arithmetic functions	40

x^y, mod_a	Exponentiation and modulus	41
Byte, *Word*, *Long*	Size of byte, word and long-word bit-vectors	130
mkBit	Bit constructor	131
bit$(i)(a)$	Bit i of bit-vector a	131
Byte	Byte constructor	131
Word, **mkWord**	Word constructors	131
Long, **mkLong**	Long-word constructors	131
mkQuad	Quad-word constructor	46
B$(i)(a)$	Byte i of bit-vector a	131
W$(i)(a)$	Word i of bit-vector a	131
ext	Sign extension of bit-vectors	132
cond(b, e_1, e_2)	Conditional expression with test b	171
Mem	Memory access for Alpha AXP	46
Inst	Label access for Alpha AXP	46
readl	Read long-word	133
writel	Write long-word	133
mkSR	Constructs value of M68000 status register	143
calcSR	Calculates value of M68000 status register	145
calcCRF	Calculates value of PowerPC condition register	162
$+_{sz}$, $-_{sz}$, $\times_{sz}$	Arithmetic on bit-vectors of size *sz*	132
$=_{sz}$, $<_{sz}$, $>_{sz}$	Comparison relations between bit-vectors of size *sz*	132
$+_{32}$, $-_{32}$, $\times_{32}$	Addition, subtraction and multiplication of long-words	132
$=_{32}$, $>_{32}$	Equality and greater-than between long-words	132
$+_{64}$, $-_{64}$, $\times_{64}$	Addition, subtraction and multiplication of quad-words	46
$=_{64}$, $>_{64}$	Equality and greater-than between quad-words	46

Chapter 1

Introduction

Programs are developed to produce a behaviour from a computer which will meet some requirement. A specification is a formal description of the properties required of the computer's behaviour. A program satisfies its specification, and is *correct*, if every behaviour which it produces has the properties described by the specification. The *correctness problem* is to show that a program satisfies its specification. This is often solved by *testing* the program on a sample of the data it is expected to process. Testing shows that the program is correct for the test cases and increases confidence in the program. It cannot show that a program satisfies its specification since it is always possible that the test data excludes those cases which cause errors.

Verification is a stronger method for establishing the correctness of program which constructs a mathematical proof that the program satisfies its specification. In verification, both the program and its specification are described in the terms of a logic. The rules of the logic are applied to show that the actions performed by the program produce a behaviour with the properties described by the specification. The logical description of the program is derived from the program text together with the semantics of the programming language. Since it is a description of the program requirements, the program specification must be produced manually and must be in a form which is suitable for verification. To verify a program therefore requires the program text, a semantics for the language and a method for translating the semantics to the terms of the logic.

Programs are typically written in high-level languages. These languages are often described informally and do not have a formally defined semantics. Informal descriptions can contain inconsistencies or errors and may be incomplete. To verify a high-level program, assumptions must therefore be made about the intended semantics of the high-level language. Because there can be no guarantee that these assumptions are correct, there is no guarantee that verifying the program will prove its correctness. Furthermore, a program of a high-level language cannot be directly executed on a computer but must be translated to an executable form, called *object code*, by a compiler. If a high-level program is to be verified then it is also necessary to verify the compiler since an error during the translation will lead to the execution of incorrect object code.

Object code is a program of a processor language and the semantics of these languages are suitable for use in a proof of correctness. Since object code can be executed on a machine, verifying the object code also ensures that the executed program is correct. However, the nature of processor languages means that it is difficult to describe, in the terms of a logic, the actions performed by object code. The size of object code programs also makes acute a general problem in program verification. Because the effect of each command in a program must be considered during program verification, the size of a correctness proof is proportional to the number of commands in the program. A processor language contains a large number of highly specialised commands (called *instructions*) which carry out relatively simple actions. These result in object code programs which are much larger than the equivalent high-level programs. Verifying even a simple program can require a large amount of work and the proof needed to verify a typical object code program is too large to be carried out manually.

Automated proof tools can assist in program verification by carrying out the basic inference and simplification steps which occur in any proof. Such tools apply the rules of the logic to the text of logical formulas. This allows quick and efficient reasoning about the properties established by an individual program command. However, an automated tool is less able to assist when considering the properties established by a group of commands. This requires inferences be drawn from the properties of the individual commands about the interaction between the commands in the group. Theorem proving techniques have been used to construct proof tools for verifying object code programs. However, these tools often require manual intervention to direct the course of the proof and this is made difficult by the methods used to reason about programs. The machine resources needed to use these tools also impose severe limits on the size of the program which can be verified.

Proof tools developed for object code verification are also limited to the programs of a single processor. A proof tool will have a model of a processor language, which is used to derive the properties of the language needed to verify a program. This model and its properties will be specialised to a single processor, because the instructions of each processor are unique to that processor language. To verify the object code of a different processor, it would be necessary to repeat the work carried out to construct the model and derive its properties. Because a processor can have several hundred instructions, this will be a time-consuming task effectively restricting the proof tool to the object code of a single processor only.

The problems posed by object code verification stem from the limitations of the methods used in program verification. These make it difficult to describe the behaviour of object code programs and to apply proof tools to infer properties of programs. To make object code verification practical, it must be possible to describe an object code program in a form suitable for a manual proof but which also allows the application of automated tools. It must also be possible to reduce the work which must be carried out manually, by allowing the use of an automated tool to simplify reasoning about a group of commands. These methods must be independent of any processor language, to avoid specialising any proof tool to a single processor. The development of such techniques would be useful for program verification in general because the problems of verifying object code apply equally to the verification of high-level programs.

1.1 Object Code Verification

This thesis describes a method for the verification of object code programs which allows the proof of correctness to be simplified by manipulating the text of a program. By only considering the text of a program, the method supports the use of automated proof tools during program verification. The method is independent of any processor language and provides a means for applying the same tools and techniques to different processor languages.

The object code programs which will be considered are sequential programs which do not modify themselves. The verification of a program is assumed to be carried out in *program logics*, logical systems such as those of Hoare (1969) and Dijkstra (1976), in which proof rules are applied to the text of a program. The specification of a program is assumed to describe its *liveness* properties, which state that execution of the program will eventually establish some property (Manna & Pnueli, 1991). It is also assumed that automated tools are available to assist in verifying a program. However, the method described in this thesis is independent of any particular proof tool. Where assumptions are made about the abilities of an automated tool, they can be satisfied by techniques which are well enough established to be considered standard.

To increase confidence in the correctness of the methods for object code verification developed in this thesis, the theory described in this thesis has been verified using the PVS theorem prover (Owre et al., 1993). Note, however, that the PVS theorem prover has not been used to implement a proof tool for program verification. The techniques needed to verify the theory are markedly different from those required to efficiently verify a program and the work with PVS cannot easily be used for program verification. The work with PVS is only intended to verify the theory which would be implemented by a proof tool for program verification.

Approach

The approach to object code verification followed in this thesis is to develop an abstract language, denoted $\mathcal{L}$, in which to describe the object code of arbitrary processors. The language $\mathcal{L}$ supports the definition of proof rules for program logics in which to verify programs of $\mathcal{L}$. An object code program is verified by translating the object code to a program of $\mathcal{L}$ then showing that the $\mathcal{L}$ program is correct. The language $\mathcal{L}$ is expressive enough that a processor instruction can be described by a single $\mathcal{L}$ command. The translation from object code to a program of $\mathcal{L}$ is therefore a simple replacement of each instruction of the object code program with its equivalent command of $\mathcal{L}$. The language $\mathcal{L}$ is independent of any processor language, allowing tools and techniques to be developed for the verification of object code independently of the processor language in which the object code is written.

The abstract language $\mathcal{L}$ provides the means for verifying object code programs but verifying object code as a program of $\mathcal{L}$ does not, in itself, simplify the proof of correctness. Because there is a command in the $\mathcal{L}$ program for every instruction of the object code, the number of commands which must be considered during the proof will not be reduced. The approach used to simplify the verification of a program p is based on constructing an *abstraction* p' of the program. The abstraction p' is a program of $\mathcal{L}$ which is correct only if the original program p is correct. The abstraction will, normally, be constructed to be simpler to verify than the program p, with a single command of the abstraction p' describing the behaviour of a group of commands in program p. Because the abstraction is a program of $\mathcal{L}$, the program logics and proof methods of proof applied to verify programs are equally applicable to the abstractions of programs. This allows the verification of a program to be carried out in a uniform environment, regardless of whether the program is the original or an abstraction of the original.

The abstractions of a program are constructed by applying transformations to the text of the program. Because only the program text is used to abstract from a program, the program transformations are suitable for efficient mechanisation. To use program transformations in verification they must be shown to be correct: the transformations must preserve the correctness of the program. The transformations described in this thesis will be shown to be correct, the correctness of a program is preserved by the transformation constructing its abstraction. In particular, the abstraction of an incorrect program will also be incorrect. It will therefore be impossible to verify an incorrect program by abstracting from the program. This ensures that it is safe to apply the transformations during program verification.

The use of the abstract language $\mathcal{L}$ to describe object code programs and the use of program transformations to simplify the proof of correctness will allow an object code program to verified through the following steps:

1. The object code program is translated into the language $\mathcal{L}$, to obtain a program p.
2. The program specification is described in terms of a program logic for the language $\mathcal{L}$.
3. An abstraction p' of program p is constructed from the program text, possibly with the use of an automated proof tool.
4. The rules of the program logic are applied to show that the abstraction p' establishes one or more properties required by the specification.
5. Steps 3 and 4 are repeated as necessary until it is shown that all properties required by the specification are established by the program.

With the exception of steps 1 and 3, this is the usual approach to program verification. The methods needed to carry out steps 1 and 3 are the principal contribution of this thesis. Step 1 exploits the ability to describe and verify an object code program of any processor as a program of $\mathcal{L}$. This allows a set of tools and techniques developed for verifying programs of $\mathcal{L}$ to be applied to the object code of a wide range of processors. Step 3 allows the verification of the $\mathcal{L}$ program to be simplified by using automated tools to abstract from the program. This approach to verifying a program is independent of the proof method used to prove properties of the program. Although the method of intermittent assertions (Manna, 1974; Burstall, 1974) will be used here, other proof methods can also be applied.

The main focus of the thesis is in the syntactic and semantic properties of the language $\mathcal{L}$ and their use to describe object code and for program abstraction. The syntax and semantics of $\mathcal{L}$ are intended to support the definition of proof rules and do not require the use of a particular program logic or method of proof. This provides flexibility in the choice of logics and methods used to verify a program. Furthermore, a program logic defined for the abstract language $\mathcal{L}$ will be a generic logic for reasoning about the object code of different processors. The use of an abstract language to model object code allows the use of abstraction to simplify the task of verifying program. The language $\mathcal{L}$ will therefore make the verification of object code both possible and practical. Any difficulty which occurs during a proof will be because the program is difficult to verify and not because it is an object code program.

1.2 Outline of the Thesis

After a section describing the mathematical **Notation** used, the thesis begins with the background of program verification and abstraction in **Chapter 2**. This describes the techniques used to reason about programs, including the standard program logics and proof methods. Processor languages will be described with a summary of the problems which make object code difficult to verify and the solutions which have been considered. The basis for reasoning about the relationship between a program and its abstraction will be summarised as will the approach used to show that a program transformation correctly constructs an abstraction. Alternative methods for verifying object code will also be considered as will alternatives to object code verification.

Commands of the language $\mathcal{L}$ are used to model the instructions of a processor language. The commands and expressions of the language $\mathcal{L}$ are described in **Chapter 3**. The syntax and the semantics of both the expressions and the commands are described in detail. The syntax of expressions and commands is used to define a transformation which abstracts from commands of $\mathcal{L}$ and which results in a command of $\mathcal{L}$. This transformation forms the basis for transformations which abstract from programs. Its correctness is justified from the semantics of the expressions and commands. The chapter ends with an example of proof rules for reasoning about the commands of $\mathcal{L}$.

Programs of $\mathcal{L}$ are used to describe object code in a form suitable for verification. The programs of $\mathcal{L}$ and methods for their abstraction are the subject of **Chapter 4**. The semantics of the programs are used to justify methods of reasoning about program execution. These, in turn, allow comparisons to be made between programs and their abstractions. Two methods for constructing an abstraction are described: the first is based on the abstraction of manually chosen commands of a program. The second method is intended for use in an automated tool and assumes the commands are chosen mechanically. Both methods define transformations on a program which are shown to preserve the correctness of the program. The chapter ends with an example of a program logic which is used to verify a simple program of $\mathcal{L}$.

The language $\mathcal{L}$ and the methods for abstracting programs are intended to allow arbitrary object code programs to be modelled and verified. **Chapter 5** describes simple models of the Motorola 680000 (Motorola, 1986) and the PowerPC (Motorola Inc. and IBM Corp., 1997) processor languages. The main interest is in the ability to model object code programs in terms of the language $\mathcal{L}$. If this is possible then the ability to manipulate and verify object code of the processors follows. The Motorola 68000 and PowerPC do not present any great difficulty and simple programs for each processor are modelled in the language $\mathcal{L}$ and shown to be correct. However, processor languages can have features which make the abstraction of object programs difficult. Some of these features and their effect on the methods used to model and verify object code are described.

The use of an abstract language permits proof methods, defined for the abstract language, to be applied to the object code of different processors. The abstract language $\mathcal{L}$ also allows the use of standard methods for reasoning about programs to be applied to object code programs. Program verification has been extensively studied and there are many such techniques. **Chapter 6** describes two proof methods which can be applied to programs of $\mathcal{L}$. The first is an example of a method which can make transferring a proof of correctness between the object code of different processors simpler. The second is an example of the definition of a proof method for the language $\mathcal{L}$ which can be applied to arbitrary object code programs.

The theory described in Chapter 3 and Chapter 4 has been verified using the PVS theorem prover (Owre et al., 1993). The theory underlying the proof methods described in Chapter 6 has also been verified with PVS. The theory defined for verification by the theorem prover closely follows that described in Chapters 3, 4 and 6. The major differences are in the additional definitions and lemmas required to prove the theory in PVS and in the definitions of the expressions of $\mathcal{L}$. **Chapter 7** describes the role played by the PVS theorem prover in the development of the theory. The differences between the theory defined in Chapters 3, 4 and 6 and the theory as it is defined in the PVS specification language will also be described. The proofs for Chapter 3 and Chapter 4 are given in **Appendix C** and **Appendix D** respectively; the proofs for Chapter 6 are given in **Appendix E**. The theory described in this thesis is intended to allow the implementation of a proof tool for program verification. The techniques and methods needed for such an implementation will be briefly discussed.

The thesis concludes with a summary, in **Chapter 8**, of the methods for verifying an object program. The transformations for abstracting programs are intended for implementation in an automated proof tool. Many of the issues surrounding the use of automated tools in program verification have already been considered and will be discussed here. The proposed techniques are related to previous work in the field and extensions to the work are considered.

Notation

Types: The basic types are the *booleans*, the *integers* and the non-negative integers. The type of boolean values is denoted *boolean* and the integers are denoted $\mathbb{Z}$. The non-negative integers are called the natural numbers, or simply the *naturals*, and denoted $\mathbb{N}$. The type of *functions* from T to S is written $T \to S$ and $\to$ associates to the right. Functions from a type T to the booleans are called *predicates* on T. The cross-product of types T and S is written $T \times S$. That a variable or constant x has type T will be denoted $(x : T)$. Where the type of a variable is obvious from the context it will sometimes be omitted.

Types are defined and the definition of a type T may be parameterised with a type S: the application of T to S is written $T(S)$ and is a type. Where S can be an arbitrary type it may be omitted: T is written for $T(S)$ when S may be any type. In general, types will only be given in the mathematical definitions and formulas and may be omitted from the text.

Functions: Functions are generally named in lower-case and predicates are generally named with a question mark. For example, *eg* names a function and *eg?* names a predicate. Functions may be defined recursively and such a definition is either by primitive recursion or can be transformed to primitive recursion (Kleene, 1952).

Operators are functions whose first argument is a cross product of types. The operator may be identified symbolically and its type may be given with underline characters indicating the placement of arguments. For example, the type of the addition operator on the integers is written $\_ + \_ : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}$.

Logical operators: The boolean values true and false are written *true* and *false* respectively. The conjunction, disjunction, implication and logical equivalence operators are written $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ respectively. The universal and existential quantifiers are $\forall$ and $\exists$ respectively. Equality is written $=$ and a defining equality is written $\stackrel{\text{def}}{=}$.

The presentation of a formula having the form $A_1 \wedge \ldots \wedge A_n \Rightarrow C$ may be given as

$$\frac{A_1 \quad \cdots \quad A_n}{C}$$

Free variables in the formula are universally quantified.

Basic operators: The operator λ applied to a variable x of type T and expression e of type S is written $\lambda x : e$ and forms a function of type $T \to S$. Nested lambda expressions are abbreviated, $\lambda x_1, x_2, \ldots x_n : e$

Textual substitution replaces variables $x_1, \ldots, x_n$ with expressions $e_1, \ldots, e_n$ in expression f (where the variables occur free) and is written $f[e_1, \ldots, e_n/x_1, \ldots, x_n]$.

The *arithmetic operators* addition, subtraction, multiplication, division and remainder applied to integers x and y are written $x + y$, $x - y$, $x \times y$, $x \div y$ and $x \bmod y$ respectively. The remainder, mod, satisfies $((x \div y) \times y) + (x \bmod y) = x$.

A *sequence* of a type T is a total function from the naturals to T, $\mathit{seq}(T) \stackrel{\text{def}}{=} (\mathbb{N} \to T)$.

The *suffix* of a sequence σ beginning at the nth element, written σ_n, is the sequence defined $\sigma_n \stackrel{\text{def}}{=} \lambda(m : \mathbb{N}) : \sigma(n+m)$.

Sets: In a set a having type *Set*(T), every member of a has type T and a is said to be *a set of type* T. Sets may be defined by comprehension, written $\{x : T \mid f\}$ where x is a variable of type T, and f is a boolean formula in which x may occur. The empty set is written $\{\}$ and a is a singleton set if there is an element x such that $a = \{x\}$.The membership operator is written $\in$, the union, intersection, subset and proper subset operators are written $\cup$, $\cap$, $\subseteq$ and $\subset$ respectively. The difference between sets a and b of type T is written $a - b$ and defined $a - b \stackrel{\text{def}}{=} \{x : T \mid x \in a \wedge \neg x \in b\}$. The power-set of a set a of type T is written $\mathbb{P}a$, has type *Set*(*Set*(T)) and definition $\mathbb{P}a \stackrel{\text{def}}{=} \{b : \mathit{Set}(T) \mid b \subseteq a\}$.

The *image of a function* f with type $(S \to T)$ on a set a with type *Set*(S) is a set with type *Set*(T) and definition:

$$f(a) \stackrel{\text{def}}{=} \{x : T | \exists (y : S) : y \in a \wedge x = f(y)\}$$

Sets and types are equivalent: a set a of type T is the subtype of T containing only those elements of T which are also in a. A type T is the set containing all elements of type T. Sets and predicates are also equivalent: the set a of type $Set(T)$ is the predicate $\lambda(x : T) : x \in a$. The predicate p on type T is the set $\{x : T \mid p(x)\}$.

The Hilbert epsilon operator is written ϵ and its application to a set a is written ϵa. When a has type *Set*(T), the result of ϵa has type T. When a is not empty, the result ϵa is a arbitrary member of a, $(\epsilon a) \in a$. When a is empty and a has the type *Set*(T), the result of ϵa is an arbitrary constant of type T.

Finite Sets: The finite sets of type T have type *FiniteSet*(T). The empty set is finite and if a is finite then so is $\{x\} \cup a$. Any subset of a finite set is also finite as is the power-set of a finite set. If sets a and b are finite sets of type T and S respectively, the cross-product of a and b, defined $\{(x, y) : (T \times S) \mid x \in a \wedge y \in b\}$, is also finite (Levy, 1979). For finite set a, $|a|$ is the cardinality of a satisfying $|\{\}| = 0$ and $|a \cup \{x\}| = 1 + |a|$ for $x \notin a$.

Two induction schemes are assumed on finite sets. Let Φ be a predicate on finite sets (with type *FiniteSet*$(T) \to$ *boolean* for some T).

(Finite induction)

$$\frac{\Phi(\{\}) \quad \forall b : \Phi(b) \Rightarrow (\forall x : x \notin b \Rightarrow \Phi(b \cup \{x\}))}{\forall a : \Phi(a)}$$

(Strong finite induction)

$$\frac{\forall b : (\forall c : c \subset b \Rightarrow \Phi(c)) \Rightarrow \Phi(b)}{\forall a : \Phi(a)}$$

Finite induction follows from the definition of a finite set. Strong finite induction can be proved from well-founded induction on the cardinality of a finite set.

Inductive definitions: The presentation of the theory makes use of inductive definitions of predicates and types. The basis of the inductive definitions is described by Paulson (1994b, 1995b) and by Camilleri & Melham (1992). The inductive definition of a predicate has the usual form. For example, the transitive closure of a relation R, written R^+, is defined by induction. If R is a predicate on type $(X \times Y)$ then R^+ is defined as the base and inductive cases.

$$\text{(Base case)} \quad \frac{R(x,y)}{R^+(x,y)} \qquad \text{(Inductive case)} \quad \frac{R^+(x,y) \quad R^+(y,z)}{R^+(x,z)}$$

The reflexive transitive closure of a relation R is written R^* and defined $R^*(x,y) \stackrel{\text{def}}{=} x = y \vee R^+(x,y)$.

Inductively Defined Types: The inductive definition of types is for the purposes of the presentation and is intended to be equivalent to a definition in Backus-Naur form.

The inductive definition of a type is on a set of *constructor* functions and a *basis* set. The constructors are not defined but are assumed to be such that the definition is *free* (see Loeckx & Sieber, 1987). In particular, it is assumed that induction and recursion on the structure of elements is possible. For example, the type of lists is written *list*(T), for some type T. The definition is from the basis set $B = \{\mathit{nil}\}$ and constructor function *cons* of type $(T \times \mathit{list}(T)) \to \mathit{list}(T)$. The type *list*$(T)$ is defined inductively.

$$\mathit{nil} \in \mathit{list}(T) \qquad \frac{x \in T \quad y \in \mathit{list}(T)}{\mathit{cons}(x,y) \in \mathit{list}(T)}$$

For an inductively defined type T, a partial order $\ll$ is assumed between the elements of the type. If x, y have type T then $x \ll y$ states that x forms part of the structure of y and x is said to be a *sub-term* of y. For example, the relation $\ll$ on the lists, *list*(T), can be defined by recursion.

$$\begin{aligned} x \ll \mathit{nil} &\stackrel{\text{def}}{=} x = \mathit{nil} \\ x \ll \mathit{cons}(y,l) &\stackrel{\text{def}}{=} x = \mathit{cons}(y,l) \vee x \ll l \end{aligned}$$

Graphs: A graph is a pair (V, R) where V is a set of type T and R is a relation, of type $(V \times V) \to \mathit{boolean}$. For $x, y \in V$, x *reaches* y through the graph iff the transitive closure of R is *true* for x and y, $R^+(x,y)$. A *path* through the graph from x to y is a sequence $v_1, v_2, \ldots, v_n$ such that $R(x, v_1,)$, $R(v_n, y)$ and, for $1 \leq j < n$, $R(v_j, v_{j+1})$ are *true* and $v_i = v_j$ iff $i = j$, where $1 \leq i, j \leq n$ and $v_i, v_j \in V - \{x, y\}$. If the sequence is empty, $n = 0$, then the path is made up of x and y, $R(x,y)$. There is a path from x to y iff x reaches y. There is a *cycle* in a graph, called a *loop* in the set V, iff there is a $x \in V$ such that $R^+(x,x)$.

Chapter 2

Verification and Abstraction

The method for object code verification described in this thesis is based on the use of the abstract language $\mathcal{L}$. This has two functions: the first, as a description language in which the programs of different processors can be modelled and verified. The second, as a standard form in which to transform and abstract programs. For both purposes, only the syntax of the language is used. For verification, the syntax of the abstract language $\mathcal{L}$ must therefore support both the description of object code and the definition of program logics in which to verify $\mathcal{L}$ programs. For abstraction, the syntax of $\mathcal{L}$ must support the manipulation of the commands and programs of $\mathcal{L}$. The semantics of the language $\mathcal{L}$ are used to justify the rules of program logics and the correctness of program transformations. This ensures that the methods used to verify and abstract from programs are sound and therefore it is not possible to verify an incorrect program.

The requirements of the abstract language $\mathcal{L}$ are those of the methods used to model, verify and abstract programs. The particular areas of interest are:

- The principles underlying program verification: modelling and specifying program behaviour and the proof methods for program verification.
- Processor languages: the execution and data models of processors and the features which complicate the verification of object code programs.
- Abstraction of programs: methods for constructing abstractions of programs and the justification for verifying a program by verifying its abstraction.
- The use of automated tools in program verification.

A number of methods for showing the correctness of object code programs have been proposed. These include the use of theorem provers to simulate processors and methods for producing correct object code from verified high-level programs. A common factor is the attempt to simplify the verification of the program which is to be executed on the machine. Although these methods have a number of drawbacks, some of the underlying techniques are useful for program verification in general and these techniques will also be reviewed.

2.1 Program Verification

Program verification has been extensively studied; its methods are described by Apt (1981), Cousot (1990) and Francez (1992) among others. A mathematical treatment is given by Loeckx & Sieber (1987). The difficulty of verifying a program is determined by the proof methods for verification. These describe how a program is verified and also suggest how the task of verifying programs can be simplified. To verify that a program is correct, the result of executing the program must be shown to establish the properties required by the program specification. This section will summarise the methods used to specify and to verify programs, beginning with the relationship between programs and specifications. The requirements of program logics and the methods for proving correctness will then be described.

Programs and Specifications

A program is made up of commands which evaluate expressions and assign the resulting values to variables. The program is verified by reasoning about the properties of the program variables during the program execution. A program *state* is an assignment of values to variables and a sequence of states describes the *behaviour* of the program when it is executed. Each command *begins* in a state and *produces* (or *ends*) in a state by assigning values to variables. No (executable) command can assign more than one value to a variable. Each program begins in an *initial* state and if the program terminates then it does so in a *final* state.

Execution of a program begins with the *selection* of a command. This is the *first* command of the program and, when executed, it produces either a final or an *intermediate* state. If the state is intermediate then the program execution continues, repeatedly selecting and executing commands until a final state is produced. A command can be executed any number of times; there is a *loop* in the program if one or more commands are repeatedly executed. The semantics of the programming language determine how commands of a program are selected as well as the relationship between the states in which a command begins and ends. The behaviour of a program is described by the initial state and the sequence of states produced during its execution.

A program specification is made up of a *precondition*, describing the properties of the initial state, and a *postcondition*, describing the properties which must be satisfied by a state during the program execution. If the program terminates then the postcondition must be satisfied by the final state. Any intermediate state of a non-terminating program which satisfies the postcondition is final with respect to the specification. A program or command is *partially* correct if its specification is satisfied whenever the program terminates; a program which never terminates is always partially correct. For *total correctness*, the program or command must both terminate and satisfy the specification. For example, assume the semantics of commands are defined by interpretation function $\mathcal{I}$ such that command c begins in state s and produces state t iff $\mathcal{I}(c)(s,t)$. Also assume P and Q are predicates on states and that P is the precondition and Q the postcondition of command c. Command c is partially correct if $\forall s,t : P(s) \wedge \mathcal{I}(c)(s,t) \Rightarrow Q(t)$ where s,t are states. The command is totally correct if $\forall s : P(s) \Rightarrow \exists t : \mathcal{I}(c)(s,t) \wedge Q(t)$.

2.1.1 Program Logics

Verification is carried out in a program logic in which axioms and proof rules are used to show that commands and programs satisfy a specification. The axioms and proof rules of the program logic specify the semantics of the programming language. A program logic also provides a means for specifying commands and programs as formulas of the logic. A program is verified by using the proof rules to show that the logical formula specifying the program is *true*. Approaches to defining proof rules for programs vary but the underlying principles are similar to those used for commands. The description here will therefore concentrate on proof rules for commands.

The method used to specify commands and programs determines whether a program logic is used to establish partial correctness or total correctness. Hoare (1969) describes a logic, based on the work of Floyd (1967), for verifying partial correctness. In this, a command c is specified by a formula of the form $\{P\}c\{Q\}$. This asserts that if c, beginning in a state satisfying precondition P, terminates then it produces a state satisfying postcondition Q. Manna (1974) developed a variant of Hoare's logic in which total correctness can be verified. Dijkstra (1976) describes a second program logic, the *wp* calculus, also for verifying total correctness. This extends a standard logical system with a *predicate transformer*, *wp*. Applied to a command c and postcondition Q, this calculates the weakest precondition, $wp(c, Q)$, which must be satisfied for command c to terminate and establish Q. A specification based in the *wp* calculus has the form $P \Rightarrow wp(c, Q)$, where P is the precondition and Q the postcondition for command c.

Proof Rules of a Program Logic

The axioms and proof rules of a program logic are determined by the semantics of the programming language and by the methods used to prove a program correct. The proof rules are defined on the syntax of the programming language; their correctness is justified from the semantics of the language. The semantics of individual commands are, in general, specified as axioms of the logic. Compound commands, formed by combining two or more commands, are specified in terms of proof rules. These describe how the behaviour of the component commands establishes the specification of the compound command.

For most commands of a programming language, the definition of proof rules requires only the standard logical operators (conjunction, negation, etc). For example, consider the conditional command *if* b *then* c_1 *else* c_2, where b is a test, command c_1 the *true branch* (executed if the test succeeds) and command c_2 the *false branch*. The conditional command is specified with arbitrary pre- and postconditions P and Q by the proof rule:

$$\frac{P \land b \Rightarrow wp(c_1, Q) \quad P \land \neg b \Rightarrow wp(c_2, Q)}{P \Rightarrow wp(\textit{if } b \textit{ then } c_1 \textit{ else } c_2, Q)}$$

This rule allows the specification of the conditional command, to be established by deriving specifications for the commands c_1 and c_2 (Dijkstra, 1976).

Proof rules for commands can also require specialised operators. In particular, *assignment commands* require operators for manipulating the variables and expressions of the programming language. Assignment commands are used to make changes to a machine state. The most general form, a simultaneous assignment command, is written $x_1, \ldots, x_n := e_1, \ldots, e_n$ where $x_1, \ldots, x_n$ are variables and $e_1, \ldots, e_n$ are expressions. This simultaneously assigns to each variable x_i the result of evaluating expression e_i. The assignment is specified, by its affect on an arbitrary precondition P, as the assignment axiom (Dijkstra, 1976):

$$P[e_1, \ldots, e_n/x_1, \ldots, x_n] \Rightarrow wp(x_1, \ldots, x_n := e_1, \ldots, e_n, P)$$

This axiom uses textual substitution to model the changes made to precondition P by the assignment to the variables. An implicit requirement of the axiom is that the assignment is executable. An assignment commands is unexecutable if a variable is assigned two distinct values: no state can associate two values with a single variable. To ensure that all assignments are executable, simple languages often exclude as syntactically incorrect assignments in which the same variable occurs twice on the left-hand side: $i \neq j \Rightarrow x_i \neq x_j$.

In languages which contain *pointers* or *arrays*, both the assignment axiom and the syntactic restriction on assignment commands are invalid (Francez, 1992). Pointers are expressions which identify a variable and which can depend on the values of variables; an array can be considered a form of pointer. Pointers give rise to the *aliasing problem*: it is not possible to determine syntactically whether a pointer identifies a given variable. Consequently, it is not possible to use syntactic comparisons between variables to determine whether an assignment command is executable. Neither is it possible to use textual substitution to define proof rules for assignment commands, since this is also based on syntactic comparisons. Instead, proof rules for assignment commands in the presence of pointers must use specialised substitution operators (Cartwright & Oppen, 1981; Manna & Waldinger, 1981). Unexecutable commands are avoided by restricting assignment commands to a single variable or (equivalently) by interpreting a multiple assignment as a sequence of single assignments (Gries, 1981; Cartwright & Oppen, 1981).

The specification of programs and proof rules for programs depend on the type of programming language being considered. A programming language is either a *structured* or a *flow-graph language* (Loeckx & Sieber, 1987). A program of a structured language is a compound command and can be specified as a command (e.g. using Hoare formulas or the *wp* function). There are a number of approaches to specifying flow-graph programs (Loeckx & Sieber, 1987; Francez, 1992). Common to all is a method for associating pre- and postconditions with the program being specified. All proof rules for programs allow the specification of a program to be established from specifications of the commands (similarly to rules for compound commands). The proof rules for programs also determine the proof methods which can be used to verify a program.

2.1.2 Proof Methods for Program Verification

There are two methods for verifying a program, both of which can be applied to any sequential program which does not modify itself. The method of inductive assertions verifies the partial correctness of programs (Floyd, 1967): a program (or command) is said to *establish* postcondition Q from precondition P if whenever the program beginning in a state satisfying P terminates, it does so in a state which satisfies Q. The method of intermittent assertions verifies the total correctness of a program (Manna, 1974; Burstall, 1974): the program is said to *establish* postcondition Q from precondition P if execution of a program in a state satisfying P eventually terminates and does so in a state satisfying Q. The method of intermittent assertions is equivalent to the method of inductive assertions together with a proof that the program terminates (Cousot & Cousot, 1987).

In both proof methods, the program to be verified is broken down into sequences of commands each of which is shown to satisfy an intermediate specification. These intermediate specifications are then used to establish the program specification. The proof methods describe how the sequences of commands and their intermediate specifications must be chosen. The proof methods also describe how the intermediate specifications are established from the sequences of commands; the approach used suggests a method for simplifying program verification.

Proving Correctness

The method of inductive assertions and the method of intermittent assertions are both applied in the same way. Assume program p is to establish postcondition Q from precondition P. Also assume that the program terminates in a state in which a *final* command is selected. As the first step, a set of *cut-points* are chosen such that the first command of the program is a cut-point, the final command is a cut-point and at least one command in every loop is a cut-point. Each cut-point cut_i is associated with an assertion A_i. If cut_i is the first command then A_i is the precondition of the program P. If it is the final command, then A_i is the postcondition of the program Q. All other assertions are *intermediate assertions*, chosen so that if the program establishes the intermediate assertions then the program specification will be established as a logical consequence. The cut-points at the loops are needed to allow the properties of loops in the program to be established by induction on the program variables (Floyd, 1967).

Associating the cut-points with the intermediate assertions breaks down the proof that the program p establishes Q from precondition P to the smaller proofs of each of the statements:

From precondition P, program p establishes assertion A_i.
From assertion A_i, program p establishes assertion A_j

$$\vdots$$

From assertion A_k, program p establishes the postcondition Q

The assertions A_i form intermediate specifications to be satisfied by the sequences of commands between each cut-point in the program. For each cut-point cut_i, the sequence of commands $cut_i, c_1, \ldots, c_n$ is formed up to, but not including, the next cut-point: command c_n is followed by a cut-point cut_j. The cut-points do not need to be distinct, if cut_i is cut_j then the commands are part of a loop in the program. The intermediate assertions, A_i and A_j, associated with the cut-points are the pre- and postcondition forming the specification to be satisfied by the sequence.

Each sequence of commands, $cut_i, c_1, \ldots, c_n$ is shown to satisfy its specification by associating a second set of assertions, $B_1, \ldots, B_n$, with each command in the sequence. Each pair of assertions B_i, B_{i+1} is a pre- and postcondition for command c_i. The precondition for cut_i is the assertion A_i and the postcondition for c_n is the assertion A_j. As before, assertions $B_1, \ldots, B_n$ are chosen so that if each command satisfies its specification then the sequence of commands also satisfies its specification. The sequence is then verified by a proof of each of the formulas:

$$\begin{array}{l} A_i \Rightarrow \mathit{wp}(\mathit{cut}_i, B_1) \\ B_1 \Rightarrow \mathit{wp}(c_1, B_2) \\ \qquad\quad \vdots \\ B_n \Rightarrow \mathit{wp}(c_n, A_j) \end{array}$$

If the sequence of commands is part of a loop (which may have been broken down into more than one sequence) then the proof will be by induction on the value of one or more variables. To show that the loop satisfies a specification, each sequence of commands in the loop must be shown to establish a property under each of the assumptions required for the induction scheme. The properties of each command in the sequence may therefore be the subject of more than one proof attempt.

Applying the Proof Methods

To verify a program using either the method of inductive assertions or the method of intermittent assertions requires a semantics for the programming language. This is needed to determine the effect of executing commands in the program as well as the method used to form sequences of commands in a program. For example, if a command c must be shown to satisfy the specification $B_i \Rightarrow \mathit{wp}(c, B_{i+1})$, then the effect of executing c in a state satisfying B_i must be to produce a state satisfying B_i. This can only be established from a specification of the command's semantics.

A large part of the work needed to verify a program is due to the method used to reason about the sequences of commands between program cut-points. Breaking down the sequences into individual commands means that the work needed to verify the program is proportional to the number of program commands. However, neither the proof method of Floyd (1967) nor that of Burstall (1974) requires that every command in the program be considered. It is only necessary to show that the program establishes the assertions at the program cut-points. If only these assertions are considered then the work needed is proportional to the number of cut-points. The number of cut-points will usually be less, and at worst no more, than the total number of program commands. It follows that transforming a program so that only the assertions at the program cut-points need to be considered can reduce the work needed to verify the program.

The language $\mathcal{L}$ has two purposes: the first, to provide a method for describing arbitrary object code programs in a form which allows the use of the proof methods of Floyd (1967) and Burstall (1974). The second, to support the transformation of a program so that only the assertions at the program cut-points need to be considered during verification. For this, the language $\mathcal{L}$ will allow a sequence of commands to be described as a single command of $\mathcal{L}$. This can be used to describe a sequence of commands between program cut-points as a single command c.

Rather than consider each command in the sequence, the specification of the sequence can then be established by reasoning about the single command c. This reduces the number of commands which must be considered to the number of sequences between program cut-points. The work required to verify a program is then proportional to the number of cut-points in the program.

For example, assume a sequence of commands, $cut_i, c_1, \ldots, c_n$, beginning with cut-point cut_i must be shown to establish postcondition A_{i+1} from precondition A_i. This is achieved by showing that each command c_i satisfies the intermediate specification, $B_j \Rightarrow wp(c_i, B_{j+1})$. This requires $n + 1$ different proofs, one for each individual command, to show that the sequence satisfies its specification. By constructing a command c which describes the behaviour of the sequence, it is only necessary to prove that $A_i \Rightarrow wp(c, A_{i+1})$. The proofs for the $n+1$ individual commands are reduced to a proof for a single command.

To reduce the work which must be carried out manually, the transformation of a program must be mechanised: an object code program contains too many commands to make manual transformation practical. Since programs of $\mathcal{L}$ are used to model object code, the requirements of the language $\mathcal{L}$ are determined by the features of processor languages and by the methods used to abstract $\mathcal{L}$ programs.

2.2 Processor Languages

A processor language provides the means for controlling the behaviour of a machine. The language of a processor is defined by the processor *architecture*, the specification which must be satisfied by a hardware implementation of the processor (Hayes, 1988). A processor can be considered an interpreter for the processor language: the action performed by an instruction is the result of the processor interpreting the instruction. The actions which can be carried out by an instruction are therefore determined by the actions which can be carried out by processor. This allows processor languages to be considered as a group, rather than individually. Any instruction (or feature) which occurs in the language of one processor could occur in any other processor. More generally, a processor language can (in principle) include any instruction whose semantics can be described as a function transforming the state of the machine.

Processors can have *privileged* and *unprivileged* execution modes, which impose restrictions on the instructions which can be executed. The privileged mode is intended for operating system software (Hayes, 1988). It provides instructions to control the resources available to the processor and to control the behaviour of the machine when an error occurs. The privileged mode of a processor may also allow a program to be executed in a parallel computing model. Application programs are executed in the unprivileged mode, which provides a subset of the processor instructions and a sequential computing model. Only sequential programs will be considered and object code programs will be assumed to be executed in the unprivileged mode of a processor.

Implementation of Processor Architectures

The implementation, in hardware, of a processor architecture can use techniques, such as *cache memory* or *pipelining* (Wakerly, 1989; Hennessy & Patterson, 1990), to optimise program execution. These do not affect the result of executing instructions or programs: the semantics of a processor language are constant across different implementations of the processor. For example, the Alpha AXP (Digital Equipment Corporation, 1996) processor architecture requires that object code programs appear to be executed sequentially. This permits the processor implementation to execute instructions in parallel provided that the result is the same as executing the instructions sequentially. Object code verification is concerned with the effect of executing programs and not on with how the processor architecture is implemented in hardware. The techniques used to optimise a processor implementation will therefore not be considered further.

Hayes (1988) and Hennessy & Patterson (1990) describe the design and implementation of processors as well as the factors influencing the choice of resources provided by a processor. A common assumption is that object code programs are produced by an optimising compiler for a high-level language. The designs of the Alpha AXP and of the PowerPC processors are described by Sites (1993) and Diefendorff (1994). Wakerly (1989) describes the techniques used to write object code programs, concentrating on the Motorola 68000 processor (Motorola, 1986).

2.2.1 Object Code Programs

An object code program is made up of one or more processor instructions. Formally, object code is the machine representation of a processor language program: each instruction is encoded in a machine-readable form. An *assembly language* program is the human-readable form of a processor program, in which instructions are represented symbolically. There is a mapping between an assembly language program and an object code program: an *assembler* translates from assembly language to object code; a *disassembler* translates from object code to assembly language (Hayes, 1988; Wakerly, 1989). Techniques for specifying the mapping between assembly language and object code are described by Ramsey & Fernández (1997). For verification, there is little practical difference between assembly languages and object code and no distinction will be made between an assembly language program and its machine representation. The term object code will be applied to any program of a processor language, regardless of its representation. Instructions of a processor language will be described using their symbolic forms.

Data Model

A processor operates on *values* which are represented on the machine as *bit-vectors* (see Hayes, 1988); these can be considered a subset of the natural numbers. The resources available to a processor include the *machine memory* and the processor *registers*. The machine memory is organised as *locations* indexed by *addresses*. The variables of an object code program are some subset of the memory locations (which will be called *memory variables*) and the processor registers. The range of values which can be stored in a memory variable or register is fixed. Two or more (generally consecutive) memory locations can be used to store values too large to be stored in a single location. There is at least one register known as the *program counter* (sometimes called the instruction pointer), denoted *pc*, which identifies the instruction selected for execution. The memory variables and processor registers will collectively be referred to as *program variables*. For object code verification, a machine *state* is an assignment of values to the program variables.

Processor registers are either *hidden*, and used only in the semantics of instructions, or are *visible* and may be referred to by an instruction. For example, the program counter of the PowerPC processor (Motorola Inc. and IBM Corp., 1997) is always hidden and occurs only in the instruction semantics. Registers can be organised in a number of ways. In the simplest, all registers are visible at all times and each register is referred to by a unique identifier. More complex methods include register *banks* and *windows*, in which only a subset of the registers are visible at any time (Hennessy & Patterson, 1990). These methods are based on the use of hidden registers to determine the visible registers referred to by an instruction.

Execution Model

Each instruction of an object code program is stored in one or more memory locations and is identified by its *label*, typically the address of the first memory location. Execution of an instruction begins in a state and produces a new state by assigning values to program variables. An instruction labelled l begins in a state s iff the value of the program counter in s is l; the instruction is selected for execution and said to have *control* of the machine. A program has control when any instruction of the program is selected for execution. Each instruction (implicitly or explicitly) assigns a label to the program counter, selecting the *successor* of the instruction. When execution of the instruction ends, control passes to the successor instruction. The *flow of control* through the program is the order in which program instructions are selected and executed.

The assignment to the program counter is implicit for the majority of instructions: the successor is the next instruction in memory. Instructions which explicitly select a successor are called *jumps* and pass control to a *target* instruction. The label assigned to the program counter by a jump instruction can be the result of an operation on data; in this case, the instruction is said to be a *computed jump*. Because the selection of a successor is an action performed by each instruction, the order in which instructions are executed is independent of the order in which they are stored in memory. This is in contrast to high-level languages, in which the flow of control is, generally, determined by the order in which commands appear in the program text.

2.2.2 Processor Instructions

A processor instruction performs an operation on a machine state and produces a new state by making one or more simultaneous assignments to variables. The operation performed by the instruction applies one or more functions to *source operands*. The resulting values are stored in variables identified by *destination operands*. The syntax of an instruction can have one or more *arguments*, from which the operands are calculated. A *source argument* is used to calculate source operands, a *destination argument* is used to calculate destination operands. The *addressing modes* of an instruction determine how the source and destination operands are calculated from the instruction arguments.

Example 2.1 Assume that an instruction has source argument *src*, destination argument *dst*, and performs an operation implementing a function f. The addressing mode of the source argument defines a function I_s, identifying the source operand as $I_s(src)$. The addressing mode of the destination argument defines a second function I_d to calculate the destination operand $I_d(dst)$. The result of the operation will be to store in the program variable $I_d(dst)$ the result of applying f to the source operand: $I_d(dst) := f(I_s(src))$.

Common addressing modes include *immediate*, *direct*, *indirect* and *indexed* addressing. Assume register r, value v and that the machine memory is represented by function **Mem** such that the nth location in memory is $\mathbf{Mem}(n)$. In immediate addressing, the instruction argument is a value v which is interpreted only as a source operand, $I_s(v) = v$. In direct addressing, the instruction argument identifies a program variable (a register or memory location) and the operand is the value of the variable. If the argument is register r then r is also the operand; if the argument is value v then the operand is memory variable $\mathbf{Mem}(v)$. Indirect addressing interprets a program variable as a pointer: if register r is an argument, then the operand is the memory location whose address is stored in r, $\mathbf{Mem}(r)$. Indexed addressing interprets two arguments as a memory address and an offset: if r and v are the arguments then the operand is $\mathbf{Mem}(r + v)$. □

Instruction Categories

The instructions of processor languages can be grouped according to their intended use in an object code program. An instruction can be considered as either *arithmetic*, *program control*, *data movement* or *input/output*. The arithmetic instructions perform operations on the program data. These include the arithmetic operations (addition, subtraction, etc.), comparisons, bit-vector operations and conversions between data representations. Program control instructions (jumps) pass control to a target instruction by assigning a label to the program counter. A jump can be conditional on the value of one or more variables. It can also assign values to variables other than the program counter, to support *sub-routines* (Wakerly, 1989). Data movement instructions transfer values between program variables, typically between registers and memory locations. A data movement instruction assigns the values of one or more program variables to one or more program variables.

Input/Output instructions perform essentially the same operation as data movement instructions, except that the transfer is between program variables and the input-output *ports* of the processor (see Wakerly, 1989). It is not necessary for a processor to provide input/output instructions and, when available, these instructions are often restricted to the privileged execution mode of a processor. Input/Output instructions can be considered to be instances of the data transfer instructions with the ports described by *memory-mapping* (Wakerly, 1989): input ports are modelled as functions whose value is unknown and output ports as variables whose value is never used.

Although processor languages can differ in the details, all provide instructions to carry out arithmetic, program control and data movement operations. The differences between these groups of instructions (and between processor languages) are primarily in the methods used to calculate the values and to identify the variables to be changed. Common to all groups is that every instruction makes an assignment to one or more variables and the changes made can depend on the initial values of the variables. In particular, every instruction makes an assignment to the program counter, to identify the successor of the instruction.

Example 2.2 The Alpha AXP processor language (Sites, 1992) includes an instruction, written `addl` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$, to store the sum of registers $\mathbf{r}_1, \mathbf{r}_2$ in register $\mathbf{r}_3$. The instruction also assigns the label, l, of the next instruction in memory to the program counter. This is an arithmetic instruction and can be described as a simultaneous assignment: $\mathbf{r}_3, pc := \mathbf{r}_1 + \mathbf{r}_2, l$. A similar instruction for the PowerPC processor (Motorola Inc. and IBM Corp., 1997) is written `add` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$. This stores the sum of $\mathbf{r}_2$ and $\mathbf{r}_3$ in $\mathbf{r}_1$, updating the program counter with label l: $\mathbf{r}_1, pc := \mathbf{r}_2 + \mathbf{r}_3, l$.

Comparison instructions for the Alpha AXP processor store the result of a test in a register. The instruction `cmpeq` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ tests the equality of registers $\mathbf{r}_1$ and $\mathbf{r}_2$. The result of the test, 1 or 0 representing *true* or *false*, is stored in register $\mathbf{r}_3$. The program counter *pc* is assigned the label l of the next instruction. This can be described as a conditional command: *if* $\mathbf{r}_1 = \mathbf{r}_2$ *then* $\mathbf{r}_3, pc := 1, l$ *else* $\mathbf{r}_3, pc := 0, l$. The Alpha AXP also includes a conditional jump `beq` $\mathbf{r}, v$ which tests the value of a register, $\mathbf{r}$. If the value is 1 then control passes the instruction at label $pc + v$, otherwise control passes to the next instruction in memory, at label l. This is the conditional command: *if* $\mathbf{r} = 0$ *then* $pc := pc + v$ *else* $pc := l$.

The data movement instruction `move.w` $\#258, \mathbf{r}@$ of the Motorola 68000 (Motorola, 1986) stores value 258 in the memory location identified by register $\mathbf{r}$. In the Motorola 68000 architecture, a memory location stores numbers in the range $0 \dots 255$. The number 258 is stored in two consecutive locations as the two values 1 and 3 (calculated from $258 = ((2^8-1)\times 1)+3$). Assume the machine memory is represented, as before, by **Mem**. The instruction can be described as: $\mathbf{Mem}(\mathbf{r}), \mathbf{Mem}(\mathbf{r}+1), pc := 1, 3, l$ (where l is the label of the next instruction). The first value, 1, is stored in location $\mathbf{Mem}(\mathbf{r})$ and the second value, 3, in the next location, $\mathbf{Mem}(\mathbf{r}+1)$. □

Execution of an instruction can cause an error, for instance if a division by zero is attempted. The result of an error is either the assignment of arbitrary (undefined) values to variables or the failure of the instruction, passing control out of the program. Because object code programs are interpreted by the processor, errors are detected when an instruction is executed. A processor language can therefore include instructions which fail in some circumstances, with the assumption that the instructions will not be used in those circumstances. For example, the PowerPC processor includes an instruction which simultaneously assigns values to any two registers $\mathbf{r}_1$ and $\mathbf{r}_2$. This instruction fails if the registers are not distinct (the assignment is not executable). However, the registers are compared only when an attempt is made to execute the instruction.

2.2.3 Features of Processor Languages

Although each processor architecture has a unique language, all processor languages have two basic features. The first, that every processor instruction is a state transformer: a state is produced by changing the values of one or more variables, possibly using the initial values of the variables to determine the changes to be made. Every change made to a state can be described as the simultaneous assignment of values to variables: if state s differs from state t in the values of variables $x_1, \ldots, x_n$ then s can be transformed to t by a simultaneous replacement of the values of $x_1, \ldots, x_n$ in s with the values of $x_1, \ldots, x_n$ in t. Every processor instruction is therefore be an instance of conditional and simultaneous assignment commands. Processor languages also include the equivalent of pointers (the expressions implementing addressing modes); unexecutable assignments are detected when an instruction is executed, allowing variables identified by pointers to be determined before they are compared. The second feature of processor languages is that the selection of instructions for execution is an action carried out by the instructions. In the execution model of object code programs, the flow of control is determined when instructions are executed rather than by the text of the program.

2.3 Modelling Object Code

To verify an object code program, the program and each instruction must be described in a form which can be reasoned about in a logic. To support the definition of program transformations, it must also be possible to manipulate programs and instructions. Methods of describing object code vary in the approach taken to the processor language in which the program is written. This affects the difficulty of verifying and manipulating a program.

A common approach is to consider processor languages and their instructions individually (Gordon, 1988; Yuan Yu, 1992; Necula & Lee, 1996): each instruction is considered to be distinct from any other instruction. To verify object code using this approach requires a proof rule for each instruction of a processor language. Because processors have a large number of instructions, treating each instruction as distinct from any other instruction leads to the repetition of a large amount of work. For example, proof rules defined for the instructions of one processor cannot be applied to the instructions of another processor, even if the instructions have similar behaviours. This approach also makes program transformations dependent on the individual processor language and can unnecessarily limit the possible transformations. For example, if a program has a sequence of two instructions which carry out the assignments $x := 1$ and $y := 2$ then the program could be simplified by replacing the instructions with the single assignment $x, y := 1, 2$. This transformation is not possible if the processor language does not include an instruction which carries out the multiple assignment to x and y.

An alternative approach is to describe processor instructions as commands of an abstract language. This allows the similarities in the semantics of instructions to be exploited, describing a specific action, such as an assignment to a register, as an instance of a more general action, such as a simultaneous assignment to variables. This approach also supports the definition and application of program transformations. Since the abstract language is not limited by a processor architecture, it can be defined to include any construct useful for the verification or transformation of programs. The use of an abstract language is similar to the use of intermediate languages to implement code generation and optimisation techniques (Aho et al., 1986). However, the commands of these intermediate languages only describe the actions which must be implemented by instructions. To describe an instruction in an intermediate language would need a sequence of such commands. This is not expressive enough for verification, where (ideally) a single command of the abstract language will be enough to describe the behaviour of any instruction.

2.3.1 Instructions and Data Operations

To describe processor instructions, an abstract language must describe both the data operations used by the instruction and the action performed with the results of the operations. The data operations of a processor calculate values, the labels of computed jumps, the results of tests or identify program variables. All but the last can be described in terms of the expressions which occur in any programming language (see for example Wakerly, 1989). Operations to identify program variables implement the addressing modes of a processor. These are equivalent to pointers and can be modelled in terms of arrays. Formal models of arrays and the operators (including substitution) needed to reason about arrays in programs are described by Dijkstra (1976) and Cartwright & Oppen (1981), among others.

To describe the action performed by an arbitrary processor instruction, an abstract language need only include a conditional and a simultaneous assignment command. However, a simultaneous assignment in the presence of pointers (of arrays) means that the abstract language will contain unexecutable commands, which assign many values to a single variable. It is necessary to be able to detect these commands since otherwise it is possible to verify an program which is incorrect; e.g. for partial correctness, an unexecutable command satisfies any specification. The aliasing problem means that it is impossible to distinguish syntactically between executable and unexecutable commands. If the abstract language includes both simultaneous assignments and pointers then the abstract language must also include unexecutable commands.

Unexecutable commands can be excluded if the assignment commands are restricted to a single assignment. An instruction would then be described as a sequence of (possibly conditional) single assignments to variables, which may be pointers or arrays. This approach is used in processor reference manuals (e.g. see Weaver and Germond, 1994 or Motorola, 1986) to describe the semantics of instructions. For example, the Motorola 68000 instruction `move.w` $\#258, \mathbf{r}@$ would be described as the sequence of three assignment commands $\mathbf{Mem}(\mathbf{r}) := 1$, $\mathbf{Mem}(\mathbf{r}+1) := 2$ and $pc := l$. Proof rules for the necessary commands and syntactic constructs forming sequences of commands are given by Hoare (1969) and Dijkstra (1976) and others. However, this approach makes the task of verifying the program more difficult: the number of commands needed to describe an instruction will be proportional to the number of assignments made by the instruction. If each instruction of an object code program makes an average of two assignments to variables then the abstract program describing the object code will have twice as many commands as the object code. The work needed to verify the program will be correspondingly increased.

To describe an instruction using a single command, the abstract language must include a simultaneous assignment command which takes into account the presence of pointers. The assignment command will be executable only if all variables assigned to by the command are distinct. The variables identified by pointers must be determined when the command begins execution. The distinctness of the variables will therefore be a precondition on the state in which the assignment command is executed. This approach is used by Cartwright & Oppen (1981) to define a multiple assignment command for arrays. However, this assignment command is interpreted as a sequence of single assignments, each of which must be considered individually in a proof of correctness. Generalising the approach to simultaneous assignments will make it possible for an abstract language to describe arbitrary instructions as a single command and to detect the unexecutable commands during the course of a proof.

2.3.2 Program Execution

The description of an object code program must model the selection and execution of the commands representing the program instructions. Two approaches are commonly used: the first embeds the object code in an *iteration command*, which repeatedly selects and executes commands. The second describes object code as a program of a language with a similar execution model. Both methods are intended to overcome the difficulties of reasoning about a program in which commands can be arbitrarily selected for execution. However, the choice of method can affect the ease which programs are transformed and verified.

Embedding in an Iteration Command

The execution model of an object code program can be describing in terms of an iteration command which repeatedly executes instructions selected by the value of a program counter (Back et al., 1994; Fidge, 1997). Assume *do* $b_1 \rightarrow c_1 \mid \ldots \mid b_n \rightarrow c_n$ *od* is an iteration command which repeatedly selects and executes the commands $c_1, \ldots, c_n$, until every test b_i, $1 \leq i \leq n$ is *false*. A command c_i is selected if its test b_i is *true*, if more than one test is *true* then the choice is arbitrary (Gries, 1981). Also assume that every instruction C_i of the object code program, with label l_i, is described by a command c_i of the abstract language with test *pc* $= l_i$. Since each instruction is stored in a unique location, and has a unique label, no two tests can be *true* simultaneously. The object code program can then be modelled as the command *do pc* $= l_1 \rightarrow c_1 \mid \ldots \mid$ *pc* $= l_n \rightarrow c_n$ *od*. This executes a command c_i only if it is selected, the precondition *pc* $= l_i$ is *true*. Proof rules for reasoning about the selection and execution of instructions and for the behaviour of a program can be derived from the rules for the iteration command, given by Hoare (1969) and Dijkstra (1976).

Embedding the object code in an iteration command models object code in two distinct parts. Instructions are modelled directly, as commands of the language, while object code programs are modelled indirectly, by the iteration command. To consider a single instruction of the object code program it is necessary to consider the whole of the iteration command, which is made up of all instructions of the object code. This makes it difficult to concentrate on a subset of a program. For example, to consider a loop in an object code program, which may be made up of a few instructions, it is necessary to consider the entire iteration command describing the object code. The number of commands in a typical object code program make this approach too unwieldy to be practical

Object Code as Flow-Graph Programs

A more natural model for object code is as a program of a *flow-graph* language (Loeckx & Sieber, 1987). A flow-graph program is made up of a set of commands and is executed by the repeated selection and execution of the commands. Since the execution model of a processor language is that of a flow-graph language, a flow-graph program can model object code directly. Program logics for flow-graph programs are often based on a temporal logic (Manna & Pnueli, 1981) although simpler logical systems have also been used (e.g see Loeckx & Sieber, 1987; Gordon, 1994a). However, the flow-graph languages which have been considered in verification are less expressive than processor languages. This causes problems when defining proof rules to specify the execution model of processor languages. These rules must permit reasoning about the transfer of control between commands and are characterised by the treatment of a *jump* command, written *goto*, where *goto* l does nothing except pass control to the command labelled l.

Proof rules for jump commands generally follow those of Clint & Hoare (1972) and de Bruin (1981). These interpret the jump as a construct which passes control to a target but which does not terminate: the jump to label l, *goto* l, satisfies the specification $\{P\}\mathit{goto}\ l\{\mathit{false}\}$ for any P. This leads to proof rules for partial correctness only, total correctness requires the *goto* to terminate. The interpretation is *false* for processor languages, in which all instructions terminate: if *goto* l describes a processor instruction then it must satisfy $\{P\}\mathit{goto}\ l\{\mathit{pc} = l\}$. An alternative interpretation is given by Jifeng He (1983), based on a *generalised wp* function *gwp*. This requires that a command both terminates, establishing a postcondition Q, and passes control to a specified label. However, the *gwp* function separates the flow of control from the program variables, which include the program counter *pc*. As a consequence the *goto* command satisfies the specification: $\mathit{pc} \neq l \Rightarrow \mathit{gwp}(\mathit{goto}\ l, \mathit{pc} \neq l)$. This specification of a jump is *false* for processor languages: a jump instruction must assign the label of the target to the program counter, satisfying $\mathit{gwp}(\mathit{goto}\ l, \mathit{pc} = l)$.

A simple approach to modelling and specifying object code programs can be based on the use of the program counter *pc* to select instructions. Assume that command c describes the action of an instruction at label l. The instruction is selected when $\mathit{pc} = l$, the command c must therefore be associated with label l. Assume that $(l : c)$ labels command c with l and that the weakest precondition of $(l : c)$ satisfies $\mathit{wp}(l : c, Q) \Rightarrow \mathit{pc} = l$. This is enough to model the selection of the instruction. Because the program counter *pc* is a variable, a jump instruction only needs to make an assignment to *pc*. A jump command is therefore an instance of an assignment command: $(goto\ l) = (pc := l)$. This allows proof rules specifying the transfer of control between commands to be derived from the proof rules for the assignment command. Using this approach, an object code program can be described by a set of commands of the abstract language. This allows the manipulation of subsets of the program, providing flexibility in the methods used to reason about and transform the program.

2.3.3 Verification and the Language $\mathcal{L}$

Since the abstract language $\mathcal{L}$ will be used to describe object code programs, it must be expressive enough to describe the data operations of a processor language, the actions performed by an instruction and the execution model of object code. This can be achieved if the language $\mathcal{L}$ is a flow-graph language with pointers, a simultaneous assignment, a conditional command and a construct for labelling commands. The data operations of a processor will be modelled as expressions of $\mathcal{L}$, which must include pointers. The presence of pointers will also require a method of detecting unexecutable assignment commands. The execution model of $\mathcal{L}$ must follow that of processor languages, with the value of a program counter used to select program commands. By satisfying these requirements, it will be possible to describe any object code program of any processor language by a program of $\mathcal{L}$ with an equal number of commands.

The language $\mathcal{L}$ must also support the definition of proof rules, for programs and commands. All required proof rules, except those for the assignment command, can be described in terms of standard logical operators. Proof rules for the assignment command of $\mathcal{L}$ require a substitution operator which takes into account the presence of pointers. This substitution operator must be provided as part of the language $\mathcal{L}$. This model for the language $\mathcal{L}$ will be enough to describe an object code program in a form which can be verified. However, the $\mathcal{L}$ program describing the object code will be as difficult to verify as the object code, since the number of commands in the $\mathcal{L}$ program is equal to the number of instructions in the object code. To simplify the verification, the program of $\mathcal{L}$ must be transformed to reduce the number of commands to be considered during verification. The method used to transform programs of $\mathcal{L}$ places additional requirements on the language $\mathcal{L}$.

2.4 Program Refinement and Abstraction

The work needed to verify a program is determined by the proof methods for program correctness. These require that each sequence of commands between cut-points of the program is shown to satisfy an intermediate specification by reasoning about the individual commands in a sequence. The approach to simplifying verification is based on transforming a program to construct an *abstraction* of the program. This abstraction will allow a sequence of commands in the program to be shown to establish its intermediate specification directly, rather than by considering each command in the sequence individually.

The basis for program abstraction is the relationship between specifications and programs.

Both are descriptions of a computer's behaviour at different levels of abstraction. A specification is more abstract than a program, describing properties of the behaviour but not how the behaviour is produced. A program is a *concrete* specification, describing how the computer behaviour is produced but not the properties of the behaviour. Considering specifications, and therefore programs, at different levels of abstraction allows the result of a program transformation to be compared with the original program. The transformation constructs an abstraction of a program only if the result of the transformation and the original program describe the same behaviour.

2.4.1 Refinement Calculus

The relationship between specifications at different levels of abstraction is the subject of the refinement calculus (Morris, 1987; Back & von Wright, 1989; Morgan, 1990). A partial order $\sqsubseteq$ is defined between specifications S_1 and S_2. If $S_1 \sqsubseteq S_2$ then S_2 describes a system with at least the properties described by S_1. The specification S_2 may be more concrete than S_1, giving more detail as to how the system is to be implemented and either of S_1 or S_2 can be a program. When S is a specification and p a program, the ordering $S \sqsubseteq p$ states that p is a correct implementation of S: every property described by S is a property of the behaviour produced by p. Programs can also be compared by refinement: a program p_1 is an abstraction of a program p_2 if any property established by p_1 is also established by p_2. The programs satisfy the ordering $p_1 \sqsubseteq p_2$ and the program p_1 is a description of the behaviour of program p_2.

Verifying that a program p satisfies a specification S is equivalent to proving that p refines S, $S \sqsubseteq p$. Refinement is transitive (Back & von Wright, 1989): if $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_3$ then $S_1 \sqsubseteq S_3$ for any specifications S_1, S_2 and S_3. This justifies the verification of a program p by verifying an abstraction p' of p, $p' \sqsubseteq p$. To show that program p is a refinement of specification S, it is only necessary to show the p' is a refinement of S, $S \sqsubseteq p'$. This establishes the ordering:

$$S \sqsubseteq p' \sqsubseteq p$$

The correctness of p then follows by the transitivity of refinement: $S \sqsubseteq p$.

The refinement calculus is often used to justify the correctness of program transformations. These are typically used in compilers, where an high-level (abstract) program p_h is refined to a low-level program p_l by replacing each command in p_h with a sequence of low-level commands. This is the reverse of the approach needed to simplify verification, where a transformation must construct an abstract program p_h from a low-level program p_l by replacing sequences of commands in p_l with a single command in p_h.

2.4.2 Compilation

A compiler translates a high-level program p_h into a processor language, producing an object code program p_l which is executable on a target machine. The compiler is a program transformation T_c which, applied to p_h produces the object code program p_l: $T_c(p_h) = p_l$. The compilation is correct if the object code correctly implements the high-level program, $p_h \sqsubseteq p_l$. In a provably correct compiler, the translation of the program p_h to p_l always satisfies the refinement ordering: $p_h \sqsubseteq T_c(p_h)$ for any program p_h (Hoare et al., 1993; Sampaio, 1993; Bowen and He Jifeng, 1994). The use of correct compilers is an alternative to object code verification. The approach is to verify a high-level program and then to use a correct compiler to produce the (correct) object code. Verifying the high-level program will be simpler than verifying the object code since high-level programs have fewer commands and are easier to reason about than the equivalent object code programs.

To prove the correctness of a compiler, the translation from the high-level language to the processor language must be shown to be correct. This requires a semantics for both the high-level and the processor languages. Since many high-level languages do not have a completely defined semantics, correct compilers are often implemented for a subset of a language only (e.g. see Polak, 1981). The need to verify the translation also limits the code generation techniques which can be used. In particular, code optimisation techniques cannot easily be applied. Bowen & He Jifeng (1994) describe a correct compiler which optimises the object code but the optimisation technique used is simple in comparison with those of standard compilers (see Aho et al., 1986). This is a serious drawback since processor languages are often designed with the assumption that object code will be produced by optimising compilers (Sites, 1993; Weaver and Germond, 1994).

Correct compilers are expensive to build: the proof of correctness must be repeated for each combination of a high-level language and target machine. Their advantage is that once a correct compiler is built, only the high-level program must be verified. This is easier than verifying object code and may justify the cost of verifying the compiler and using less efficient object code. However, object code programs are not necessarily produced by a compiler and correct compilers do not solve the problem of how to verify an arbitrary object code program.

2.4.3 Abstraction

Abstraction is the reverse of compilation. Given a program p_l, abstraction constructs a program (or specification) p_h such that the ordering $p_h \sqsubseteq p_l$ is satisfied. A transformation T_a abstracts from a program p if the refinement ordering $T_a(p) \sqsubseteq p$ is satisfied. The transformation correctly abstracts from programs if the ordering $T_a(p) \sqsubseteq p$ is satisfied for any program p. The result of the transformation $T_a(p)$ can be a specification, a program of the same language as p or a program of a different language. This allows a wide range of methods for constructing program abstractions, not all of which are useful. For example, the transformation which does nothing, $T_a(p) = p$, constructs an abstraction of p but does not simplify the verification of p.

Methods for program abstraction include the reconstruction of high-level programs from the object code produced by a compiler. Breuer & Bowen (1994) applied the translation rules of a compiler to an object code program in reverse, to obtain the original high-level program. This method is limited: the object code must be produced by the compiler and the method fails when code optimisation is used. Pavey & Winsborrow (1993) used a similar technique to show that the compilation of a high-level program was correct. Sequences of commands in the object code program were matched to commands of the original high-level program by analysing the flow of control through both programs. Although an automated tool was used, it was necessary to perform part of the analysis manually and neither program was verified. As before, the technique cannot be applied to the object code produced by an optimising compiler.

More general methods for abstracting programs have been developed. *Symbolic execution* is a technique based on the semantics of programs (King, 1976; Clarke & Richardson, 1981) which is used to analyse programs for code optimisation. Methods based on the text of a program can be derived from the rules of a calculus, described by Hoare et al. (1987), for manipulating programs. This approach is similar to the method which will be used to abstract programs of $\mathcal{L}$.

Symbolic Execution

The symbolic execution of a program specifies the behaviour produced by the program as a relation between the initial and final values of the program variables (King, 1976; Clarke & Richardson, 1981). The relations are constructed by combining the boolean tests of the conditional commands with assertions describing the expressions assigned to the variables by the assignment commands. The flow of control through a program determines the order in which program commands are considered. This is limited to finite sequences of program commands for which the execution order can be determined, disallowing the use of computed jumps. Symbolic execution can be applied in the presence of pointers and Colby (1996) describes a technique which considers programs with multiple assignments to pointers. The aliasing problem is solved by constructing a relation with a series of comparisons between pointers. The properties described by the relation depend on the result of evaluating the comparisons between the pointers.

The use of symbolic execution in object code verification is limited since object code can include an arbitrary number of computed jumps. Furthermore, the relations constructed by symbolic execution describe the semantic properties of a sequence of commands. For object code, these relations will be large since the properties of instructions can be complex. The work needed to verify a program using these relations will be therefore be larger than needed for verification in a program logic, where proof rules are applied to the syntax of commands. The separation of semantics and syntax in a program logic also means that the formulas occurring in a proof are simpler than the relations built up by symbolic execution. In a program logic, only the properties required to prove the specification need be considered; in symbolic execution, all the semantic properties of instructions are described. In addition, the syntactic approach allows automated proof tools to perform simplifications on commands more efficiently than would be possible when considering logical relations.

2.4.4 Abstraction by Program Manipulation

The abstraction of a program can be constructed by manipulating the text of the program. The basis for this approach is the ability to manipulate and abstract the commands of the programming language. Hoare et al. (1987) describe a set of algebraic rules defining relationships between different combinations of commands. These rules can be applied to commands c_1 and c_2 to construct a new command c'. This command is an abstraction of c_1 and c_2: the effect of executing c' is the same as executing c_1 followed by c_2. Since c' is a single command, replacing c_1 and c_2 with c' in a program will simplify the verification of the program: only the command c' must be considered to establish the properties of c_1 followed by c_2. The algebraic rules of Hoare et al. (1987) are defined on the syntax of the commands and can be applied mechanically. This makes it possible to efficiently implement abstracting transformations as proof tools for verification, in the same way as compiling transformations can be implemented as compilers for high-level languages.

The rules of Hoare et al. (1987) are defined for a structured language which includes a simultaneous assignment command but does not include pointers or jumps (computed or otherwise). Both pointers and computed jumps make application of the rules impossible. Pointers affect the abstraction of assignment commands, which is based on merging the list of variables to which assignments are made and substituting values for variables. For example, assume x, y, z are variables and $e_1, \ldots, e_4$ are expressions. The abstraction of the two assignment commands $x, y := e_1, e_2$ and $y, z := e_3, e_4$ (executed in sequence) results in the command:

$$x, y, z := e_1, e_3[e_1, e_2/x, y], e_4[e_1, e_2/x, y]$$

The abstraction of the two assignment commands is a single command in which the assignments to variables x, y and z are carried out simultaneously. The expressions e_1 and e_2 assigned to x and y by the first command are substituted for the variables in the expressions of the second command. The variable y which occurs in both commands is assigned $e_3[e_1, e_2/x, y]$, since the second assignment to y supersedes the first. The occurrence of y in both assignment commands is determined syntactically, by comparing the variables occurring in the commands. When the assignment commands include pointers, the aliasing problem means that it is not possible to merge the lists of variables since the variables cannot be compared syntactically.

Computed jumps cause similar problems. An abstraction of commands c_1 and c_2 can be constructed only if it is known that control will pass from c_1 to c_2. When c_1 is a computed jump, it is undecidable whether c_2 is selected by c_1: the target of a computed jump cannot be determined from the syntax of the jump command. Both pointers and computed jumps cause difficulties because the rules of Hoare et al. (1987) require information which is undecidable from the text of commands in a language which includes pointers or computed jumps. This means that the rules of Hoare et al. (1987) cannot be applied directly programs of the language $\mathcal{L}$, which must contain both pointers and computed jumps to be expressive enough to model object code.

The advantage of abstracting a program by manipulating the program text is its efficiency. Because only the program text is considered, an abstraction of the program can be constructed more easily than is possible when abstraction is based on the semantics of the programming language. In addition, transformations which manipulate the text of a program can be efficiently mechanised (this is the basis of compilers). This allows the straightforward implementation of tools to construct program abstractions. Although the rules of Hoare et al. (1987) cannot be applied to the programs of $\mathcal{L}$, more general rules can be defined which take into account the presence of pointers and computed jumps.

2.4.5 Abstraction and the Language $\mathcal{L}$

The abstraction of $\mathcal{L}$ programs will be based on abstracting commands of $\mathcal{L}$ using an approach similar to that of (Hoare et al., 1987). There are three problems to be overcome: the use of substitution, the manipulation of lists of assignments in the presence of pointers and the treatment of computed jumps. These must be solved using syntactic operations only, to ensure that only the syntax of a program is needed to construct its abstraction. The syntactic operations must be provided by the language $\mathcal{L}$, which must also include a simultaneous assignment command (to permit the abstraction of assignment commands). To abstract from programs of $\mathcal{L}$, it is also necessary to identify the sequences of commands between cut-points in the program. The program abstraction will then be constructed by abstracting from each sequence of commands.

The language $\mathcal{L}$ is a flow-graph language and a program of $\mathcal{L}$ does not have a syntactic structure which identifies the loops in the program. The cut-points of a program must therefore be selected manually or by an automated analysis of the flow-graph of the program (Hecht, 1977; Aho et al., 1986). Because, processor language often support code optimisation (Weaver & Germond, 1994) (and therefore flow-graph analysis), the use of such techniques will allow a high degree of automation when abstracting a program. Although these techniques do not provide a general solution, which can be applied to any program, they allow the combination of automated and manual reasoning to simplify program verification.

2.5 Automated Tools

A number of techniques have been used to automate program analysis and verification. Fully automatic methods include static analysers, which use the program text together with the semantics of the language to predict the behaviour of a program. These can be used to automatically analyse sections of a program, although information must often be supplied manually (Pavey & Winsborrow, 1993). Model checking (Clarke et al., 1986; Clarke et al., 1994) is based on generating all states that will be produced by a program. These are then verified to satisfy intermediate assertions which lead to the specification of the programs. However, the large number of states which are generated means that the size of the program which can be verified is severely restricted.

General purpose theorem provers have been widely used in the verification of systems, using a combination of automated and manual proof. Many theorem provers are based on higher order logic (Church, 1940) or on set theory, both of which are expressive enough to define the logics used in program verification. Gordon (1988) describes the definition of Hoare logic on the HOL theorem prover (Gordon & Melham, 1993). The HOL theorem prover has been used for a number of studies; Melham (1993) describes its use in hardware verification and it was also used in the SafeMOS project (Bowen, 1994). Other systems include PVS (Owre et al., 1993), a theorem prover for higher order logic, and Isabelle (Paulson, 1994a), a generic theorem prover.

The techniques used in automated theorem proving are described by Bundy (1983) and Duffy

(1991). The methods relevant to verification and abstraction include decision procedures, rewriting techniques and simplification (Duffy, 1991; Boulton, 1994). Decision procedures allow the truth of a formula to be determined by a machine. Many of the formulas which occur in program verification are concerned with arithmetic properties and Shostak (1977) describes a decision procedure, based on work by (Bledsoe, 1974), for such formulas. A decision procedure for bit-vectors is described by Cyrluk et al. (1997) and procedures also exist for quantifier free logical formulas (Moore, 1994). Rewriting techniques implement the rules for equality, allowing the replacement of a term with its equivalent (if $x = y$ then $P(x) = P(y)$, Duffy, 1991). Simplification procedures apply the rules of a logic, possibly with rewriting and the use of decision procedures, to reduce the size of a formula which must be proved manually.

Theorem provers can carry out a large part of the simplification and basic reasoning needed to show that a command satisfies a specification. However, completely automating program verification is not possible. To show that a program establishes a postcondition may require a proof that a variable will be assigned a value or that a command will eventually be selected. Both the variables referred to by pointers and the targets of computed jumps are undecidable: the steps taken to prove such properties must be determined manually. Reasoning about the loops in a program is also difficult to automate. A loop is shown to establish a property by reasoning about an assertion which is invariant for the loop (Dijkstra, 1976). The automatic construction of such an invariant is computationally difficult (Wegbreit, 1977), it is simpler to supply the invariant manually. In general, it is more efficient to verify a program by a combination of automated and manual reasoning. In this approach, the method by which a property is to be proved and the steps to be taken are determined manually. When it is known how the proof is to proceed, as many of the steps in the proof as possible are carried out mechanically, by applying simplification and decision procedures. This method of automating proofs is described by Milner (1984).

2.5.1 Processor Simulation

An automated proof tool for verifying the object code programs for the Motorola 68000 processor (Motorola, 1986) was built by Yuan Yu (1992); a brief description is given by Boyer & Yuan Yu (1996). The tool uses the NQTHM theorem prover (Boyer & Moore, 1979) to simulate the processor as it executes an object code program. Although the NQTHM theorem prover is fully automated, the techniques used are based on a database of intermediate lemmas which are formulated manually. Since the choice of lemmas and the order in which they are proved can determine the success of a proof attempt, the proof is carried out under manual guidance (Yuan Yu, 1992).

The processor simulation is based on an interpreter for the processor language, defined as a function in the logic of the theorem prover, (Boyer & Moore, 1997). The interpreter simulates the processor accurately enough to be able to execute an object code program and to produce the same result as executing the program on the processor. A program is verified by showing that the result of applying the interpreter to the program produces the properties required by the specification. Processor simulation is a semantic approach to verifying a program. An object code program is proved correct by reasoning, directly, about the states produced by the interpreter function. This avoids many of the problems associated with processor languages, since an object code program is considered to be data to which the interpreter function is applied. This approach to reasoning about object code is used by McCarthy & Painter (1967) among others (e.g. Young, 1989; Bowen & He Jifeng, 1994) to prove the correctness of compilers. The approach has also been used to define processor languages (Windley, 1994; Müller-Olm, 1995) and in the verification of systems other than processors (Bevier et al., 1989).

The disadvantage of simulation is that instead of verifying a program p, the verification is of the interpreter applied to p. This involves reasoning about how an instruction is interpreted as well as the effect of executing the instruction. Verifying a processor simulation will therefore be more complicated than verification in a program logic, which considers only the properties required by the specification. Simulation also limits the transformations which can be applied to a program requiring an object code program transformation to replace instructions with other instructions. This can prohibit simplifications such as the replacement of two assignment commands with a single assignment, if the result is not a processor instruction. In particular, it prohibits the abstraction of commands using methods such as those of Hoare et al. (1987).

Although processor simulation has been used to verify object code programs, the level of detail required means that it is difficult to apply to large programs. The detail required also means that it is difficult to combine manual with automated reasoning, as is possible when verifying object code in a program logic. A program logic allows verification to consider only the properties of the object code program needed to establish the specification. This reduces the detail which must be considered in a proof of correctness, making verification easier to carry out.

2.6 Conclusion

The techniques for program verification provide a framework in which the properties of a program can be compared with those required by its specification. The correctness of a program is established using either the method of inductive assertions or the method of intermittent assertions. These break down the specification of the program into assertions to be established by sequences of commands in the program. The work needed to apply the proof methods is proportional to the number of commands in the program. However, the work required by the proof methods is proportional to the number of loops in the program (which determine the number of cut-points). Mechanically abstracting from the sequences of commands making up loops in the program can therefore reduce the manual work needed to verify a program.

There are two approaches to verifying and abstracting programs: either by using the semantics of the programming language directly or by defining rules using the syntax of the language. The semantic approach is used in processor simulation and symbolic execution. It can be applied to any program but requires a large amount of detail to be considered during the course of a proof. In the syntactic approach, a program is verified and abstracted by applying rules to the text of the program. This approach is efficient since only the details needed to verify or abstract a program must be considered. However, the methods developed using this approach can only be applied to a restricted class of program (which excludes object code). Because of the size of object code programs, it is impractical to use the semantic approach to verify or abstract object code. It is therefore necessary to develop methods for verifying and abstracting object code programs based on the text of a program.

The abstract language $\mathcal{L}$ is intended to model arbitrary object code programs in a form suitable for verification and abstraction based on the text of a program. The language $\mathcal{L}$ must describe the execution model of object code programs as well as the behaviour of processor instructions. To do this, the language $\mathcal{L}$ will be a flow-graph language which includes pointers, computed jumps and simultaneous assignment commands. The language will also include a conditional command and a method of associating labels with commands. This is enough to model any object code program as a program of $\mathcal{L}$ with an equal number of commands, ensuring that the translation to $\mathcal{L}$ does not increase the difficulty of verification. Proof rules for $\mathcal{L}$ can be defined using the standard logical operators together with a substitution operator (for the assignment commands). The presence of pointers means that the substitution operator must take into account the aliasing problem between variables. In addition, a method is required to detect the unexecutable assignment commands, which cannot be excluded from $\mathcal{L}$ (because of aliasing).

Program abstraction using the program text can be efficiently mechanised and results in a program which can be verified using the same methods as any other program. Because the language $\mathcal{L}$ contains pointers and computed jumps, information needed to abstract from a program cannot be determined from the text of the program. For example, the target of a computed jump and the variable referred to by a pointer are both undecidable. To support program abstraction, the language $\mathcal{L}$ must provide syntactic constructs which describe the operations needed for abstraction. These operations include determining the target of a jump, substitution in the presence of pointers and merging lists of assignments to pointers. The abstraction of a program will be constructed by program transformations which form and abstract the sequences of commands between program cut-points. Defining these transformations for programs of $\mathcal{L}$ will allow their use to abstract any object code program. Because the language $\mathcal{L}$ can model arbitrary object code programs, the methods used to verify and abstract $\mathcal{L}$ programs can be used to verify and abstract the object code programs of any processor language.

The development of the language $\mathcal{L}$ will be described in two parts, beginning with the expressions and commands of $\mathcal{L}$. These must be expressive enough to model processor instructions and must also support the abstraction of sequences of $\mathcal{L}$ commands. The methods for abstracting and reasoning about commands of $\mathcal{L}$ will also be described. The second part is concerned with the programs of $\mathcal{L}$, which model object code programs. The main interest in this part is to use the method for abstracting from sequences of commands to define transformation which abstract from programs. These transformations must also be shown to be preserve the correctness of a program, to allow their use in program verification.

Chapter 3

Commands

Program verification and abstraction is based on reasoning about and manipulating program commands. Since an object code program is made up of processor instructions, it is necessary to be able to reason about and abstract from processor instructions. Basing methods for verifying and abstracting programs on the instructions of an individual processor limits the methods to that processor and imposes unnecessary constraints on the methods used for program abstraction. To allow verification and abstraction of object code programs to be independent of a particular processor, instructions are modelled by commands of the language $\mathcal{L}$. Reasoning about an instruction is therefore based on its model in terms of $\mathcal{L}$, requiring the ability to define proof rules which can be applied to $\mathcal{L}$ commands. Abstraction is carried out by manipulating the commands of $\mathcal{L}$ and requires the ability to manipulate the text of $\mathcal{L}$ commands.

To ensure that the use of the language $\mathcal{L}$ does not increase the difficult of verifying a program, the language $\mathcal{L}$ must be expressive enough to model any instruction by a single $\mathcal{L}$ command. An instruction is described in terms of the action it performs with the result of evaluating one or more data operations. The expressions of the language $\mathcal{L}$ will model the data operations of a processor and include the equivalent of pointers, to describe the addressing modes of a processor. The commands of $\mathcal{L}$, made up of conditional commands and simultaneous assignments, will model the action performed by a processor instruction. Instructions implement the execution model of processor languages. The execution model of $\mathcal{L}$ is similar to that of processor languages: each $\mathcal{L}$ command includes the selection of its successor. This is based on a program counter, to identify the selected commands, and means the language $\mathcal{L}$ includes computed jumps.

The abstraction of commands will be defined by a function, on the syntax of $\mathcal{L}$ commands, which constructs the abstraction of a pair of commands. The abstraction of a sequence of commands can then be obtained by repeated application of this function to commands of the sequence. The method used for abstraction is similar to that of Hoare et al. (1987): the principal operations required are the ability to combine the assignments made by commands and to describe the effect of the assignments on the expressions occurring in a command. To allow abstraction to be carried out on the text of commands, the operations needed for abstraction must be described syntactically. Because the language $\mathcal{L}$ includes pointers, these operations must take into account the aliasing problem (which is undecidable). The abstraction of commands must also take into account the presence of computed jumps, which mean that is is undecidable whether two commands will be executed in sequence.

Verification is based on specifying commands and defining proof rules which can be applied to reason about the specification of commands. The operations required to define proof rules are the standard logical operators together with a substitution operator for the expressions of $\mathcal{L}$. As an example of proof rules for the commands of $\mathcal{L}$, a logic similar to the *wp* calculus (Dijkstra, 1976) will be defined. This will be based on a simple first order logic in which commands of $\mathcal{L}$ are specified by a *wp* predicate transformer. The proof rules for the logic will allow reasoning about commands specified in terms of the *wp* function. The ability to define these proof rules ensures that the language $\mathcal{L}$ provides the necessary support for specifying and reasoning about commands in a system of logic.

This chapter is structured as follows: the expressions of $\mathcal{L}$ are described, in **Section 3.1**, in two parts. First, the syntax and semantics of the expressions needed to model the data operations of processor languages are described. This is followed by the definition of the substitution expressions needed to abstract and reason about commands of $\mathcal{L}$. The syntax and semantics of the commands of $\mathcal{L}$ are defined in **Section 3.2** and followed, in **Section 3.3** by the method used for the abstraction of commands. The specification of commands and their proof rules are described in **Section 3.4**. The chapter ends with examples of the verification and abstraction of commands.

3.1 Expressions of $\mathcal{L}$

Much of the complexity of a processor language is due to the data operations provided by the processor. These data operations will be modelled by expressions of $\mathcal{L}$. The data operations fall into two classes: the first calculate a value by applying functions to data items. This includes the arithmetic and comparison operations and the operations used to calculate the target of a computed jump. This class of data operation can be described in terms of functions and constants, as is usual in programming languages (Loeckx & Sieber, 1987; Gries, 1981). The second class of data operation identifies the variables available to a program, implementing the addressing modes of a processor. These data operations, which will be referred to as *memory operations*, are equivalent to the pointers or arrays of high-level languages.

Memory Operations

The memory operations of a processor can be described in terms of pointers or arrays. However, the commonly used models of pointers and arrays complicate the definition and use of expressions. A common approach, used by Dijkstra (1976) and others, considers an array a as a variable which identifies a function f from integers to data items: the ith element of the array, $a(i)$, is $f(i)$. Assignments to the array update the entire function: the assignment of x to $a(i)$, $a(i) := x$, is interpreted as the assignment $a := \mathit{assign}(a, i, x)$, where $\mathit{assign}(a, i, x)$ is a function such that *assign*$(a, i, x)(j)$ is $f(j)$ when $i \neq j$ and x when $i = j$. This approach complicates the definition of a language, which must consider both simple variables (in which a single data item is stored) and the array variables.

An alternative approach is described by Manna & Waldinger (1981): variables are associated with memory locations, in which either data items or the addresses of memory locations can be stored. A variable x refers to a second variable y if the memory location associated with x identifies the memory location associated with y. The model used by Manna & Waldinger (1981) does not permit the use of expressions to identify program variables. The approach of Cartwright & Oppen (1981) combines the models of arrays used by Dijkstra (1976) and Manna & Waldinger (1981) with an array identifying a memory location in which a function from values to values is stored. Because this model considers arrays to be distinct from the simple variables, it also requires two separate approaches to reasoning about variables, complicating the expressions of a language.

In the approach used for the language $\mathcal{L}$, memory operations of a processor will be modelled as expressions identifying variables. These expressions will be defined in terms of functions and constants. The constants are the identifiers of variables (the registers and memory locations) while the functions allow identifiers to be calculated from values. Since variables are used to store values, this is enough to provide pointers to variables and therefore to model the memory operations of a processor language. This model is also consistent with the data operations used to calculate values, allowing expressions which identify variables to be treated in the same way as expressions which result in values.

Support for Verification and Abstraction

As well as modelling processor data operations, the expressions of $\mathcal{L}$ must support the abstraction and specification of commands. Both require a substitution operation on expressions of $\mathcal{L}$, to describe the changes made to the values of variables by the execution of commands. Because the expressions of $\mathcal{L}$ include the equivalent of pointers, the substitution operator of $\mathcal{L}$ must take into account the presence of pointers. For the abstraction of commands (which uses a method similar to that of Hoare et al., 1987), the expressions of $\mathcal{L}$ must also support the combination of the lists of assignments made by a command. The expressions of $\mathcal{L}$ will support both substitution and the combination of assignment lists by syntactic constructs whose semantic interpretation carries out the required operations. This will allow the verification and abstraction of commands to be carried out using only the syntax of expressions and commands.

The description of $\mathcal{L}$ expressions begins with the basic model, describing the constants and functions from which expressions are constructed. The expressions of $\mathcal{L}$ are then built up in two parts. First, the expressions modelling the data and memory operations are described. These are used to define equivalence relations between expressions; which are required for substitution in the presence of pointers. In the second part, the substitution operator of $\mathcal{L}$ is defined. This is based on a data type which represents the assignment lists of commands and allows the combination of assignments lists to be described syntactically.

3.1.1 Basic Model

The basic model of the expressions determines the constants and functions which can be used in expressions of $\mathcal{L}$ to model the data items and operations of a processor language. The data items of an object code program are represented by a set of *values*. A value can be stored either in a processor register or in a memory location. The *registers* are a set of symbolic names identifying processor registers; the *memory variables* are the subset of the values containing the addresses of the memory locations in which a data item can be stored. The *labels* are a subset of the values identifying the memory locations in which program instructions can be stored. The labels are distinct from the memory variables (prohibiting self-modifying programs). An interpretation of values as booleans will be assumed, to allow tests on the values of variables to be defined in terms of expressions which result in values.

The values and memory variables, *Values* and *Vars*, serve different purposes. A value is a data item while a memory variable is the address of a location in which a value can be stored. Because *Vars* is a subset of *Values*, it is necessary to be able to distinguish between them. It is also convenient to combine the sets of registers and variables into a single set of variable *names*, since there is no distinction between the way in which a register and a memory variable are used. The names are distinct from the values and identify all variables which can occur in a processor instruction. There is at least one name, the *program counter pc*, which will be used in the semantics of the $\mathcal{L}$ commands to select commands for execution.

Definition 3.1 *Constants*

Values is a set of some type T containing the data items which can be represented by the processor. The set *Regs* contains the register identifiers. The sets $\mathit{Vars} : \mathit{Set}(\mathit{Values})$ and $\mathit{Labels} : \mathit{Set}(\mathit{Values})$ are subsets of the values identifying locations of the memory variables and the commands.

$$\mathit{Vars} \subseteq \mathit{Values} \qquad \mathit{Labels} \subseteq \mathit{Values}$$

The set *Names* contains elements constructed by the application of a function, *name*, to the elements of the sets of variables and registers. Function *name* has type $\mathit{name} : (\mathit{Vars} \cup \mathit{Regs}) \to \mathit{Names}$ and satisfies, for $x, y \in (\mathit{Vars} \cup \mathit{Regs})$:

$$\mathit{name}(x) \in \mathit{Names} \qquad x = y \Leftrightarrow \mathit{name}(x) = \mathit{name}(y)$$

The names are distinct from the values: $\mathit{Names} \cap \mathit{Values} = \{\}$. There is a name $\mathit{pc} \in \mathit{Names}$.

There are at least two values, $\mathbf{true}, \mathbf{false} \in \mathit{Values}$, representing the Boolean *true* and *false* and a boolean interpretation $\mathcal{B}$ of type $\mathit{Values} \to \mathit{boolean}$ satisfying:

$$\mathcal{B}(\mathbf{true}) \qquad \neg\mathcal{B}(\mathbf{false})$$

□

The definition of the basic model will depend on the processor languages to be modelled in terms of $\mathcal{L}$. The sets *Values*, *Vars*, *Regs* and *Labels* must contain at least enough elements to model the data items and registers of a processor. The program counter *pc*, contained in the set of names, is required only for the semantics of the language $\mathcal{L}$. Although the name *pc* can be used to model the program counter of a processor, this is not a formal requirement. The name *pc* is needed only to support the selection of commands in programs $\mathcal{L}$. In practice, it is useful to chose the sets of values, names and labels to be larger than needed for a processor language's semantics since restrictions can be imposed on the functions which model the processor operations. This approach provides a greater degree of flexibility when modelling processor operations and when reasoning about the expressions of $\mathcal{L}$.

Machine State

All data operations are evaluated in a machine state, which is a record of the values stored in the program variables at point in a program execution. Since each program variable is identified by a name, in *Names*, a *state* is modelled as a function from names to values.

Definition 3.2 *States*

A state is a total function from the names to the values.

$$\mathit{State} \stackrel{\text{def}}{=} (\mathit{Names} \to \mathit{Values})$$

$\square$

The value of a name x in a state s is obtained as the result of $s(x)$. Because the program counter *pc* is a name, each state s also identifies a command by the value $s(\mathit{pc})$ assigned to the program counter; this value is assumed to be a label in *Labels*. The changes made to a machine state s by a command can be modelled by updating the state s with the values assigned to the names.

Undefined Values

An operation may be undefined for some values, for example when a division by zero is attempted. The result of such an operation is unknown and if a subsequent operation depends on this result then the result of that operation must also be unknown. Undefined data can be modelled as the result of applying the Hilbert epsilon operator to the empty set. The result of such an application is always unknown.

Definition 3.3 *Undefined values*

Given a set S, the result of function $\mathit{undef}(S)$ is an element of S which makes $\mathit{false} = \mathit{true}$.

$$\begin{aligned} &\mathit{undef} : \mathit{Set}(T) \to T \\ &\mathit{undef}(S) \stackrel{\text{def}}{=} \epsilon(\{x : S \mid \mathit{false}\}) \end{aligned}$$

$\square$

The function *undef* can be applied to any set including the values, the names and the registers. The result of all such applications is undefined: there is no element of a set S which satisfies *false*. This approach does not allow reasoning about the undefined value (as the approach of Barringer et al., 1984 and Grundy, 1993 does) but is enough to allow expressions of $\mathcal{L}$ to model processor data operations.

Basic Functions

The expressions of $\mathcal{L}$ are built up from the values, names and labels and from the application of functions. The functions of $\mathcal{L}$ are defined by identifiers and by an interpretation of these identifiers. The function identifiers are used in the syntax of an expression while the interpretation of the identifiers is used in the semantics of the expressions. All arguments to a function are interpreted as values, the domain of a function of arity n is the nth product of the set *Values*. The result of a function is a value, a name or a label and each function identifier is associated with one of the sets of values, names and labels. The result of a *value function* is a value, the result of a *name function* is a name and the result of a *label function* is a label.

Definition 3.4 *Function names and interpretation*

There is a set, $\mathcal{F}$, of function identifiers and a total function, *arity*, giving the arity of each function name, $\textit{arity} : \mathcal{F} \to \mathbb{N}$. There is also an interpretation function, $\mathcal{I}_f$, on the function identifiers, with type:

$$\mathcal{I}_f : \mathcal{F} \to (\textit{Values} \times \cdots \times \textit{Values}) \to (\textit{Values} \cup \textit{Names})$$

For each of the sets *Values*, *Labels* and *Names*, there is an associated set of function identifiers called the value functions, $\mathcal{F}_V$, the label functions, $\mathcal{F}_l$ and the name functions, $\mathcal{F}_n$ respectively. Each of these sets is a subset of $\mathcal{F}$: $(\mathcal{F}_V \cup \mathcal{F}_l \cup \mathcal{F}_n) \subseteq \mathcal{F}$. The sets are defined:

$$\begin{aligned}
\mathcal{F}_V &\stackrel{\text{def}}{=} \{f \in \mathcal{F} \mid \forall (v_1, \ldots, v_m) : \mathcal{I}_f(f)(v_1, \ldots, v_m) \in \textit{Values}\} \\
\mathcal{F}_n &\stackrel{\text{def}}{=} \{f \in \mathcal{F} \mid \forall (v_1, \ldots, v_m) : \mathcal{I}_f(f)(v_1, \ldots, v_m) \in \textit{Names}\} \\
\mathcal{F}_l &\stackrel{\text{def}}{=} \{f \in \mathcal{F} \mid \forall (v_1, \ldots, v_m) : \mathcal{I}_f(f)(v_1, \ldots, v_m) \in \textit{Labels}\}
\end{aligned}$$

where m is the arity of the function identifier.

An identifier, **equal**, with arity 2 and in the set of value functions, is interpreted as the equality between values.

$$\begin{aligned}
&\textbf{equal} \in \mathcal{F}_V \\
&\mathcal{I}_f(\textbf{equal})(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } v_1 = v_2 \\ \textbf{false} & \text{otherwise} \end{cases}
\end{aligned}$$

For any v_1, v_2, $\textbf{equal}(v_1, v_2)$ will usually be written $v_1 =_a v_2$. □

The functions of $\mathcal{L}$, identified by names in the set $\mathcal{F}$, are the basis for the expressions of $\mathcal{L}$. The value functions will normally include the arithmetic operations (addition, subtraction, etc.) and comparison functions (in addition to the equality **equal**). The name and label functions construct names and labels from values and are used to model the processor operations which identify variables or the labels of commands. Because the labels are a subset of the values, the set of label functions is also a subset of the value functions, $\mathcal{F}_l \subseteq \mathcal{F}_v$. A typical definition of the sets $\mathcal{F}_n$ and $\mathcal{F}_l$ contains a single identifier, of arity 1, whose interpretation results in the name or label, identified by the value argument. The result of applying a name or label function to an argument which does not identify a valid name or location will be undefined.

Example 3.1 *Basic name and label functions*

The values can index the variables in memory by the use of a name function. Let **ref** be a name function, **ref** $\in \mathcal{F}_n$, with arity 1 and definition:

$$\mathcal{I}_f(\mathbf{ref})(v) \stackrel{\text{def}}{=} \begin{cases} \mathit{name}(v) & \text{if } v \in \mathit{Vars} \\ \mathit{undef}(\mathit{Names}) & \text{otherwise} \end{cases}$$

If the argument v (a value) identifies a location in memory in which the program can store data then **ref**(v) constructs the name identifying that location. If v is not a variable, the result of **ref**(v) is the undefined name.

The values can also index the program instructions. A label function **loc** $\in \mathcal{F}_l$ analogous to the name function **ref** can be defined:

$$\mathcal{I}_f(\mathbf{loc})(v) \stackrel{\text{def}}{=} \begin{cases} v & \text{if } v \in \mathit{Labels} \\ \mathit{undef}(\mathit{Labels}) & \text{otherwise} \end{cases}$$

□

The functions **ref** and **loc** will be used in the examples as the basic name and label functions from which other functions accessing the names and labels are derived.

The basic model of $\mathcal{L}$ defines the data items and operations which form the basis for all expressions of the language $\mathcal{L}$. To model the data operations of a processor language, the basic model of $\mathcal{L}$ would include the natural numbers, as the values and variables, the arithmetic operations, as value functions, and simple name and label functions, to model memory access. A processor language imposes limits on the data items which are permitted in an instruction, e.g. limiting values to a fixed set of numbers. Such restrictions are better imposed by suitable models of the operations as expressions of $\mathcal{L}$ rather than restricting the constants and functions on which these expressions are based.

Example 3.2 *Processor data operations: basic model*

A basic model in which the operations of a processor can be defined is given in Figure (3.1). The set of values, *Values*, on which an object code program operates is modelled by the set of natural numbers. The names of program variables are the memory locations, the set *Vars*, together

$$Values \stackrel{\text{def}}{=} \mathbb{N} \quad Labels \stackrel{\text{def}}{=} \mathbb{N} \quad Vars \stackrel{\text{def}}{=} \mathbb{N} \quad Regs \stackrel{\text{def}}{=} \{pc, \mathbf{r0}, \mathbf{r1}, \dots\}$$

$$\{\mathbf{lt}, \mathbf{plus}, \mathbf{minus}, \mathbf{mult}, \mathbf{mod}, \mathbf{exp}\} \subseteq \mathcal{F}_\mathcal{V}$$

$$\mathcal{I}_f(\mathbf{lt})(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} \mathbf{true} & \text{if } v_1 < v_2 \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$\begin{array}{ll} \mathcal{I}_f(\mathbf{plus})(v_1, v_2) \stackrel{\text{def}}{=} v_1 + v_2 & \mathcal{I}_f(\mathbf{mod})(v_1, v_2) \stackrel{\text{def}}{=} v_1 \bmod v_2 \\ \mathcal{I}_f(\mathbf{minus})(v_1, v_2) \stackrel{\text{def}}{=} v_1 - v_2 & \mathcal{I}_f(\mathbf{exp})(v_1, v_2) \stackrel{\text{def}}{=} v_1^{v_2} \\ \mathcal{I}_f(\mathbf{mult})(v_1, v_2) \stackrel{\text{def}}{=} v_1 \times v_2 & \end{array}$$

Notation: For any v_1, v_2, $\mathbf{lt}(v_1, v_2)$ is written $v_1 <_a v_2$, $\mathbf{plus}(v_1, v_2)$ is written $v_1 +_a v_2$, $\mathbf{mult}(v_1, v_2)$ is written $v_1 \times_a v_2$, $\mathbf{minus}(v_1, v_2)$ is written $v_1 -_a v_2$, $\mathbf{mod}(v_1, v_2)$ is written $v_1 \bmod_a v_2$ and $\mathbf{exp}(v_1, v_2)$ is written ${v_1}^{v_2}$

Figure 3.1: Basic Model for Data Operations

with the processor registers, the set *Regs*. Both the variables, in *Vars*, and labels, in *Labels*, are memory addresses and memory access is modelled in terms of the function identifiers **ref**, for variables, and **loc**, for labels. Registers will be referred to by their symbolic name, e.g. **r0** is the name (in *Names*) constructed from register **r0**.

The data operations of a processor are defined on *bit-vectors* of a fixed size (Hayes, 1988). These operations can be defined in terms of the value functions of Figure (3.1), with a bit-vector represented as a natural number. The value functions are the basic arithmetic operations on natural numbers. Functions **lt** is the less-than comparison between values in *Values*, functions **plus**, **minus** and **mult**, are the addition, subtraction and multiplication of values. Functions **mod** and **exp** are the modulus and exponentiation operators, used to define the operations on fixed sizes of bit-vectors (see e.g. Yuan Yu, 1992). A value function **div** $\in \mathcal{F}_\mathcal{V}$ for the division of values can also be defined in terms of the Hilbert epsilon operator:

$$\mathcal{I}_f(\mathbf{div})(v_1, v_2) \stackrel{\text{def}}{=} \epsilon\{v : Values \mid \exists m \in Values : m < v \wedge v_1 = ((v_2 \times v) + m)\}$$

Since $Values = \mathbb{N}$, for any $v_1, v_2 \in Values$, the result of $\mathcal{I}_f(\mathbf{div})(v_1, v_2)$ is the integer division of v_1 by v_2. Note that if $v_2 = 0$ then $\mathcal{I}_f(\mathbf{div})(v_1, v_2) = undef(Values)$. □

Unless otherwise stated, the examples in this thesis will use the name function **ref**, the label function **loc** and the data model of Figure (3.1), including the value functions. The data model will be extended as needed for the examples. In particular, identifiers and definitions will be added to the set of registers *Regs* and the value functions $\mathcal{F}_\mathcal{V}$.

3.1.2 Syntax of the Expressions

The expressions of $\mathcal{L}$ are built up from the constants and from the application of functions to arguments. As with the constants and functions, the expressions are partitioned into *value expressions*, which are interpreted as values, and *name expressions* and *label expressions*, interpreted as names and labels respectively. This allow restrictions to be imposed on the occurrence of an expression; for example, assignments can only be made to expressions which result in a name. The name and label expressions are subsets of the value expressions. A name expression is interpreted as identifying a register or memory variable in which a value is stored. The labels are a subset of the values, *Labels* $\subseteq$ *Values*, and any label expression is also a value expression.

Substitution will be defined in terms of *substitution expressions*. This allows the syntax of substitution, describing the changes to be made to an expression, to be separated from its semantics, which carries out the substitution. The definition of substitution is not straightforward since its semantics requires a semantics for the expressions, leading to mutual recursion over the expressions. To simplify the development, substitution expressions will be considered separately. The syntax of the expressions defined here will include a syntactic construct, *subst*, for use when the substitution expressions are described.

The expressions of $\mathcal{L}$ are defined as a set $\mathcal{E}$ containing the value expressions. The sets of name and label expressions are constructed as subsets of the set $\mathcal{E}$. The name expressions are those expressions in $\mathcal{E}$ which are either names or the application of a name function to arguments. The label expressions are either labels or the application of a label function. The value expressions also provide the tests used in the language $\mathcal{L}$. These are described by the Boolean expressions of $\mathcal{L}$, defined as a synonym of the value expressions.

Definition 3.5 *Syntax of the expressions*

The set of value expressions $\mathcal{E}$ is inductively defined:

$$\frac{e \in \mathit{Names} \cup \mathit{Values}}{e \in \mathcal{E}} \qquad \frac{f \in \mathcal{F} \quad e_1, \ldots, e_m \in \mathcal{E}}{f(e_1, \ldots, e_m) \in \mathcal{E}} \; (m = \mathit{arity}(f))$$

A substitution expression is defined in terms of a function of type *State* $\to$ *Values*:

$$\forall (f : \mathit{State} \to \mathit{Values}) : \mathit{subst}(f) \in \mathcal{E}$$

The set of name expressions, $\mathcal{E}_n$, is defined:

$$\frac{e \in \mathit{Names}}{e \in \mathcal{E}_n} \qquad \frac{f \in \mathcal{F}_n \quad e_1, \ldots, e_m \in \mathcal{E}}{f(e_1, \ldots, e_m) \in \mathcal{E}_n} \; (m = \mathit{arity}(f))$$

The set of label expressions, $\mathcal{E}_l$, is defined:

$$\frac{e \in \mathit{Labels}}{e \in \mathcal{E}_l} \qquad \frac{f \in \mathcal{F}_l \quad e_1, \ldots, e_m \in \mathcal{E}}{f(e_1, \ldots, e_m) \in \mathcal{E}_l} \; (m = \mathit{arity}(f))$$

The set of boolean expression $\mathcal{E}_b$ is the set of value expressions: $\mathcal{E}_b \stackrel{\text{def}}{=} \mathcal{E}$. □

Data operations which calculate a value from arguments are modelled by value expressions. The name and label expressions model memory operations, the name expressions identify the variables, either registers or in memory, available to a program. The label expressions are used, mainly in computed jumps, to identify the label of a command. Both name and label expressions allow restrictions to be imposed on the expressions which occur in a command of $\mathcal{L}$. Name expressions calculate the name of a variable from one or more value expressions, restricting the result to valid program variables. Label expressions occur in computed jumps and provide a means for ensuring that the target of the jump is the label of a program command.

Example 3.3 Typical value expressions include the arithmetic expressions with the function identifiers of Figure (3.1). Assume that $x, y, z \in$ *Names* and $l_1, l_2 \in$ *Labels*. The value expressions, using infix notation, include:

$$\begin{array}{l} 0, 1, 2 +_a 3, (2 +_a 3) \times_a 4 \\ x, y, z, x +_a 1, 3 +_a x, x +_a y, (x +_a y) \times_a (z -_a 2) \\ l_1, l_2, l_1 +_a 1, l_1 +_a l_2 l_1 +_a x, (y \times_a l_1) -_a x \end{array}$$

The value expressions also include the name expressions:

$$\mathbf{ref}(0), \mathbf{ref}(10), \mathbf{ref}(x), \mathbf{ref}(x +_a 1), \mathbf{ref}(x +_a y), \mathbf{ref}(l_1), \mathbf{ref}(l_1 \times_a x)$$

The name expressions include those constructed from the names and the name function **ref**: expressions $x, y, \mathbf{ref}(0), \mathbf{ref}(x +_a y), \mathbf{ref}(l_1)$ are names. The result of an expressions formed with **ref** may be undefined but since the result of *undef*(*Names*) is a name, such expressions are elements of $\mathcal{E}_n$. □

3.1.3 Semantics of the Expressions

The semantics of expressions are defined by interpretation functions which range over the sets of values, labels and names. The value and label expressions have the same interpretation, a label expression is constrained by its syntax to result in a label (in *Labels*). The interpretation of a value expression e in a state s results in a value. If e is a variable name then it is the value of the variable in state s. The interpretation of a name expression n as a name is either n, if n is a constant (in *Names*), or the result of applying a name function to arguments interpreted as values. The difference in the interpretation of value expressions and name expressions corresponds to the difference between *r-expressions* and *l-expressions* in program analysis (Aho et al., 1986).

Definition 3.6 *Interpretation of expressions*

The interpretation functions $\mathcal{I}_e$, $\mathcal{I}_n$ and $\mathcal{I}_l$, on expressions, name expressions and label expressions respectively, have types:

$$\begin{array}{ll} \mathcal{I}_e & : \mathcal{E} \to \mathit{State} \to \mathit{Values} \\ \mathcal{I}_n & : \mathcal{E}_n \to \mathit{State} \to \mathit{Names} \\ \mathcal{I}_l & : \mathcal{E}_l \to \mathit{State} \to \mathit{Labels} \end{array}$$

The interpretation as a value of expression $e \in \mathcal{E}$ in state s is defined by function $\mathcal{I}_e$.

$$\mathcal{I}_e(e)(s) \stackrel{\text{def}}{=} \begin{cases} e & \text{if } e \in \mathit{Values} \\ f(s) & \text{if } e = \mathit{subst}(f) \\ s(\mathcal{I}_n(e)(s)) & \text{if } e \in \mathcal{E}_n \\ \mathcal{I}_f(f)(\mathcal{I}_e(a_1)(s), \dots, \mathcal{I}_e(a_m)(s)) & \text{if } e = f(a_1, \dots, a_m) \end{cases}$$
$$\text{where } m = \mathit{arity}(f)$$

The interpretation as a name of the name expression $e \in \mathcal{E}_n$ in state s is defined by function $\mathcal{I}_n$.

$$\mathcal{I}_n(e)(s) \stackrel{\text{def}}{=} \begin{cases} e & \text{if } e \in \mathit{Names} \\ \mathcal{I}_f(f)(\mathcal{I}_e(a_1)(s), \dots, \mathcal{I}_e(a_m)(s)) & \text{if } e = f(a_1, \dots, a_m) \end{cases}$$
$$\text{where } m = \mathit{arity}(f)$$

The interpretation of the label expressions, $\mathcal{I}_l$, is the interpretation of the expressions: $\mathcal{I}_l \stackrel{\text{def}}{=} \mathcal{I}_e$.

The Boolean interpretation of an expression is defined by the function $\mathcal{I}_b$.

$$\begin{aligned} &\mathcal{I}_b : \mathcal{E}_b \to \mathit{State} \to \mathit{boolean} \\ &\mathcal{I}_b(e)(s) \stackrel{\text{def}}{=} \mathcal{B}(\mathcal{I}_e(e)(s)) \end{aligned}$$

□

When the interpretation function $\mathcal{I}_e$ is applied to an expression e, in a state s, any name expression x occurring in e is interpreted as a name x' and replaced with the value $s(x')$. When the interpretation of name expressions, $\mathcal{I}_n$, is applied to the name expression $f(a_1, \dots, a_n)$ in state s, the arguments $a_1, \dots, a_n$ are interpreted as values and evaluated in state s. The interpretation of the name function $\mathcal{I}_f(f)$ is applied to the resulting values to obtain the result of the expression. When either interpretation function is applied to a constant e, the result is e. However, the notion of a constant differs, a name $x \in \mathit{Names}$ is a constant only under the interpretation as a name, $\mathcal{I}_n$. The value of x under interpretation $\mathcal{I}_e$ depends on the state in which it is interpreted.

Example 3.4 Using the expressions of Example (3.3), the interpretations in a state s of the value expressions are:

$$\begin{array}{ll} \mathcal{I}_e(1)(s) = 1 & \mathcal{I}_e(2 +_a 3)(s) = \mathcal{I}_f(\textbf{plus})(2, 3) \\ \mathcal{I}_e(x)(s) = s(\mathcal{I}_n(x)) = s(x) & \mathcal{I}_e(x +_a 1)(s) = \mathcal{I}_f(\textbf{plus})(s(x), 1) \\ \mathcal{I}_e(\textbf{ref}(x)) = s(\mathcal{I}_n(\textbf{ref}(x))(s)) & \end{array}$$

The interpretations as names (using function $\mathcal{I}_n$) of the name expressions are:

$$\begin{aligned} \mathcal{I}_n(x)(s) &= x \\ \mathcal{I}_n(\textbf{ref}(x))(s) &= \mathcal{I}_f(\textbf{ref})(\mathcal{I}_e(x)(s)) = \mathcal{I}_f(\textbf{ref})(s(x)) \\ \mathcal{I}_n(\textbf{ref}(x +_a 1))(s) &= \mathcal{I}_f(\textbf{ref})(\mathcal{I}_e(x +_a 1)(s)) \\ &= \mathcal{I}_f(\textbf{ref})(\mathcal{I}_f(\textbf{plus})(s(x), 1)) \end{aligned}$$

□

The expressions of $\mathcal{L}$ are based on standard definitions for programming languages (see Loeckx & Sieber, 1987) extended with the name expressions, which provide both pointers and arrays. In Example (3.4), the expression **ref**(x) is a pointer: x is a name which is interpreted as a value e. This is the argument to the name function **ref** and if e identifies a memory variable, $e \in$ *Vars*, then the interpretation of **ref**(e) is also a name. Arrays can be modelled in a similar way. An array a is a name function in $\mathcal{F}n$. If e is an expression, then the expression $a(e)$ will be a name expression (identifying variables).

Boolean Expressions

The value expressions model the tests on program data performed by a processor instruction. This uses the Boolean interpretation $\mathcal{I}_b$ of the value expressions. For example, the equality of values $v_1, v_2 \in \mathcal{E}$ in a state s is tested by the expression, $\mathcal{I}_b(v_1 =_a v_2)(s)$. Since the Boolean constants are represented by values **true** and **false**, the Boolean negation and conjunction operators can be defined as value functions. This allows Boolean expressions of $\mathcal{L}$ to include formulas of a propositional (quantifier-free) logic.

Definition 3.7 *Boolean operators*

The negation and conjunction operators of $\mathcal{E}_b$ are defined as the value functions **not** and **and**, with arities 1 and 2 respectively.

$$\mathbf{not} \in \mathcal{F}_V \qquad\qquad \mathbf{and} \in \mathcal{F}_V$$

$$\mathcal{I}_f(\mathbf{not})(v) \stackrel{\text{def}}{=} \begin{cases} \mathbf{false} & \text{if } \mathcal{B}(e) \\ \mathbf{true} & \text{otherwise} \end{cases} \qquad \mathcal{I}_f(\mathbf{and})(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} \mathbf{true} & \text{if } \mathcal{B}(v_1) \wedge \mathcal{B}(v_2) \\ \mathbf{false} & \text{otherwise} \end{cases}$$

For any x, y, **and**(x, y) will be written x **and** y. □

Since both the negation and conjunction operators are value functions, any expression formed by their application to arguments will be a value expression (in $\mathcal{E}$) and therefore in $\mathcal{E}_b$. Other Boolean operators can be defined in terms of negation and conjunction. For example, the disjunction of Boolean expressions **or** is also a Boolean expression of $\mathcal{L}$.

$$\_\ \mathbf{or}\ \_ : (\mathcal{E}_b \times \mathcal{E}_b) \to \mathcal{E}_b$$
$$x\ \mathbf{or}\ y \stackrel{\text{def}}{=} \mathbf{not}\ (\mathbf{not}\ x\ \mathbf{and}\ \mathbf{not}\ y)$$

The Boolean operators allow comparison functions to be derived from a small number of basic operations. For example, the greater than or equal, written $e_1 \geq_a e_2$, is defined $e_1 \geq_a e_1 \stackrel{\text{def}}{=} \mathbf{not}\ (e_2 <_a e_1)$. Operator $\geq_a$ will have type $\mathcal{E} \times \mathcal{E} \to \mathcal{E}$, constructing an expression of $\mathcal{E}$.

Example 3.5 *Data and memory operations*

For an example of the use of expressions of $\mathcal{L}$ to model data operations, consider the arithmetic operations of a processor which manipulates data items as bit-vectors (called *quad-words*), representing numbers in the range $0, \ldots, 2^{64} - 1$. All arithmetic operations of the processor must

$$
\begin{array}{ll}
\textit{Arithmetic:} & e_1 =_{64} e_2 \stackrel{\text{def}}{=} (e_1 \bmod_a 2^{64}) =_a (e_2 \bmod_a 2^{64}) \\
& e_1 <_{64} e_2 \stackrel{\text{def}}{=} (e_1 \bmod_a 2^{64}) <_a (e_2 \bmod_a 2^{64}) \\
& e_1 +_{64} e_2 \stackrel{\text{def}}{=} (e_1 +_a e_2) \bmod_a 2^{64} \\
& e_1 -_{64} e_2 \stackrel{\text{def}}{=} (e_1 -_a e_2) \bmod_a 2^{64} \\
& e_1 \times_{64} e_2 \stackrel{\text{def}}{=} (e_1 \times_a e_2) \bmod_a 2^{64} \\
& \mathbf{mkQuad}(e) \stackrel{\text{def}}{=} e \bmod_a 2^{64} \\
& \\
\textit{Memory:} & \mathbf{Mem}(e) \stackrel{\text{def}}{=} \mathbf{ref}(\mathbf{mkQuad}(a -_a (a \bmod_a 4))) \\
& \mathbf{Inst}(e) \stackrel{\text{def}}{=} \mathbf{loc}(\mathbf{mkQuad}(a -_a (a \bmod_a 4))) \\
& \\
& \text{where } e, e_1, e_2 \in \mathcal{E}
\end{array}
$$

Figure 3.2: Example of Expressions Modelling Data Operations

be performed within this range. The model of the basic arithmetic operations, as expressions of $\mathcal{L}$, is given in Figure (3.2). The arithmetic operations are written in the infix notation and for bit-vectors of size 64. e.g. $e_1 =_{64} e_2$ is the equality between values less than 2^{64} and is defined in terms of the value function **equal** $\in \mathcal{F}_\mathcal{V}$ of $\mathcal{L}$. Since all arithmetic operators of Figure (3.2) are defined in terms of $\mathcal{E}$, the type of each operator is $(\mathcal{E} \times \mathcal{E}) \to \mathcal{E}$. The function **mkQuad** is the conversion of an arbitrary value to value in the range which can be represented by the processor.

A processor can restrict the memory locations which can be accessed by an instruction; a common restriction is to require the address of a location to be a multiple of a constant. This can be modelled by name and label expressions (of $\mathcal{E}_n$ and $\mathcal{E}_l$). Figure (3.2) defines name expression $\mathbf{Mem}(e) \in \mathcal{E}_n$ and label expression $\mathbf{Inst} \in \mathcal{E}_l$ which restrict memory access to the locations whose address is a multiple of 4, rounding down if necessary. e.g. $\mathbf{Mem}(0)$ and $\mathbf{Mem}(4)$ identify distinct names $\mathbf{ref}(0)$ and $\mathbf{ref}(4)$ respectively while $\mathbf{Mem}(0)$ and $\mathbf{Mem}(1)$ are both name $\mathbf{ref}(0)$.

The addressing modes of a processor can be described in terms of the name expressions **ref** or **Mem**. Consider the addressing modes described in Chapter (2) (with $\mathbf{r0}, \mathbf{r1} \in \mathit{Regs}$ and $v \in \mathit{Values}$): In immediate addressing, the value v is an expression $v \in \mathcal{E}$ (by definition). In direct addressing, the name identified by address v is $\mathbf{Mem}(v)$. For indirect addressing, the name identified by the value stored in **r0** is $\mathbf{Mem}(\mathbf{r0})$. Furthermore, the name identified by the value stored in this variable is $\mathbf{Mem}(\mathbf{Mem}(\mathbf{r0}))$. In indexed addressing, the variable identified by the sum of **r0** and v is $\mathbf{Mem}(\mathbf{r0} +_{64} v)$. Since v is a value expression, it can be replaced with any other value expressions. e.g. Indexed addressing can be defined with two registers, $\mathbf{Mem}(\mathbf{r0} +_{64} \mathbf{r1})$. *Relative addressing* is a form of indexed addressing which uses the program counter: $\mathbf{Mem}(\mathit{pc} +_{64} e)$ (for any $e \in \mathcal{E}$). Similar expressions can be built up for the label expressions. For example, the label identified by the value stored in register **r0** is $\mathbf{Inst}(\mathbf{r0})$. □

3.1.4 Equivalence Between Expressions

Expressions can be compared by syntactic equality or by semantic equivalence. Equality compares the textual form of two expressions while equivalence compares the interpretation of the expressions in a state. Syntactic equality is stronger than equivalence: the expression $1 + 1$ is not syntactically equal to 2, although it is equivalent in an interpretation which includes integer arithmetic. The difference between syntactic equality and semantic equivalence is important when comparing name expressions, as required for substitution. To avoid the aliasing problem, it is necessary to compare name expressions using semantic equivalence. This allows the names referred to by the name expressions to be determined before the comparison is carried out.

There are two forms of equivalence: the weaker asserts that two expressions have the same interpretation in a given state. Strong equivalence asserts that the expressions are equivalent in all states. Because the name expressions can be interpreted as names or values, the interpretation under which expressions are to be compared is an argument to the equivalence relations.

Definition 3.8 *Equivalence*

Expressions e_1 and e_2 are *equivalent* in a state s and interpretation $\mathcal{I}$, written $e_1 \equiv_{(\mathcal{I},s)} e_2$, if the interpretation of the expressions in s are equal.

$$e_1 \equiv_{(\mathcal{I},s)} e_2 \stackrel{\text{def}}{=} \mathcal{I}(e_1)(s) = \mathcal{I}(e_2)(s)$$

Expressions e_1 and e_2 are *strongly equivalent* in $\mathcal{I}$, written $e_1 \equiv_{\mathcal{I}} e_2$, if they are equivalent in all states.

$$e_1 \equiv_{\mathcal{I}} e_2 \stackrel{\text{def}}{=} \forall (s : \mathit{State}) : e_1 \equiv_{(\mathcal{I},s)} e_2$$

$\Box$

Because the equivalence relations are based on the semantics of the expressions, syntactically equal expressions are also equivalent. Names which are equivalent under the name interpretation, $\mathcal{I}_n$, will also be equivalent under the value interpretation $\mathcal{I}_e$ and arguments to functions can always by replaced with an equivalent expression.

Lemma 3.1 *Properties of equivalence relations*[1]

Assume $e_1, e_2 \in \mathcal{E}$, $n_1, n_2 \in \mathcal{E}_n$, $\mathcal{I}, \mathcal{I}_1, \mathcal{I}_2 \in \{\mathcal{I}_e, \mathcal{I}_n\}$, $f \in \mathcal{F}$ *and* $s \in \mathit{State}$.

1. *Syntactic equality establishes strong equivalence:*

$$\frac{e_1 = e_2}{e_1 \equiv_{\mathcal{I}} e_2}$$

[1] See the appendix for the proof of this and subsequent lemmas and theorems.

2. *Equivalent names are equivalent as values:*

$$\frac{n_1 \equiv_{(\mathcal{I}_n,s)} n_2}{n_1 \equiv_{(\mathcal{I}_e,s)} n_2}$$

3. *Arguments to functions can be replaced with equivalent expressions:*

$$\frac{e_1 \equiv_{(\mathcal{I}_1,s)} e_2}{f(e_1) \equiv_{(\mathcal{I}_2,s)} f(e_2)}$$

Because name expressions which are equivalent under $\mathcal{I}_n$ are also equivalent under $\mathcal{I}_e$, the interpretation function will generally not be given. For expressions e_1, e_2 and $s \in$ *State*, $e_1 \equiv_s e_2$ will be written under the assumption that if both e_1 and e_2 are name expressions then the equivalence is under $\mathcal{I}_n$. If either of e_1 or e_2 is not a name expression then the equivalence is under the value interpretation $\mathcal{I}_e$.

The ability to compare the interpretation of names is important for substitution in the presence of name expressions. The use of syntactic equality would result in textual substitution, which can replace constant names only (those in the set *Names*). The textual substitution of an expression for a name expression $f(e)$ would only replace name expressions which are syntactically equal to $f(e_1)$. Name expressions which are equivalent to $f(e_1)$ would not be replaced.

Example 3.6 Let $\mathbf{Id}$ be the identify function such that $\mathbf{Id}(x) \equiv x$ for any x. The expressions $\mathbf{Id}(x)$ and x are not syntactically equal but are strongly equivalent. The textual substitution of expression e for $\mathbf{Id}(x)$ in x is x, $x[e/\mathbf{Id}(x)] = x$. However, the textual substitution of e for x is e, $x[e/x] = e$. □

3.1.5 Substitution

The language $\mathcal{L}$ must provide a substitution operator, to allow the definition of proof rules and to abstract from programs. Substitution is used to describe the changes made to a machine state by the commands of a program. These changes are made as values are assigned to the program variables during the execution of a program and affect the expressions which depend on the values stored in the variables. Substitution allows these changes to be reflected in expressions occurring in a specification or in the commands formed by abstraction.

For example, assume an instruction begins in state s and ends in state t and that the instruction assigns the result of evaluating expression e in state s to name $x \in$ *Names*. Assume also that the instruction makes no other assignment. The value of name x in state t is the value of expression e in state s,

$$\mathcal{I}_e(x)(t) = \mathcal{I}_e(e)(s)$$

If the name x occurs in expression f then the value of expression f in state t will differ from its value in state s, the value of x having changed. The effect of the instruction on the value of expression f can be described by the textual substitution of e for x in f:

$$\mathcal{I}_e(f)(t) = \mathcal{I}_e(f[e/x])(s)$$

Textual substitution can be used when a language contains only constant names (in *Names*) but fails in the presence of name expressions. Let the instruction evaluate a name expression $g(e_1) \in \mathcal{E}_n$ in state s and assign e to the resulting name $x_1 \in$ *Names*. Also assume that the instruction makes no other assignment and let f depend on any name. Using textual substitution to describe the changes to the value of f made by the instruction will fail: the substitution $f[e/g(e_1)]$ is not the same as the substitution $f[e/x_1]$ (see Example 3.6).

Definition of Substitution

To overcome the problems caused by the presence of name expressions, the substitution operator of $\mathcal{L}$ must be based on the equivalence of name expressions rather than on syntactic equality. This is the approach taken when verifying programs with arrays (Dijkstra, 1976; de Bakker, 1980; Tennent, 1991; Francez, 1992). The definition which will be used here is essentially that described by Francez (1992) (which is similar to work by de Bakker, 1980). However, the substitution operator of $\mathcal{L}$ differs from that of Francez (1992) in allowing the simultaneous substitution of expressions for name expressions.

The substitution operator of $\mathcal{L}$ is defined as a function which is applied to an expression e and a list of assignments to form an expression of $\mathcal{L}$. The interpretation of this expression evaluates e in a state updated with the assignments in the list. Assume that *Alist* is the type of *assignment lists*, where an assignment is a pair of a name expression and a value expression. The substitution operator will be written $\triangleleft$ (in infix notation) and has type $(\mathcal{E} \times \textit{Alist}) \to \mathcal{E}$. For expression e and assignment list al, the substitution expression $e \triangleleft al$ interpreted in a state s is the value of e in s after replacing every name in assignment list al with the value it is assigned.

Example 3.7 Assume, as before, that an instruction begins in state s, assigns the expression e to the result of evaluating name expression $g(e_1)$ in state s and ends in state t. The assignment list for the instruction contains only the pair $(g(e_1), e)$. Assume that the result of evaluating the name expression $g(e_1)$ in s is the name $x \in$ *Names* and that the expression f depends on the value of x. The changes made to the value of f by the instruction can be described as the expression $(f \triangleleft (g(e_1), e))$.

$$\mathcal{I}_e(f)(t) = \mathcal{I}_e(f \triangleleft (g(e_1), e)))(s)$$

Any name expression occurring in f which is equivalent in state s to $g(e_1)$ is replaced with e. $\square$

Substitution and Abstraction

The use of textual substitution to describe the changes made to the value of an expression is the basis for abstracting commands in which only basic names occur. The rules of Hoare et al. (1987) describe how an abstraction c of two assignment commands c_1 and c_2, executed in sequence, can be constructed from the syntax of c_1 and c_2. Command c is formed by substituting the assignments of c_1 into the expression of c_2. The assignment lists of c_1 and c_2 are then merged so that each variable is assigned to at most once. If a variable is assigned to be both commands c_1 and c_2, the later assignment, of c_2, is used.

Example 3.8 Let the assignment list of c_1 contain only (x_1, e_1) and (x_2, e_2) and let the assignment list of c_2 contain only (x_2, e_2) and (x_3, e_3), where $x_1, x_2, x_3 \in$ *Names* and $e_1, e_2, e_3 \in \mathcal{E}$.

The abstraction c is constructed by substituting the assignments made by c_1 in the expressions of c_2: the updated assignments of c_2 are $(x_2, e_2[e_1, e_2/x_1, x_2])$ and $(x_3, e_3[e_1, e_2/x_1, x_2])$. The two assignment lists are combined, x_2 is assigned to by both c_1 and c_2 and the assignment of c_2 is used. The command c therefore makes three assignments: (x_1, e_1), $(x_2, e_2[e_1, e_2/x_1, x_2])$ and $(x_3, e_3[e_1, e_2/x_1, x_2])$. □

The programming language considered by Hoare et al. (1987) allowed only constant names, which could be compared by syntactic equality. This is used when merging the two assignment lists to determine whether a variable is assigned to by both c_1 and c_2. Since the language $\mathcal{L}$ includes name expression, it is not possible to use this method of merging assignment lists (because of aliasing). An alternative can be based on the fact that the assignment list of the abstraction c is the result of combining the assignments of c_1 and c_2, with a higher priority given to the assignments of c_2. The merger of the assignment lists is needed only to remove duplicate assignments to a single variable (which would result in an unexecutable command).

Both the substitution operator and the assignment commands, which must be simultaneous, require that each variable is assigned a single value. However, both can be defined using an ordering on the assignments to search for the value assigned to a given name (Paulson, 1985, defined substitution using this approach for efficiency). Ordering the assignments in a list allows the assignments of command c_1 to be combined with the assignments of command c_2, without the need to find and remove the variables common to both. The order is defined such that if a variable is assigned to by c_2 and by c_1, then it is the assignment of c_2 which is used. The data structure used to represent assignment lists can be used to impose the order on assignments. For example, Paulson (1985) used association lists and the assignment used was the first to be found by traversing the list.

Example 3.9 Using lists, as in Paulson (1985), the assignment list for c is constructed by updating the expressions of c_2 with the assignments of c_1 and then constructing the list which contains the assignments of c_2 followed by those of c_1. The assignment list for c will be (in order) $(x_2, e_2[e_1, e_2/x_1, x_2])$, $(x_3, e_3[e_1, e_2/x_1, x_2])$, (x_1, e_2), (x_2, e_2). □

The advantage of this method is that the assignment list for the abstraction c can be efficiently constructed from the syntax of c_1 and c_2. It simply requires the two assignment lists to be appended together. However, using a list to represent the assignments made by a command means that there can be no distinction between a correct assignment, which assigns only one value to each variable, and an incorrect assignment, assigning two values to a single variable. For example, the assignment $x, x := 1, 0$ would be a valid command: only the first assignment to x would be used, making the command equivalent to $x := 1$. To disallow incorrect assignments, the combination of assignment lists must allow individual assignment lists to be extracted. Each list would then be examined to ensure that none assigns more than one value to any variable.

The approach which will be used here is based on representing the combination of assignment lists syntactically, in terms of an operator $\oplus$ with type $(\mathit{Alist} \times \mathit{Alist}) \to \mathit{Alist}$. Assume al is the assignment list of command c_1 and bl the assignment list of command c_2. The command c abstracting c_1 and c_2 will have the assignment list $al \oplus bl$. This allows the individual lists al and bl to be extracted from the assignment list of c. Command c is a correct assignment iff neither al nor bl assigns two values to the same variable. The assignment list $al \oplus bl$ includes two assignments to x_2 but since the first occurs in al and the second occurs in bl, this is a correct assignment. To find the assignment made to a name in the combined list $al \oplus bl$, the list bl is first searched for an assignment to x then the list al. The value assigned to x_2 will be the value assigned in bl by c_2. Furthermore, the command $x, x := 1, 0$ will be incorrect since the name x is assigned two values by the single assignment list.

The substitution operator of $\mathcal{L}$ will be developed as follows: first the data structure used to represent the lists of assignments will be described. This will be followed by the definition of functions on assignments lists which are then used to define a state transformer, to update a state with a list of assignments. This is followed by the syntax and semantics of the basic substitution operator of $\mathcal{L}$. This operator can be applied to any value expression and results in a value expression. Substitution operators for name expressions and for lists of assignments are then derived from the basic substitution operator. Together, these provide the constructs needed to manipulate the expressions of $\mathcal{L}$ when verifying and abstracting commands.

3.1.6 Assignment Lists

The assignments made by a command are a list of name expressions and value expressions, where each name expression is paired with the value expression it is assigned. An assignment list can be empty, the result of adding a name-value expression pair to an assignment list or the combination of two assignment lists.

Definition 3.9 *Assignment lists*

There is a set *Alist* and constructor functions *nil*, *cons* and *combine* satisfying:

$$\mathit{nil} \in \mathit{Alist} \qquad \frac{x \in \mathcal{E}_n \quad e \in \mathcal{E} \quad al \in \mathit{Alist}}{\mathit{cons}((x, e), al) \in \mathit{Alist}} \qquad \frac{al \in \mathit{Alist} \quad bl \in \mathit{Alist}}{\mathit{combine}(al, bl) \in \mathit{Alist}}$$

$(x, e) \cdot al$ will be written for $cons((x, e), al)$ and $(x, e) \cdot nil$ will be abbreviated (x, e). $al \oplus bl$ will be written for $combine(al, bl)$.

Function *combine?* is a recogniser for assignment lists constructed by combination.

$$\begin{aligned} &combine? : Alist \to boolean \\ &combine?(al) \stackrel{\text{def}}{=} \exists(bl, cl : Alist) : al = bl \oplus cl \end{aligned}$$

□

The combination of assignments lists is by the use of the syntactic construct *combine*. This allows the order of assignments to be preserved. When a sequence of assignment lists is combined, the order in which assignments to a name are made can be determined from the syntax of the assignment lists.

The structure of an assignment list, in the set *Alist*, is strictly a tree. The assignment lists occurring in processor commands would not be formed using the *combine* constructor and can be described by the subset of the assignment lists which excludes this constructor. This subset contains the *simple* assignment lists, corresponding to the usual list structures. The prefix of an assignment list, up to the first combination of lists, is a simple list and an assignment list is made up of the combination of a finite set of simple lists.

Definition 3.10 *Simple lists*

An assignment list al is *simple*, $simple?(al)$, if no sub-list of al is constructed by *combine*.

$$\begin{aligned} &simple? : Alist \to boolean \\ &simple?(al) \stackrel{\text{def}}{=} \forall(bl : Alist) : bl \ll al \Rightarrow \neg combine?(bl) \end{aligned}$$

Slist is the set of all simple lists, $Slist \stackrel{\text{def}}{=} \{al : Alist \mid simple?(al)\}$.

Function *initial* constructs a simple list from the prefix of an assignment list.

$$\begin{aligned} initial &: Alist \to Slist \\ initial(nil) &\stackrel{\text{def}}{=} nil \\ initial((v, x) \cdot al) &\stackrel{\text{def}}{=} (v, x) \cdot initial(al) \\ initial(al \oplus bl) &\stackrel{\text{def}}{=} nil \end{aligned}$$

□

Function *initial* allows an arbitrary assignment list to be considered as a set of simple lists. This allows the assignments made by an individual list to be examined separately from the remainder of an assignment list.

Example 3.10 With assignment lists $al, bl \in Alist$ and name-value pairs, $a, b \in (\mathcal{E}_n \times \mathcal{E})$, the set of assignment lists, *Alist*, includes $a \cdot (al \oplus bl)$ and $a \cdot b \cdot ((b \cdot al) \oplus bl)$. Neither of these would occur in an command nor would they be formed to describe a sequence of instructions. The result of applying the function *initial* to the first list is the simple list $a \cdot nil$ and to the second is the simple list $a \cdot b$. □

Finding Assignments in Lists

An assignment list associates name expressions with the value expressions which they are assigned. Name expression x is associated in state s with a value v by an assignment list al if there is a pair $(x', e') \in (\mathcal{E}_n \times \mathcal{E})$ in al such that $x \equiv_s x'$ and $v \equiv_s e'$. The value associated with a name x in a state s by al is found by traversing the assignment list. If the assignment list is constructed from the addition of a pair (x_1, e_1) to an assignment list bl then x_1 is compared (in state s with x) before searching bl. If the assignment list is constructed from the combination of two assignment lists $(bl \oplus cl)$ then the list cl is searched before the list bl (the choice is arbitrary).

Definition 3.11 *Membership and find*

For an assignment list al and state s, the name $x \in_s \mathcal{E}_n$ is a member in s of al iff there is a name expression in al which is equivalent in s to x.

$$\begin{aligned}
_ \in_{_} _ &: (\mathcal{E}_n \times \mathit{State} \times \mathit{Alist}) \to \mathit{boolean} \\
x \in_s \mathit{nil} &\stackrel{\text{def}}{=} \mathit{false} \\
x \in_s (x_1, e_1) \cdot al &\stackrel{\text{def}}{=} (x \equiv_s x_1) \vee (x \in_s al) \\
x \in_s (al \oplus bl) &\stackrel{\text{def}}{=} x \in_s al \vee x \in_s bl
\end{aligned}$$

Function *find* searches an assignment list for the value expression assigned to a name expression.

$$\begin{aligned}
\mathit{find} &: (\mathcal{E}_n \times \mathit{Alist}) \to \mathit{State} \to \mathcal{E} \\
\mathit{find}(x, \mathit{nil})(s) &\stackrel{\text{def}}{=} x \\
\mathit{find}(x, (x_1, e_1) \cdot al)(s) &\stackrel{\text{def}}{=} \begin{cases} e_1 & \text{if } x \equiv_s x_1 \\ \mathit{find}(x, al) & \text{otherwise} \end{cases} \\
\mathit{find}(x, (al \oplus bl))(s) &\stackrel{\text{def}}{=} \begin{cases} \mathit{find}(x, bl)(s) & \text{if } x \in_s bl \\ \mathit{find}(x, al)(s) & \text{otherwise} \end{cases}
\end{aligned}$$

□

Given a name expression x, assignment list al and state s, the result of $\mathit{find}(x, al)(s)$ is the value expression associated with name x in al, if x is a member in s of al. If x is not a member in x of al, the result is the name expression x.

Example 3.11 Let x_1, x_2, x_3 be name expressions, e_1, e_2, e_3, e_4 be expressions, s a state, al the assignment list $(x_1, e_2) \cdot (x_2, e_2)$ and bl the assignment list $(x_2, e_3) \cdot (x_3, e_4)$. Assume that x_1, x_2 and x_3 are distinct in s: $x_1 \not\equiv_s x_2$, $x_2 \not\equiv_s x_3$ and $x_1 \not\equiv_s x_3$.

The expressions associated with x_1, x_2 and x_3 in al are

$$\mathit{find}(x_1, al)(s) = e_1, \quad \mathit{find}(x_2, al)(s) = e_2, \quad \mathit{find}(x_3, al)(s) = x_3$$

The values associated with x_1, x_2 and x_3 in $al \oplus bl$ are

$$\mathit{find}(x_1, al \oplus bl)(s) = e_1, \quad \mathit{find}(x_2, al \oplus bl)(s) = e_3, \quad \mathit{find}(x_3, al \oplus bl)(s) = e_4$$

the assignment list bl being searched before the assignment list al. □

3.1.7 State Update

A state s is a record of the values assigned to each name: when a command c assigns values to names, the state is updated with the new values of the names. The value assigned to a name may the result of evaluating an expression e and the name may be the result of evaluating an name expression x. Both expressions e and x are evaluated in the state s. The change made to the state is therefore the assignment of $\mathcal{I}_e(e)(s)$ to $\mathcal{I}_n(x)(s)$. Assume the assignments made by command c are given as an assignment list al. To find the value of a name x after the command c has been executed, the expression associated by al with x in state s must be found and evaluated in state s. This defines how a state is updated with the assignment list of a command.

Definition 3.12 *State update*

For assignment list al and state s, *update*(al, s) is a state which differs from s only in the assignments given in al.

$$\begin{aligned}&\mathit{update} : (\mathit{Alist} \times \mathit{State}) \to \mathit{State}\\&\mathit{update}(al, s) \stackrel{\text{def}}{=} (\lambda(x : \mathit{Names}) : \mathcal{I}_e(\mathit{find}(x, al)(s)))\end{aligned}$$

□

Any name x which is not assigned a value by the command c will not be a member of the assignment list al and the value of x in the updated state will be its value in s. If the name is assigned an expression e by command c then its value in the updated state is the value of e in state s. This definition of a state update is a generalisation of the definition of Francez (1992), which considers names-value pairs individually and interprets a multiple assignment as a sequence of single assignments.

3.1.8 Substitution Expressions

A substitution is made up of an expression e and an assignment list al and replaces in e the names occurring in al with their assigned value. This is equivalent to evaluating e in a state updated with the assignments of al. The substitution operator of $\mathcal{L}$ constructs a substitution expression using the construct *subst* (see Definition 3.5). This requires a function of type *State* $\to$ *Values*, which will define the semantics of substitution. For expression e and assignment list al, this can be formed, using the function *update*, as $\lambda(s : \mathit{State}) : \mathcal{I}_e(e)(\mathit{update}(al, s))$.

Definition 3.13 *Substitution*

For assignment list al and expression e, the substitution of al in e is a value expression in $\mathcal{E}$ written $e \triangleleft al$ and defined:

$$\begin{aligned}&\_ \triangleleft \_ : (\mathcal{E} \times \mathit{Alist}) \to \mathcal{E}\\&e \triangleleft al \stackrel{\text{def}}{=} \mathit{subst}(\lambda(s : State) : \mathcal{I}_e(e)(\mathit{update}(al, s)))\end{aligned}$$

□

The interpretation of a substitution expression is that of the constructor *subst* of the set $\mathcal{E}$ (see Definition 3.5 and Definition 3.6). For $e \in \mathcal{E}$, $al \in$ *Alist* and $s \in$ *State*, the result of $\mathcal{I}_e(e \triangleleft al)(s)$ is $\mathcal{I}_e(e)(\mathit{update}(al, s))$. Note that although substitution is a value expression (in $\mathcal{E}$), if it is applied to a label expression ($e \in \mathcal{E}_l$) then the result will also be a label expression. The set $\mathcal{E}_l$ contains only basic labels or expressions constructed from label functions. Any name which occurs in a label expression can do so only as an argument to a label function.

Example 3.12 *Let* s be a state and al the assignment list $(x_1, e_1) \cdot (x_2, e_2)$ where $x_1 \not\equiv_s x_2$. Assume $x \in$ *Names*, $e \in \mathcal{E}$, $l \in$ *Labels* and label function $f \in \mathcal{F}_l$. Assume that $x \equiv_s x_2$. The substitution of al in the expressions is:

$$(x \triangleleft al) \equiv_s e_2 \qquad (l \triangleleft al) \equiv_s l \qquad (f(e) \triangleleft al) \equiv_s f(e \triangleleft al)$$

□

Substitution in Name Expressions

Substitution must be applied to name expressions as well as to value expressions: the interpretation of a name expression also depends on the state in which it is evaluated. It is possible to extend the name expressions with a substitution expression which is interpreted as a name. This would restrict the substitution operator to assignment lists which replace name expressions with name expressions. The interpretation, as a name, of the resulting operator would be $\mathcal{I}_n(\mathit{subst}(e, al))(s) = \mathcal{I}_n(e)(\mathit{update}(al, s))$. However, this approach separates the assignment command from the substitution operator since an assignment is a replacement of name expressions with value expressions.

The approach used here is to define substitution as a function on the syntax of name expressions. This applies substitution to the value expressions which occur as arguments to name functions. It is not equivalent to substitution as a name expression, where names are replaced with names, but retains the association with the assignment command.

Definition 3.14 *Substitution in name expressions*

The substitution of assignment list al in the value expressions occurring in name expression x is written $x \triangleleft al$.

$$\_ \triangleleft \_ : (\mathcal{E}_n \times \mathit{Alist}) \to \mathcal{E}_n$$
$$x \triangleleft al \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \in \mathit{Names} \\ f(e_1 \triangleleft al, \ldots, e_m \triangleleft al) & \text{if } x = f(e_1, \ldots, e_m) \end{cases}$$

where $f \in \mathcal{F}_n$, $m = \mathit{arity}(f)$ and $e_1, \ldots, e_m \in \mathcal{E}$. □

The effect of performing the substitutions in assignment list al on a name expression is equivalent (using the interpretation $\mathcal{I}_n$) to updating a state with al.

Lemma 3.2 *For name expression $x \in \mathcal{E}_n$, assignment lists $al, bl \in$ Alist and state s,*

$$\mathcal{I}_n(x \lhd al)(s) = \mathcal{I}_n(x)(\mathit{update}(al, s))$$

The interpretation of substitution in a name expression is consistent with its interpretation as a value. If substitution is applied to a constant name then the name is unchanged. If substitution is applied to a function application then the arguments to the function are evaluated in the updated state.

Example 3.13 Let s be a state and al an assignment list. Let x be a name in *Names*, e an expression and f a name function. The substitution of al in the name x and name expression $f(e)$ (interpreted as names) is

$$x \lhd al \equiv_{(\mathcal{I}_n, s)} x \qquad f(e) \lhd al \equiv_{(\mathcal{I}_n, s)} f(e \lhd al)$$

□

Substitution in Assignment Lists

The assignment lists in *Alist* are used both for the substitution expression and to describe the lists of assignments made by a command. The abstraction of assignment commands requires the ability to apply substitution to the assignment lists of the commands (to model the changes made to a machine state). A particular application of substitution in assignment lists is to describe syntactically the changes made by executing two assignment commands in succession. The correctness of this application justifies the method of abstracting from assignment commands and is established from the semantic interpretation of substitutions.

Substitution is applied to the expressions in an assignment list by applying the substitution for name expressions and the substitution for value expressions to each name-value pair in the assignment list.

Definition 3.15 *Substitution in assignment lists*

The substitution of assignment list bl in assignment list al is written $al \lhd bl$ and defined:

$$\begin{aligned} \_ \lhd \_ &: (\mathit{Alist} \times \mathit{Alist}) \to \mathit{Alist} \\ \mathit{nil} \lhd bl &\stackrel{\text{def}}{=} \mathit{nil} \\ ((x, e) \cdot al) \lhd bl &\stackrel{\text{def}}{=} (x \lhd bl, e \lhd bl)) \cdot (al \lhd bl) \\ (cl \oplus dl) \lhd bl &\stackrel{\text{def}}{=} (cl \lhd bl) \oplus (dl \lhd bl) \end{aligned}$$

□

The effect of substituting assignment list bl in assignment list al and then evaluating an expression e of al in state s is equivalent to evaluating e in the state s updated with bl. The substitution of assignment lists together with the combination operator allows the effect on a state of two assignment commands executed in sequence to be described as a single assignment list.

Theorem 3.1 *Basis for abstraction*

For assignment lists al, bl and state s,

$$\mathit{update}(al, \mathit{update}(bl, s)) = \mathit{update}(bl \oplus (al \triangleleft bl), s)$$

Theorem (3.1) is the semantic basis for the abstraction of commands by manipulating the text of the commands. If a command c_1 begins in a state s and has the assignments in list al, it will end in state $\mathit{update}(al, s)$. If a second command c_2 begins in this state and has assignment list bl, it will end in state $\mathit{update}(bl, \mathit{update}(al, s))$. From Theorem (3.1), the effect of the two commands on the state s is described by the assignment list $(al \oplus (bl \triangleleft al))$. This assignment list can be constructed from the syntax of the commands and used to construct an abstraction of commands c_1 and c_2. The command c with assignment list $(al \oplus (bl \triangleleft al))$ beginning in state s will produce the same state as the execution of c_1 followed by c_2. Command c is therefore the abstraction of c_1 followed by c_2 and can be constructed from the text of c_1 and c_2. For example, if expression e is interpreted in the state produced after by the execution of both c_1 and c_2 then its value (relative to state s) is $\mathcal{I}_e(e)(\mathit{update}(bl, \mathit{update}(al, s))$. This can be described syntactically as the expression $e \triangleleft (al \oplus (bl \triangleleft al))$.

Rules for Substitution

Rules for simplifying the expressions formed by the substitution operator are given in Figure (3.3), their proof is given in Section C.2.2 of the appendix. In Figure (3.3), every substitution expression is the substitution of value expressions (Definition 3.13). Rules (sr1) to (sr5) are the standard rules for substitution. Rules (sr6) and (sr7) describe substitution when the expression is a name function: the substitution is applied to the arguments; the function is evaluated to obtain a name x and the assignment list is searched for x. If x is a member of the list then its associated value is the result, otherwise the result is x. Rules (sr8) to (sr11) describe substitution and the combination of assignment lists: the substitution is carried out on the arguments to any functions; the name expressions are reduced to a name x and the assignment list is searched for a value assigned to x.

Example 3.14 Let s be a state and al the assignment list $(x_1, v_1) \cdot (x_2, v_2)$. Assume that x_1 is a basic name, $x_1 \in \mathit{Names}$ and is distinct from $x \in \mathit{Names}$, $x_1 \neq x$. Since both x_1 and x are constant, $\mathcal{I}_n(x_1)(s) = x_1$ and $\mathcal{I}_n(x)(s) = x$. The substitution $x \triangleleft al$ is therefore equivalent to $x \triangleleft (x_2, v_2)$.

Assume the registers and memory variables are distinct, $\mathit{Regs} \cap \mathit{Vars} = \{\}$, and that $a \in \mathcal{F}n$ constructs a name from the memory variables only, $\mathcal{I}_f(a)(v) = \mathit{name}(v)$ if $v \in \mathit{Vars}$. Also assume that x is constructed from a register, $r \in \mathit{Regs}$ and $x = \mathit{name}(r)$. For any $v \in \mathit{Values}$, $\mathcal{I}_f(a)(v) \neq x$ and, for all expressions $e \in \mathcal{E}$, $a(e) \not\equiv_s x$. If $x_1 = a(e)$ then $x \triangleleft al \equiv x \triangleleft (x_2, v_2)$. However, if x is the name expression $a(v)$ and $e \equiv v$ then $x \triangleleft al \equiv v_1$. □

$$e \triangleleft \mathit{nil} \equiv e \qquad \text{(sr1)} \qquad\qquad v \triangleleft al \equiv v \qquad \text{(sr2)}$$

$$\frac{x \equiv_s t}{x \triangleleft ((t, r) \cdot al) \equiv_s r} \quad \text{(sr3)} \qquad\qquad \frac{x \not\equiv_s t}{x \triangleleft ((t, r) \cdot al) \equiv_s x \triangleleft al} \quad \text{(sr4)}$$

$$\frac{f \notin \mathcal{F}_n}{f(a_1, \ldots, a_n) \triangleleft al \equiv f(a_1 \triangleleft al, \ldots, a_n \triangleleft al)} \quad \text{(sr5)}$$

$$\frac{f \in \mathcal{F}_n \quad f(a_1 \triangleleft ((t, r) \cdot al), \ldots, a_n \triangleleft ((t, r) \cdot al)) \equiv_s t}{f(a_1, \ldots, a_n) \triangleleft ((t, r) \cdot al) \equiv_s r} \quad \text{(sr6)}$$

$$\frac{\begin{array}{ll} & f \in \mathcal{F}_n \wedge v_1 \equiv_s a_1 \triangleleft ((t, r) \cdot al) \wedge \ldots \wedge v_n \equiv_s a_n \triangleleft ((t, r) \cdot al) \\ \wedge & f(a_1 \triangleleft ((t, r) \cdot al, \ldots, a_n \triangleleft ((t, r) \cdot al) \not\equiv_s t \end{array}}{f(a_1, \ldots, a_n) \triangleleft ((t, r) \cdot al) \equiv_s f(v_1, \ldots, v_n) \triangleleft al} \quad \text{(sr7)}$$

$$\frac{x \equiv_s t}{x \triangleleft (bl \oplus ((t, r) \cdot al) \equiv_s r} \quad \text{(sr8)} \qquad\qquad \frac{x \not\equiv_s t}{x \triangleleft (bl \oplus ((t, r) \cdot al) \equiv_s x \triangleleft (bl \oplus al)} \quad \text{(sr9)}$$

$$\frac{f \in \mathcal{F}_n \quad f(a_1 \triangleleft (bl \oplus (t, r) \cdot al), \ldots, a_n \triangleleft (bl \oplus (t, r) \cdot al)) \equiv_s t}{f(a_1, \ldots, a_n) \triangleleft (bl \oplus (t, r) \cdot al) \equiv_s r} \quad \text{(sr10)}$$

$$\frac{\begin{array}{ll} & f \in \mathcal{F}_n \wedge v_1 \equiv_s a_1 \triangleleft (bl \oplus (t, r) \cdot al) \wedge \ldots \wedge v_n \equiv_s a_n \triangleleft (bl \oplus (t, r) \cdot al) \\ \wedge & f(a_1 \triangleleft (bl \oplus (t, r) \cdot al, \ldots, a_n \triangleleft (bl \oplus (t, r) \cdot al) \not\equiv_s t \end{array}}{f(a_1, \ldots, a_n) \triangleleft (bl \oplus (t, r) \cdot al) \equiv_s f(v_1, \ldots, v_n) \triangleleft (bl \oplus al)} \quad \text{(sr11)}$$

where $v, v_1, \ldots, v_n \in \mathit{Values}, x \in \mathit{Names}, f \in \mathcal{F}$,
$t \in \mathcal{E}_n, r \in \mathcal{E}, e \in \mathcal{E}, a_1, \ldots, a_n \in \mathcal{E}, s \in \mathit{State}, al, bl \in \mathit{Alist}$,

Figure 3.3: Rules for the Substitution Operator

3.2 Commands of $\mathcal{L}$

The commands of the abstract language $\mathcal{L}$ describe the actions performed by the instructions in an object code program. The abstract language $\mathcal{L}$ has a *labelling*, a *conditional* and an *assignment* command. The labelling command associates $\mathcal{L}$ commands with labels; the value of the program counter *pc* in a state s determines whether a labelled command is selected for execution in s. The conditional command evaluates a Boolean expression and, depending on the result, executes one of two branches. The assignment command of $\mathcal{L}$ is a simultaneous assignment. This simplifies the description of instructions and is also necessary to support the abstraction of commands. However, it requires a method for detecting the unexecutable commands, which assign different values to a single name.

The three commands of $\mathcal{L}$ are sufficiently expressive to describe the result of abstracting from a sequence of commands. The changes made to a state by a sequences of assignments can be described by assignment lists and by substitution, as a consequence of Theorem (3.1). The tests on program variables which are carried out by a sequence of conditional commands can also be described in terms of conditional commands. The method used to abstract a sequence of commands is based on an operator for sequential composition. This is defined on the syntax of $\mathcal{L}$ commands and constructs the abstraction of a pair of commands. Repeated application of the sequential composition operator can then be used to abstract from a sequence of commands. Note that only commands are considered in this chapter; the identification of sequences of commands in a program is a matter for program transformations. The abstraction of commands simply provides the tools necessary for abstracting from programs.

The development of the commands of $\mathcal{L}$ will be as follows: the syntax of the commands will be defined first. The semantics of the assignment commands require a method for detecting unexecutable assignments. This will be described and followed by the definition of the semantics of the commands. The abstraction of commands will be described in Section 3.3.

3.2.1 Syntax of the Commands

The commands of $\mathcal{L}$ are defined as a set of commands $\mathcal{C}_0$ of which only a subset is needed to model processor instructions. This is to simplify the development of the language and its properties. The set $\mathcal{C}_0$ is inductively defined from labelling, conditional and assignment commands.

Definition 3.16 *Syntax of the commands of $\mathcal{L}$*

There is an inductively defined set $\mathcal{C}_0$ and functions:

$$\begin{aligned} \mathbf{if}\ \_\ \mathbf{then}\ \_\ \mathbf{else}\ \_ &: (\mathcal{E} \times \mathcal{C}_0 \times \mathcal{C}_0) \to \mathcal{C}_0 \\ :=\ \_, \_ &: (\mathit{Alist} \times \mathcal{E}_l) \to \mathcal{C}_0 \\ \_ : \_ &: (\mathit{Labels} \times \mathcal{C}_0) \to \mathcal{C}_0 \end{aligned}$$

The set $\mathcal{C}_0$ is defined:

$$\frac{e \in \mathcal{E} \quad c_1, c_2 \in \mathcal{C}_0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) \in \mathcal{C}_0} \qquad \frac{al \in \textit{Alist} \quad l \in \mathcal{E}_l}{:= (al, l) \in \mathcal{C}_0} \qquad \frac{l \in \textit{Labels} \quad c \in \mathcal{C}_0}{(l : c) \in \mathcal{C}_0}$$

The assignment command of $\mathcal{L}$ is $:= (al, l)$, the conditional command is **if** b **then** c_1 **else** c_2 and the labelling command is $(l : c)$ where $al \in \textit{Alist}, l \in \mathcal{E}_l, b \in \mathcal{E}_b$ and $c, c_1, c_2 \in \mathcal{C}_0$. The label of a labelled command $(l : c)$ is l, $\textit{label}(l : c) \stackrel{\text{def}}{=} l$. The *successor expression* of assignment command $:= (al, l)$ is l (a label expression).

An assignment command made up of a simple list will be written using infix notation. e.g. The command $:= ((x_1, e_1) \cdots (x_n, e_n) \cdot \textit{nil}, l)$ will be written $x_1, \ldots, x_n := e_1, \ldots, e_n, l$. □

All commands of $\mathcal{C}_0$ contain at least one assignment command and every assignment command assigns a value to the program counter, to select the next command for execution. An assignment command c is made up of an assignment list al and a label expression l. The command assigns the label expression l to the program counter *pc* simultaneously with assignments of al. The full list of assignments made by the command c is therefore the list $(\textit{pc}, l) \cdot al$. (In effect, the assignment command $x := y, l$ is short-hand for $x, \textit{pc} := y, l$.)

The commands of $\mathcal{L}$ used to model the instructions of an object code program are labelled commands in $\mathcal{C}_0$. The set containing these commands is denoted $\mathcal{C}$. Commands of the set $\mathcal{C}_0$ can be labelled with two or more different labels. If a command has two distinct labels, $l_1 : (l_2 : c)$ and $l_1 \neq l_2$, then the command cannot be executed: A command is selected by the value of the name *pc* and the name *pc* cannot have two values. These commands are excluded from the set $\mathcal{C}$.

Definition 3.17 *Commands of $\mathcal{L}$*

A command $c \in \mathcal{C}_0$ is *regular* if all labelled commands occurring in c have the same label.

$$\begin{aligned}
&\textit{regular?} : \mathcal{C}_0 \to \textit{boolean} \\
&\textit{regular?}(c) \stackrel{\text{def}}{=} \begin{cases} \forall (l_1, l_2 : \textit{Labels}, c_1, c_2 : \mathcal{C}_0) : \\ \qquad (l_1 : c_1) \ll c \wedge (l_2 : c_2) \ll c \Rightarrow l_1 = l_2 \end{cases}
\end{aligned}$$

The set $\mathcal{C}$ is the subset of $\mathcal{C}_0$ containing only labelled, regular commands:

$$\mathcal{C} \stackrel{\text{def}}{=} \{(l : c) \mid (l : c) \in \mathcal{C}_0 \wedge \textit{regular?}(l : c)\}$$

□

A command c which is selected in a state s can fail if it attempts an impossible assignment or it is labelled with two distinct labels. The labels of a command can be distinguished by syntactic equality and commands with distinct labels are excluded from set $\mathcal{C}$. This ensures that if a command in $\mathcal{C}$ fails, it does so because of an incorrect assignment.

$$
\begin{array}{rcl}
\text{value} & ::= & \text{any element of the set } \mathit{Values} \\
\text{name} & ::= & \text{any element of the set } \mathit{Names} \\
\text{label} & ::= & \text{any element of the set } \mathit{Labels} \\
\\
\text{value function} & ::= & \text{any element of the set } \mathcal{F}_v \\
\text{name function} & ::= & \text{any element of the set } \mathcal{F}_n \\
\text{label function} & ::= & \text{any element of the set } \mathcal{F}_l \\
\\
\mathcal{E} & ::= & \langle\text{value}\rangle \;|\; \langle\text{value function}\rangle(\langle\mathcal{E}\rangle, \ldots, \langle\mathcal{E}\rangle) \\
 & & |\; \langle\mathcal{E}_n\rangle \;|\; \langle\mathcal{E}_l\rangle \;|\; \langle\mathcal{E}\rangle \triangleleft \langle\mathit{Alist}\rangle \\
\mathcal{E}_n & ::= & \langle\text{name}\rangle \;|\; \langle\text{name function}\rangle(\langle\mathcal{E}\rangle, \ldots, \langle\mathcal{E}\rangle) \;|\; \langle\mathcal{E}_n\rangle \triangleleft \langle\mathit{Alist}\rangle \\
\mathcal{E}_l & ::= & \langle\text{label}\rangle \;|\; \langle\text{label function}\rangle(\langle\mathcal{E}\rangle, \ldots, \langle\mathcal{E}\rangle) \;|\; \langle\mathcal{E}_l\rangle \triangleleft \langle\mathit{Alist}\rangle \\
\\
\mathit{Alist} & ::= & \mathit{nil} \;|\; (\langle\mathcal{E}_n\rangle, \langle\mathcal{E}\rangle) \cdot \langle\mathit{Alist}\rangle \;|\; \langle\mathit{Alist}\rangle \oplus \langle\mathit{Alist}\rangle \\
\\
\text{com} & ::= & \textbf{if } \langle\mathcal{E}\rangle \textbf{ then } \langle\text{com}\rangle \textbf{ else } \langle\text{com}\rangle \;|\; := \langle\mathit{Alist}\rangle, \langle\mathcal{E}_l\rangle \\
\mathcal{C}_0 & ::= & \langle\text{com}\rangle \;|\; \langle\text{label}\rangle : \langle\text{com}\rangle \\
\mathcal{C} & ::= & \langle\text{label}\rangle : \langle\text{com}\rangle
\end{array}
$$

Figure 3.4: Summary of Syntax for Expressions and Commands of $\mathcal{L}$

Example 3.15 Assume assignment lists $al, bl \in \mathit{Alist}$, distinct labels $l_1, l_2 \in \mathit{Labels}$, label expression $l \in \mathcal{E}$ and Boolean expression $b \in \mathcal{E}_b$.

The commands of $\mathcal{C}_0$ include:

$$
\begin{array}{l}
:= (al, l), := (bl, l_1), := ((pc, l_1) \cdot al, l_2) \\
\textbf{if } b \textbf{ then } l_1 : (:= (al, l)) \textbf{ else } := (bl, l_1) \\
l_2 : \textbf{if } b \textbf{ then } l_1 : (:= (al, l)) \textbf{ else } l_2 : (:= (bl, l_1)) \\
l_1 : l_2 : (:= (al, l)) \\
l_1 : (:= (al, l)), l_2 : (:= (bl, l_1))
\end{array}
$$

Of these, the regular commands are:

$$
\begin{array}{l}
:= (al, l), := (bl, l_1), := ((pc, l_1) \cdot al, l_2) \\
\textbf{if } b \textbf{ then } l_1 : (:= (al, l)) \textbf{ else } := (bl, l_1) \\
l_1 : (:= (al, l)), l_2 : (:= (bl, l_1))
\end{array}
$$

The commands which are also in $\mathcal{C}$ are $l_1 : (:= (al, l))$ and $l_2 : (:= (bl, l_1))$. $\square$

Basic Commands of $\mathcal{L}$

A summary of the syntax of the commands and expressions of $\mathcal{L}$ is given in Figure (3.4). The syntactic category $\mathcal{C}$ describes the commands of the set $\mathcal{C}$, which will be used to model the instructions of an object code program. These are the commands labelled with a single label. The syntactic category $\mathcal{C}_0$ of Figure (3.4) describes the commands of the set $\mathcal{C}_0$, which will be used when deriving and manipulating commands of $\mathcal{L}$.

3.2.2 Correct Assignment Lists

The assignment command of $\mathcal{L}$ is a simultaneous assignment and its semantics require a means for detecting impossible assignments. The assignment command $:= (al, l)$ can be executed only if it does not assign two different values to the same name. The assignment list of the command, $(pc, l) \cdot al$, is said to be *correct* iff every name is assigned at most one value. Because every name expression in the assignment list of a command is evaluated in the state in which the command begins execution, the correctness of the assignment depends on this state.

An assignment list al can be constructed from the combination of assignment lists. However, it is only necessary for the simple lists occurring in al to be correct. If a name is assigned two different values by two different simple lists, only one of the assignments will be used. The correctness of an assignment list al is therefore established by considering each simple list which occurs in al. A simple assignment list is correct in a state s iff all values associated with a name x are equivalent in s. This allows a variable to be assigned the same value any number of times and is needed to accommodate name expressions. For example, the assignment $\mathbf{ref}(x), y := 1, 1$ would be otherwise be incorrect when $\mathbf{ref}(x) \equiv_s y$ even though the assignment would be equivalent in s to $y := 1$, a valid assignment.

Definition 3.18 *Correct assignment lists*

The name-value expressions pair $(x, v) \in (\mathcal{E}_n \times \mathcal{E})$ occurs in assignment list al in state s, written *occs?*$((x, v), al)(s)$, iff there is a pair (x_1, v_1) in al such that x_1 is equivalent to x and v_1 is equivalent to v.

$$\begin{aligned}
\mathit{occs?} &: ((\mathcal{E}_n \times \mathcal{E}) \times \mathit{Alist}) \to \mathit{State} \to \mathit{boolean} \\
\mathit{occs?}((x, e), \mathit{nil})(s) &\stackrel{\text{def}}{=} \mathit{false} \\
\mathit{occs?}((x, e), (x_1, e_1) \cdot al)(s) &\stackrel{\text{def}}{=} (x \equiv_s x_1 \wedge e \equiv_s e_1) \vee \mathit{occs?}((x, e), al)(s) \\
\mathit{occs?}((x, e), (al \oplus bl))(s) &\stackrel{\text{def}}{=} \mathit{occs?}((x, e), al)(s) \vee \mathit{occs?}((x, e), bl)(s)
\end{aligned}$$

The set of values associated with a name x in a state s by an assignment list al, *Assoc*$(x, al)(s)$ contains all values e such that (x, e) occurs in al in state s.

$$\begin{aligned}
&\mathit{Assoc} : (\mathcal{E}_n \times \mathit{Alist}) \to \mathit{State} \to \mathit{Set}(\mathcal{E}) \\
&\mathit{Assoc}(x, al)(s) \stackrel{\text{def}}{=} \{e : \mathcal{E} \mid \mathit{occs?}(x, e)(al)(s)\}
\end{aligned}$$

There is a predicate *correct?* on assignment lists and states with type *Alist* $\to$ *State* $\to$ *boolean* such that assignment list al is correct in state s iff *correct?*$(al)(s)$ is *true*.

Predicate *correct?* satisfies the following:
Simple lists: If al is a simple list, *simple?*(al), and every name is associated by al with at most one value in a state s then al is correct.

$$\mathit{correct?}(al)(s) \Leftrightarrow \begin{pmatrix} \forall(x : \mathit{Names}) : \\ \quad \exists(e : \mathcal{E}) : \forall(e_1 : \mathcal{E}) : e_1 \in \mathit{Assoc}(z, al)(s) \Rightarrow e \equiv_s e_1 \end{pmatrix}$$

Assignment lists: For any list al, if the initial prefix of al is correct in state s and every combination of lists in al is correct in state s then al is correct in state s.

$$\mathit{correct?}(al)(s) \Leftrightarrow \begin{pmatrix} \mathit{correct?}(\mathit{initial}(al))(s) \\ \land \forall(bl, cl : \mathit{Alist}) : \\ \quad (bl \oplus cl) \ll al \Rightarrow \mathit{correct?}(bl)(s) \land \mathit{correct?}(cl)(s) \end{pmatrix}$$

□

A definition for the predicate *correct?* is given in Appendix C. In a correct assignment list, each name is associated with at most one value by each simple list combined with the $\oplus$ operator. The correctness of a combined assignment list, $al \oplus bl$, depends on the correctness of al and bl independently of each other. The names in al are not considered when checking the list bl. This follows from the intended use of the combination construct to describe a sequence of state updates: bl describes the changes to the state which has been updated with al.

Corollary 3.1 *For any $s \in$ State, $x \in \mathcal{E}_n$, $e \in \mathcal{E}$ and $al, bl \in$ Alist,*

1. *The empty assignment list is always correct: correct?*$(\mathit{nil})(s)$.
2. *A single assignment is always correct: correct?*$((x, e) \cdot \mathit{nil})(s)$
3. *If correct?*$(al)(s)$ *and correct?*$(bl)(s)$ *then the combination of al and bl is correct in s: correct?*$(al \oplus bl)(s)$.

Proof. Straightforward, from definition of *correct?*. □

Corollary (3.1) describes simple cases which occur frequently. In particular, items (2) and (3) will be applicable to the assignment lists which result from the combination of commands. The correctness of an assignment list al in a state s updated with assignment list bl is preserved in state s when the assignment list bl is substituted into al. (This is needed to allow the abstraction of assignment commands using the property of Theorem 3.1.)

Theorem 3.2 *For assignment lists $al, bl \in$ Alist and state s,*

$$\mathit{correct?}(al)(\mathit{update}(bl, s)) \Leftrightarrow \mathit{correct?}(al \lhd bl)(s)$$

The predicate *correct?* is a precondition which must be satisfied by the state in which an assignment command begins execution. For processor languages, the majority of commands have assignment lists which are correct in any state. The means that, when verifying object code, the correctness of assignment lists in all states only needs to be established once. In general, it is not necessary to re-establish the correctness of an assignment list during the course of a verification proof.

Example 3.16 For $v_1, v_2 \in \mathcal{E}_n$ and expressions $e_1, e_2 \in \mathcal{E}$, the assignment list $(n_1, e_1) \cdot (n_2, e_2)$ is correct in state s if $n_1 \not\equiv_s n_2$, or if $n_1 \equiv_s n_2$ and $e_1 \equiv_s e_2$. If $n_1 \not\equiv_s n_2$ for any state s then the assignment list $(n_1, e_1) \cdot (n_2, e_2)$ is correct in any state.

For assignment lists al and bl and state s, if both al and bl are correct in s then so is $al \oplus bl$. Also, if al is correct in s and bl is correct in $update(al, s)$ then $al \oplus (bl \lhd al)$ is also correct in s. □

3.2.3 Semantics of the Commands

The semantics of the commands of $\mathcal{C}_0$ (and therefore $\mathcal{C}$) are defined by an interpretation function $\mathcal{I}_c$ which relates the state in which a command begins with the state it produces. A command c begins execution in state s and produces state t by assigning values to names. A command labelled with l can begin only if it is selected: the name pc must have the value l in state s. A conditional command with test b, true branch c_t and false branch c_f will execute c_t if the expression b is *true* in s; if b is *false* then c_f is executed. An assignment command produces state t by updating state s with the assignment list; if the assignment list is not correct, the assignment command fails.

Definition 3.19 *Semantics of the commands*

The interpretation function on commands has type:

$$\mathcal{I}_c : \mathcal{C}_0 \to (\mathit{State} \times \mathit{State}) \to \mathit{boolean}$$

The semantics of the commands are defined by $\mathcal{I}_c$ as follows:
Conditional commands:

$$\mathcal{I}_c(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2)(s,t) \stackrel{\text{def}}{=} \begin{cases} \mathcal{I}_c(c_1)(s,t) & \text{if } \mathcal{I}_b(b)(s) \\ \mathcal{I}_c(c_2)(s,t) & \text{otherwise} \end{cases}$$

Labelled commands:

$$\mathcal{I}_c(l : c)(s,t) \stackrel{\text{def}}{=} \mathcal{I}_e(pc)(s) = l \wedge \mathcal{I}_c(c)(s,t)$$

Assignment commands:

$$\mathcal{I}_c(:= (al, l))(s,t) \stackrel{\text{def}}{=} \mathit{correct?}((pc, l) \cdot al)(s) \wedge t = \mathit{update}((pc, l) \cdot al, s)$$

□

The commands of $\mathcal{L}$ are deterministic: if command c beginning in state s produces state t or state u then $t = u$. Because of this, the relations defining the semantics of the commands can also be considered to be functions transforming states.

Lemma 3.3 *Determinism*

For all commands $c \in \mathcal{C}_0$ and states $s, t, u \in$ State,

$$\frac{\mathcal{I}_c(c)(s,t) \quad \mathcal{I}_c(c)(s,u)}{t = u}$$

Proof. Straightforward, by induction on the command c. □

Example 3.17 Assume command $c_1 = (:= ((x_1, v_1) \cdot (x_2, v_2), l))$. Also assume that $x_1 \in \mathcal{E}_n$ is distinct from $x_2 \in \mathcal{E}_n$ in every state s, $x_1 \not\equiv_s x_2$ and *pc* is distinct from both x_1 and x_2 in any state s. The assignment list of c_1 is correct in any state. If c_1 begins in state s and produces state t, $\mathcal{I}_c(c_1)(s,t)$, then the value of x_1 in t is the value of v_1 in s, $\mathcal{I}_e(x_1)(t) = \mathcal{I}_e(v_1)(s)$. The value of x_2 in t is the value of v_2 in s, $\mathcal{I}_e(x_2)(t) = \mathcal{I}_e(v_2)(s)$.

Let $c_2 =$ **if** b **then** c_t **else** c_f. If b is equivalent in s to **true**, $b \equiv_s$ **true**, then c_2 is equivalent to c_t, $\mathcal{I}_c(c_2)(s,t) = \mathcal{I}_c(c_t)(s,t)$. □

Derived Commands

The set $\mathcal{C}_0$ contains the basic commands of the abstract language. Other commands, which describe a particular action, can be defined in terms of those in the set $\mathcal{C}_0$. In particular, the jump command, **goto**, and the command which always fails, written **abort**, can be derived from the assignment command. Both are useful when defining the semantics of processor instructions: a jump instruction occurs in most processor languages while the command **abort** models the failure of an instruction to terminate.

The jump and **abort** commands are defined in terms of assignments to variables. Command $:= (al, l)$ updates the program counter with the label expression l. If $\mathcal{I}_c(:= (al, l))(s,t)$ then $t = \mathit{update}((pc, l) \cdot al, s)$ and the value of *pc* in t is $\mathcal{I}_l(l)(s)$. An assignment command of the form $:= (\mathit{nil}, l)$ is therefore a computed jump: the value of expression l is assigned to *pc*. If the assignment list al of command $:= (al, l)$ is incorrect for all states then the command always fails to terminate and is the command **abort**.

Definition 3.20 goto *and* **abort**

The computed jump, **goto** l, is the assignment command with successor expression l. The command which always fails, **abort**, is the assignment command which is always incorrect.

$$\begin{array}{ll} \textbf{goto} : \mathcal{E}_l \to \mathcal{C}_0 & \textbf{abort} : \mathcal{C}_0 \\ \textbf{goto}\ l \stackrel{\text{def}}{=}\ := (\mathit{nil}, l) & \textbf{abort} \stackrel{\text{def}}{=} (pc, pc := \textbf{true}, \textbf{false}, \mathit{undef}(\mathcal{E}_l)) \end{array}$$

□

From the semantics of the assignment command, the assignment list of **goto** l beginning in states s must satisfy *correct?*$((pc, l) \cdot nil)(s)$. From Corollary (3.1), this is *true* in all states. Since **abort** assigns both **true** and **false** to the program counter, **abort** can never be executed.

Commands of $\mathcal{L}$ are labelled and a command c can begin in a state s iff the value of *pc* in s is the label of c. Following Lamport (1994), the command c is said to be *enabled* in s. When a command c of $\mathcal{L}$ is selected in state s, c must either terminate and update state s or fail to terminate. If c fails to terminate then it is said to *halt* in state s.

Definition 3.21 *Enabled and Halts*

A labelled command is enabled iff it is selected for execution.

$$\begin{array}{l} \mathit{enabled} : \mathcal{C} \to \mathit{State} \to \mathit{boolean} \\ \mathit{enabled}(c)(s) \stackrel{\text{def}}{=} \mathcal{I}_e(\mathit{pc})(s) = \mathit{label}(c) \end{array}$$

Command c halts in state s if c is enabled in s and there is no state in which c can terminate.

$$\begin{array}{l} \mathit{halt?} : \mathcal{C} \to \mathit{State} \to \mathit{boolean} \\ \mathit{halt?}(c)(s) \stackrel{\text{def}}{=} \mathit{enabled}(c)(s) \wedge \forall (t : \mathit{State}) : \neg \mathcal{I}_c(c)(s, t) \end{array}$$

□

For any label l, the command $(l$: **abort**$)$ halts whenever it is enabled. The command $(l$: **goto** $l)$ never halts: if it is enabled in state s then it will terminate and the state it produces is s. The value of the program counter *pc* in s is l and the command updates s with the assignment of l to *pc*, producing no change in s. The command l : **goto** l is therefore equivalent to the command *skip* which terminates but does nothing (Gries, 1981).

Example 3.18 *Processor instructions*

The semantics of a processor instruction can be described in terms of $\mathcal{L}$ either directly or through the use of derived commands, such as **goto**. For the example, the instructions described in Chapter 2 will be modelled in $\mathcal{L}$. Assume $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ are any distinct registers in *Regs*; also assume $v \in \mathit{Values}$ and $l \in \mathit{Labels}$. The label expression $\mathbf{loc}(\mathit{pc} +_a 1)$ will be used to informally identify the next instruction in memory (the exact expression will differ between processors).

Arithmetic and comparison instructions: The Alpha AXP instruction `addl` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ can be modelled by the $\mathcal{L}$ command $\mathbf{r}_3 := \mathbf{r}_1 +_a \mathbf{r}_2, \mathbf{loc}(\mathit{pc} +_a 1)$. The PowerPC instruction `add` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ is modelled by the $\mathcal{L}$ command $\mathbf{r}_1 := \mathbf{r}_2 +_a \mathbf{r}_3, \mathbf{loc}(\mathit{pc} +_a 1)$. The Alpha AXP instruction `cmpeq` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$, testing the equality of $\mathbf{r}_1$ and $\mathbf{r}_2$ is the $\mathcal{L}$ command **if** $\mathbf{r}_1 =_a \mathbf{r}_2$ **then** $\mathbf{r}_3 := 1, \mathbf{loc}(\mathit{pc} +_a 1)$ **else** $\mathbf{r}_3 := 0, \mathbf{loc}(\mathit{pc} +_a 1)$.

Program control instructions: The conditional jump instruction of the Alpha AXP, `beq` $\mathbf{r}, v$, can be modelled as the conditional command **if** $\mathbf{r} =_a 1$ **then goto** $\mathbf{loc}(\mathit{pc} +_a v)$ **else goto** $(\mathit{pc} +_a 1)$. Many jump instructions can be described in terms of the **goto** command. For example, a simple branch instruction of the Motorola 68000 processor, `jmp` v, passes control to the instruction labelled v; this is the $\mathcal{L}$ command **goto** $\mathbf{loc}(v)$. Note that, by definition, this is the assignment command $:= (nil, \mathbf{loc}(v))$. Program control instructions can be more complex. For example, the Alpha AXP instruction `jmp` $\mathbf{r}, v$, assigns the label of the next instruction to register $\mathbf{r}$ and passes control to the label $\mathbf{loc}(v)$: $\mathbf{r} := \mathbf{loc}(pc +_a 1), \mathbf{loc}(v)$.

Data movement: The Motorola 68000 data movement instruction, `move.w` $\#258, \mathbf{r}@$, is simply an assignment to two memory variables. These will be identified in terms of **ref**. The $\mathcal{L}$ command modelling the instruction is: $\mathbf{ref}(\mathbf{r}), \mathbf{ref}(\mathbf{r} +_a 1) := 1, 3, \mathbf{loc}(pc +_a 1)$. Note that this is a simultaneous assignment to name expressions and these name expressions implement the indirect addressing modes of the instruction. The assignment to $\mathbf{ref}(\mathbf{r})$ and $\mathbf{ref}(\mathbf{r} +_a 1)$ is correct in any state because the $\mathbf{r}$ and $\mathbf{r} +_a 1$ are distinct in any state (by definition of **ref**).

The commands of $\mathcal{L}$ are not limited by the instructions of a processor and can model instructions formed by combining the features of different processor languages. For example, Alpha AXP processor does not permit arithmetic operations to refer to memory variables. However, these can easily be modelled in $\mathcal{L}$. For example the addition of memory variables identified by registers is the assignment: $\mathbf{ref}(\mathbf{r}_3) := \mathbf{ref}(\mathbf{r}_1) +_a \mathbf{ref}(\mathbf{r}_2), \mathbf{loc}(pc +_a 1)$. This action is not supported by the Alpha AXP processor design, which restricts arithmetic operations to registers and values. □

3.3 Abstraction of Commands

A command c is an abstraction of the two $\mathcal{L}$ commands c_1 and c_2 if it produces the same state produced by c_1 followed by c_2. Formally, for any states s and t, c must satisfy:

$$\frac{\exists(u : \mathit{State}) : \mathcal{I}_c(c_1)(s, u) \wedge \mathcal{I}_c(c_2)(u, t)}{\mathcal{I}_c(c)(s, t)} \tag{3.1}$$

Because c_2 does not necessarily follow c_1, which may select any command, c must also have the property that if c_1 does not select c_2 then c has the same behaviour as c_1.

$$\frac{\mathcal{I}_c(c_1)(s, t) \quad pc \not\equiv_t \mathit{label}(c_1)}{\mathcal{I}_c(c)(s, t) = \mathcal{I}_c(c_1)(s, t)} \tag{3.2}$$

Any command which has both these properties can replace c_1 in a program. While this will not reduce the number of commands of the program, it will reduce the number of commands which must be considered during a proof of correctness.

The sequential composition operator, used in structured languages (Loeckx & Sieber, 1987), is used to combine two commands. In a structured language, this operator is a syntactic construct which constructs a compound command, written $c_1; c_2$, from the commands c_1 and c_2. The interpretation of this construct would be defined:

$$\mathcal{I}_c(c_1; c_2)(s, t) = \exists u : \mathcal{I}_c(c_1)(s, u) \wedge \mathcal{I}_c(c_2)(u, t) \tag{3.3}$$

which satisfies Property (3.1) but not Property (3.2). For $c_1 = \mathbf{goto}\ l_1$, $c_2 = l2 : \mathbf{goto}\ l_1$ and $l_1 \neq l_2$, the interpretation would never hold.

The interpretation of sequential composition using Equation (3.3) assumes that the language does not contain labelled commands. When labelled commands are included, the interpretation can be defined, using Property (3.1) and Property (3.2), as follows:

$$
\mathcal{I}_c(c_1; c_2)(s,t) \stackrel{\text{def}}{=} \begin{cases} \exists u : \mathcal{I}_c(c_1)(s,u) \\ \qquad \wedge\ \textit{labelled?}(c_2) \wedge \textit{enabled}(c_2)(u) \Rightarrow \mathcal{I}_c(c_2)(u,t) \\ \qquad \wedge\ \textit{labelled?}(c_2) \wedge \neg\textit{enabled}(c_2)(u) \Rightarrow u = t \\ \qquad \wedge\ \neg\textit{labelled?}(c_2) \Rightarrow \mathcal{I}_c(c_2)(u,t) \end{cases} \tag{3.4}
$$

(where $\textit{labelled?}(c) = \textit{true} \Leftrightarrow (\exists l', c' : c = (l' : c')))$

This definition of sequential composition is consistent with that used in languages without jumps and labels. However, it does not construct an abstraction of the two commands c_1 and c_2. This form of composition is simply a syntactic device for the combination of commands. The interpretation of this form of composition depends on the interpretation of the two individual commands. To verify that the composed commands establish a property, each command must therefore be considered separately.

To construct an abstraction which reduces the verification task, sequential composition must result in a single command. The sequential composition operator is defined here as a function on commands of $\mathcal{L}$ which ranges over the commands. This function is based on the algebraic laws described by Hoare et al. (1987), extended to take into account name expressions and computed jumps. The substitution expressions of $\mathcal{L}$ and the combination of assignment lists provide the operations on name expressions. To treat the flow of control correctly, the sequential composition of command c_1 and c_2 uses the label of c_2 to guard execution of c_2. A conditional command is formed which compares the program counter with the label of c_2. If the two are equal then $c_1; c_2$ is the command satisfying Property (3.1) otherwise it is the command satisfying Property (3.2).

3.3.1 Sequential Composition

The sequential composition of commands c_1 and c_2 is written $c_1; c_2$ and defined by recursion over the commands of the set $\mathcal{C}_0$. For simplicity, the definition will be given in a number of steps.

Definition 3.22 *Type of the sequential composition operator*

The composition operator ; is a function from a pair of commands to a single command.

$$
\_;\_ : ((\mathcal{C}_0 \times \mathcal{C}_0) \to \mathcal{C}_0)
$$

□

If c_1 is a labelled command, c_2 is composed with the command being labelled. When c_1 is a conditional, both branches are composed with c_2. When c_1 is an assignment command, the definition is by recursion over c_2.

Definition 3.23 *Labelled and conditional commands*

The composition of $l : c_1$ and c_2 is defined:

$$(l : c_1); c_2 \stackrel{\text{def}}{=} l : (c_1; c_2)$$

The composition of command $(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2)$ with command c is the composition of the branches with c.

$$(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2); c \stackrel{\text{def}}{=} \textbf{if}\ b\ \textbf{then}\ (c_1; c)\ \textbf{else}\ (c_2; c)$$

□

The properties of composition of a labelled or conditional commands are straightforward from the semantics of the commands. If c is any command, the composition of a labelled command $(l : c_1)$ with c, $(l : c_1; c)$, has label l and is enabled in a state s iff $(l : c_1)$ is enabled in s. The composition of a conditional command **if** b **then** c_t **else** c_f with c is equivalent to $(c_t; c)$ whenever test b succeeds and is equivalent to $(c_f; c)$ whenever test b fails.

Composition and Assignment

An assignment command updates the state in which it starts with new values for some of the names. Any command c following an assignment will begin in the updated state and any expression in c is evaluated in the updated state. This is equivalent to substituting the values assigned to the names in the expressions of c. When the assignment command is composed with a labelled command $(l_1 : c)$, the command c can be executed only if the successor expression of the assignment command is equivalent to l. The command formed by sequential composition uses the equality operator of $\mathcal{E}$, **equal**, to test the equivalence of the program counter and the successor expression in a state.

Corollary 3.2 *Equality operator of $\mathcal{L}$ and equivalence*

For expressions $e_1, e_2 \in \mathcal{E}$ and state s: $\mathcal{I}_b(e_1 =_a e_2)(s) \Leftrightarrow e_1 \equiv_s e_2$.

Proof. Immediate, from definitions. □

When an assignment command $:= (al, l)$ is composed with a conditional command, the Boolean test is updated with the assignment list and the assignment command is composed with each branch of the conditional.

Definition 3.24 *Assignment with labelled or conditional commands*

The composition of an assignment with a labelled command is defined:

$$(:= al, l); (l : c) \stackrel{\text{def}}{=} \begin{cases} \textbf{if}\ (l =_a l_1)\ \textbf{then}\ (:= al, l); c \\ \qquad\qquad\quad \textbf{else}\ (:= al, l) \end{cases}$$

The composition of an assignment with a conditional command is defined:

$$(:= al, l); (\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) \stackrel{\text{def}}{=} \begin{cases} \textbf{if } b \lhd ((pc, l) \cdot al) \textbf{ then } (:= al, l); c_1 \\ \qquad\qquad\qquad\quad \textbf{else } (:= al, l); c_2 \end{cases}$$

□

The composition of two assignment commands $:= (al, l_1)$ and $:= (bl, l_2)$, is obtained by: substituting l_1 for every occurrence of the program counter *pc* in the expressions of *bl*; substituting each expression in al for its associated name when it occurs in the expressions of *bl*; combining the two resulting lists. This yields the assignment list of Theorem (3.1), which describes the changes made by the two commands to a state.

Definition 3.25 *Composition of assignment commands*

The composition of two assignment commands $:= (al, l_1)$ and $:= (bl, l_2)$ is the assignment command which updates the states with the assignments made by $:= (al, l_1)$ followed by $:= (bl, l_2)$.

$$:= (al, l_1); (bl, l_2) \stackrel{\text{def}}{=} (((pc, l_1) \cdot al) \oplus ((pc, l_2) \cdot bl \lhd (pc, l_1) \cdot al), l_2 \lhd ((pc, l_1) \cdot al))$$

□

The semantics of the assignment command require that the assignment list of $:= (al, l_1); := (bl, l_2)$ is correct:

$$\mathit{correct?}((pc, l_2 \lhd ((pc, l_1) \cdot al)) \cdot (((pc, l_1) \cdot al) \oplus ((pc, l_2) \cdot bl \lhd (pc, l_1) \cdot al)))$$

From the rules for predicate *correct?* (Corollary 3.1), it is enough to show the correctness of al, of $bl \lhd ((pc, l_1) \cdot al)$ and of $(pc, l_2 \lhd ((pc, l_1) \cdot al)) \cdot nil$. Assume that the first command begins in state s and ends in state u and that the second command begins in state u and ends in state t. Assume also that $:= (al, l_1); := (bl, l_2)$ begin and ends in state s and t respectively. The assignment list al is correct in state s since $:= (al, l_1)$ begins in s and terminates. The correctness of assignment list $(bl \lhd (pc, l_1) \cdot al)$ in state s is equivalent to the correctness of *bl* in u and follows from the semantics of $:= (bl, l_2)$ and the correctness of $(pc, l_2 \lhd ((pc, l_1) \cdot al)) \cdot nil$ is immediate.

3.3.2 Properties of Composition

Since a command of $\mathcal{L}$ is a labelled, regular command of $\mathcal{C}_0$, the composition of commands $c_1, c_2 \in \mathcal{C}$ is also in $\mathcal{C}$ iff $c_1; c_2$ is a regular command. This is ensured by the composition of an assignment with a labelled command which replaces the labelling construct with a conditional command.

Theorem 3.3 *The result of composition of a regular command $(l : c) \in \mathcal{C}_0$ with any command $c_2 \in \mathcal{C}_0$ is a regular command.*

$$\frac{\textit{regular?}(l : c_1)}{\textit{regular?}((l : c_1); c_2)}$$

Sequential composition satisfies Property (3.1) and Property (3.2). For any two commands $c_1, c_2 \in \mathcal{C}_0$ and states s, u, t, if c_1 begin in state s and ends in state u and c_2 begins in state u and ends in state t then $c_1; c_2$ begins in state s and ends in state t.

Theorem 3.4 *Property (3.1)*

For any commands $c_1, c_2 \in \mathcal{C}_0$ and states s, t,

$$\frac{\exists(u : \textit{State}) : \mathcal{I}_c(c_1)(s, u) \wedge \mathcal{I}_c(c_2)(u, t)}{\mathcal{I}_c(c_1; c_2)(s, t)}$$

For any commands $c_1 \in \mathcal{C}_0, c_2 \in \mathcal{C}$ and states s, t, if c_1 begins in state s and ends in state t and c_2 is not enabled in t then $c_1; c_2$ begins in state s and ends in state t.

Theorem 3.5 *Property (3.2)*

For any commands $c_1, c_2 \in \mathcal{C}_0$, label $l \in$ Labels and states $s, t \in$ State,

$$\frac{\mathcal{I}_c(c_1)(s, t) \quad pc \not\equiv_t l}{\mathcal{I}_c(c_1; (l : c_2))(s, t) = \mathcal{I}_c(c_1)(s, t)}$$

Assume c is the result of composing any two commands c_1 and c_2. The interpretation of c in states s and t implies that either there is an intermediate state in which c_1 terminates and c_2 begins or c_1 begins in state s and ends in t.

Theorem 3.6 *For commands $c_1, c_2 \in \mathcal{C}_0$ and states s, t,*

$$\frac{\mathcal{I}_c(c_1; c_2)(s, t)}{(\exists(u : \textit{State}) : \mathcal{I}_c(c_1)(s, u) \wedge \mathcal{I}_c(c_2)(u, t)) \vee \mathcal{I}_c(c_1)(s, t)}$$

Theorem (3.6) is a general result which relates the interpretation of $c_1; c_2$ to the effect of attempting to execute c_1 followed by c_2. Either both are executed and there is an intermediate state between c_1 and c_2 or only c_1 is executed to produce state t. It is not possible to state that c_2 is not enabled in t since c_2 (which is in $\mathcal{C}_0$) may have more than one label. Because the composition function translates the labelling construct into conditionals, with the failure of the test leading to the behaviour of c_1, it is possible for c_2 to be enabled in t but never executed. For example, composing any command c with $l_1 : l_2 : c'$ where $l_1 \neq l_2$ results in the command:

$$\textbf{if}\ pc = l_1\ \textbf{then}\ (\textbf{if}\ pc = l_2\ \textbf{then}\ c; c'\ \textbf{else}\ c)\ \textbf{else}\ c$$

Since $l_1 \neq l_2$, one or both of the tests will fail and result in a command which behaves as c. When the commands are regular, every label occurring in the command must be the same and Theorem (3.6) can be strengthened.

Theorem 3.7 *For regular commands $c_1, c_2 \in \mathcal{C}$ and states s, t,*

$$\frac{\mathcal{I}_c(c_1; c_2)(s, t)}{(\exists(u : \mathit{State}) : \mathcal{I}_c(c_1)(s, u) \wedge \mathcal{I}_c(c_2)(u, t)) \vee (\mathcal{I}_c(c_1)(s, t) \wedge \mathit{pc} \not\equiv_t \mathit{label}(c_2))}$$

Theorems (3.4) to (3.7) describe the behaviour of composition when both commands are executed and terminate. Composition also preserves the failures of the commands: if the composition of commands c_1 and c_2 halts then either c_1 or c_2 must also fail to terminate.

Theorem 3.8 *For commands $c_1, c_2 \in \mathcal{C}$ and states $s, t, u \in \mathit{State}$,*

$$\frac{\mathit{halt?}(c_1; c_2)(s)}{\mathit{halt?}(c_1)(s) \vee (\exists u : \mathcal{I}_c(c_1)(s, u) \wedge \mathit{halt?}(c_2)(u))}$$

Conversely, if either command c_1 or c_2 halts then so does $(c_1; c_2)$.

Theorem 3.9 *For commands $c_1, c_2 \in \mathcal{C}$ and states $s, t \in \mathit{State}$,*

1. *If c_1 halts in s then so does $c_1; c_2$.*

$$\frac{\mathit{halt?}(c_1)(s)}{\mathit{halt?}(c_1; c_2)(s)}$$

2. *If c_1 beginning in state s ends in a state u and c_2 halts in u then $c_1; c_2$ halts in s.*

$$\frac{\mathcal{I}_c(c_1)(s, t) \quad \mathit{halt?}(c_2)(t)}{\mathit{halt?}(c_1; c_2)(s)}$$

Theorems (3.8) and (3.9), together with the earlier theorems, show that if the composition $c_1; c_2$ establishes a property then so will the two commands c_1 and c_2 executed in sequence. If the composition $(c_1; c_2)$ cannot be executed or cannot establish the property then neither can the two commands considered individually. Verification based on $(c_1; c_2)$ is therefore equivalent to and simpler than considering c_1 and c_2 individually because any properties of the two commands executed in sequence can be established directly from the single command $(c_1; c_2)$.

3.3.3 Applying Sequential Composition

The commands which result from sequential composition can be complex. For example, to ensure that composition has Property (3.2), an assignment command c_1 composed with labelled command $l : c_2$ results in a conditional command in which both c_1 and $(c_1; c_2)$ occur. However, the result of sequential composition can, in some circumstances, be simplified using the properties of the expressions and commands.

The simplification of a command c must ensure the result c' is equivalent to c in all states, $\mathcal{I}_c(c)(s,t) = \mathcal{I}_c(c')(s,t)$. This is possible by the replacement of expressions in the command with strongly equivalent expressions. This can then be followed by the replacement of commands with equivalent commands. The conditions for strong equivalence between expressions can often be determined from the syntax of the commands; in these cases, the simplifications can be carried out mechanically. For example, assume $l_1 = l$: in the conditional command **if** $(l_1 =_a l)$ **then** $c_1; c_2$ **else** c_1, the expression $(l_1 =_a l)$ can be replaced with **true**. From the semantics of the conditional command, the result is a command equivalent to $(c_1; c_2)$.

A second method, essentially that of symbolic execution (King, 1971), uses the properties of the commands to determine the values of expressions. e.g. Assume $x \in$ *Names* and $v \in$ *Values* and commands c_t, c_f. From $\mathcal{I}_c($**if** $x =_a v$ **then** c_t **else** $c_f)(s,t)$ it follows that if x occurs in c_t then it can be replaced with v and where x occurs in c_f, x will not have value v.

Example 3.19 Assume name $x \in$ *Names*, values $v_1, v_2 \in$ *Values*, labels $l, l_1 \in$ *Labels* and expressions $e_1, e_2 \in \mathcal{E}$. Let $a \in \mathcal{F}n$ be defined such that for all $v \in$ *Values* and $s \in$ *State*, $a(v)$ is distinct from x in s. Also assume for $v_1, v_2 \in$ *Values* that $a(v_1) \equiv_s a(v_2)$ iff $v_1 = v_2$.

The assignment command $:= ((x, e_1) \triangleleft (a(e_2), v_2), l)$ is equivalent to $:= (x, e_1 \triangleleft (a(e_2), v_2), l)$. The assignment command $:= ((x, v_1) \triangleleft (a(e_2), v_2), l)$ is equivalent to $:= ((x, v_1), l)$.

The expression **ref**$(a(x)) \triangleleft (a(v_2), v_1)$ is equivalent to **ref**(v_1) when x has the value v_2. The conditional command:

$$\begin{array}{l} \textbf{if}\ x =_a v_2\ \textbf{then}\ (\textbf{ref}(a(x)) \triangleleft (a(v_2), v_1) := e_1, l) \\ \qquad\qquad\quad \textbf{else}\ (\textbf{ref}(a(x)) \triangleleft (a(v_2), v_1) := e_2, l) \end{array}$$

is equivalent to the command:

$$\begin{array}{l} \textbf{if}\ x =_a v_2\ \textbf{then}\ (\textbf{ref}(v_1) := e_1, l) \\ \qquad\qquad\quad \textbf{else}\ (\textbf{ref}(a(x) \triangleleft (a(v_2), v_1) := e_2, l)) \end{array}$$

□

A property of sequential composition with labelled commands is that it is not associative (as it is in structured languages). The composition of command $l_1 : c_1$ with c_2, $(l_1 : c_1); c_2$ has label l_1 and is enabled iff $l_1 : c_1$ is enabled. The composition of any command c with $(l_1 : c_1); c_2$, $c; ((l_1 : c_1); c_2)$, will select $((l_1 : c_1); c_2)$ iff c selects $l_1 : c_1$ and will behave as c otherwise. Even if c selects c_2, c_2 will not be executed since it can only follow $l_1 : c_1$. The composition of c with $l_1 : c_1$ then with c_2, $(c; l_1 : c_1); c_2$ has a different behaviour. First c is executed then, if $l_1 : c_1$ is enabled, the command $l_1 : c_1; c_2$ is executed. If $l_1 : c_1$ is not selected and c selects c_2 then c_2 will be executed.

Example 3.20 Let $l_1, l_2, l_3, l_4 \in$ *Labels* be distinct. The command **goto** $l_2; (l_1 :$ **goto** $l_3; l_2 :$ **goto** $l_4)$ is equivalent to **goto** l_2. However, the command $($**goto** $l_2; l_1 :$ **goto** $l_3); l_2 :$ **goto** l_4 is equivalent to **goto** $l_2; l_2 :$ **goto** l_4. □

3.4 Proof Rules for Commands

A program logic provides a means for reasoning about the the commands of a language. The proof rules of the logic are used when verifying a program, to show that the commands of the program satisfy some specification. Because the commands of the language $\mathcal{L}$ are used to model processor instructions, proof rules for $\mathcal{L}$ commands can be applied to any processor instruction which is described in terms of $\mathcal{L}$.

Proof rules are defined for a program logic and are applied to logical formulas describing the specification of commands. The rules for commands of $\mathcal{L}$ will be defined for a logic made up of assertions on states. The specification of a command will be based on a *wp* function (Dijkstra, 1976) for commands in the set $\mathcal{C}_0$. This will allow the total correctness of commands to be established. The proof rules for the commands will be based on the manipulation of the specifications formed by this *wp* function.

Assertions

The properties of states will be specified in terms of an assertion language, denoted $\mathcal{A}$. An assertion is a predicate on states; the set $\mathcal{A}$ is made of the assertions, the operators of a first order logic and the Boolean expressions $\mathcal{E}_b$. The logical operators of the assertion language include the negation, conjunction and universal quantifier. The assertion language also includes a substitution operator for the expressions of $\mathcal{L}$.

Definition 3.26 *Assertion language*

Assertions have type $\mathcal{A}$, defined as the functions from states to Booleans.

$$\mathcal{A} \stackrel{\text{def}}{=} \mathit{State} \to \mathit{boolean}$$

An assertion P is *valid* if it is *true* for all states. A valid assertion is written $\vdash P$.

$$\begin{aligned} &\vdash \_ : \mathcal{A} \to \mathit{boolean} \\ &\vdash P \stackrel{\text{def}}{=} \forall (s : \mathit{State}) : P(s) \end{aligned}$$

The negation and conjunction of assertions are defined

$$\begin{array}{ll} \neg\_ : \mathcal{A} \to \mathcal{A} & \_ \wedge \_ : (\mathcal{A} \times \mathcal{A}) \to \mathcal{A} \\ \neg P \stackrel{\text{def}}{=} \lambda(s : \mathit{State}) : \neg P(s) & P \wedge Q \stackrel{\text{def}}{=} \lambda(s : \mathit{State}) : P(s) \wedge Q(s) \end{array}$$

The disjunction, $\vee$, and implication, $\Rightarrow$, operators can be defined in terms of the negation and conjunction.

The universal quantifier is defined on functions from values to assertions.

$$\begin{aligned} &\forall : (\mathit{Values} \to \mathcal{A}) \to \mathcal{A} \\ &\forall F \stackrel{\text{def}}{=} \lambda(s : \mathit{State}) : \forall (v : \mathit{Values}) : F(v)(s) \end{aligned}$$

Substitution in assertions is equivalent to updating the state.

$$
\begin{aligned}
&\_ \lhd \_ : (\mathcal{A} \times \mathit{Alist}) \to \mathcal{A}\\
&P \lhd al \stackrel{\text{def}}{=} \lambda(s : \mathit{State}) : P(\mathit{update}(al, s))
\end{aligned}
$$

The Boolean expressions $\mathcal{E}_b$ are taken to be assertions; $e \in \mathcal{E}_b \Rightarrow \mathcal{I}_b(e) \in \mathcal{A}$. For $e \in \mathcal{E}_b$, the assertion $\mathcal{I}_b(e)$ will be written e. □

The set $\mathcal{A}$ describes a first order logic which can be used to reason about the properties of the program variables. Proof rules for the logic are defined in terms of valid formulas, as usual in logical systems (e.g. see Paulson, 1995a or Gordon, 1988). To distinguish the assertions of $\mathcal{A}$ from the logical formulas used in the presentation, universally quantified assertions of $\mathcal{A}$ will be written using lambda notation. For example, with $v \in \mathit{Values}$, $\forall(\lambda v : v +_a 1 >_a v)$ is an assertion of $\mathcal{A}$ which is *true* for any state: $\vdash \forall(\lambda v : v +_a 1 >_a v)$. Assume $x, y \in \mathit{Names}$, the assertion $\forall(\lambda v : x =_a v \Rightarrow v >_a y)$, is *true* for any state in which the assertion $x >_a y$ is *true*: $\vdash (x >_a y) \Rightarrow \forall(\lambda v : x =_a v \Rightarrow v >_a y)$. The existential quantifier of $\mathcal{A}$ can be defined $\exists F \stackrel{\text{def}}{=} \neg\forall(\lambda v : \neg F(v))$, where $F : \mathit{Values} \to \mathcal{A}$.

Weakest Precondition of Commands

The commands of $\mathcal{L}$ are specified in terms of a *wp* predicate transformer (Dijkstra, 1976) which defines the weakest precondition necessary for a command to terminate and establish a postcondition. The *wp* transformer for the commands of $\mathcal{L}$ is written **wp** and satisfies, for postcondition Q, command c and states s, t:

$$\mathbf{wp}(c, Q)(s) = \exists t : \mathcal{I}_c(c)(s, t) \wedge Q(t)$$

The weakest precondition required for command c to establish postcondition Q is calculated from the weakest precondition required by each command occurring in c.

Definition 3.27 *Weakest precondition*

For assertion Q, commands c, c_1, c_2 and state s, t, **wp** is defined

$$
\begin{aligned}
\mathbf{wp} &: (\mathcal{C}_0 \times \mathcal{A}) \to \mathcal{A}\\
\mathbf{wp}(l : c, Q) &\stackrel{\text{def}}{=} \mathit{pc} =_a l \wedge \mathbf{wp}(c, Q)\\
\mathbf{wp}((:= al, l), Q) &\stackrel{\text{def}}{=} \begin{cases} \quad \mathit{correct?}((\mathit{pc}, l) \cdot al) \\ \wedge Q \lhd (\mathit{pc}, l) \cdot al \end{cases}\\
\mathbf{wp}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, Q) &\stackrel{\text{def}}{=} \begin{cases} \quad b \Rightarrow \mathbf{wp}(c_1, Q) \\ \wedge \neg b \Rightarrow \mathbf{wp}(c_2, Q) \end{cases}
\end{aligned}
$$

□

The rules for the weakest precondition are given in Figure (3.5). The assignment rule (tl1) is similar to that of Cartwright and Oppen (1981) (except that the assignment is simultaneous) and requires that the assignment list of the command be correct. The label rule (tl2) requires that a labelled command is selected before it is executed. Rules (tl3) and (tl4) are the standard rules for conditional commands and for the composition of commands. The proof of the rules is straightforward by induction on the commands.

There is no rule for sequential composition since it is not a primitive construct of the language $\mathcal{L}$. The result of combining commands by sequential composition is a single command made up of the labelling, conditional and assignment commands, to which the rules of Figure (3.5) can be applied. The rule for the sequential composition operator of structured languages (Dijkstra, 1976) is:

$$\frac{\vdash P \Rightarrow \mathbf{wp}(c_1, R) \quad \vdash R \Rightarrow \mathbf{wp}(c_2, Q)}{\vdash P \Rightarrow \mathbf{wp}(c_1; c_2, Q)}$$

This rule can be obtained as an instance of Theorem (3.4). However, the rule increases the difficulty of a proof, requiring commands c_1 and c_2 to be considered individually. The simpler approach is to reason about the single command which results from $(c_1; c_2)$. The sequential composition operator of $\mathcal{L}$ abstracts from c_1 and c_2 allowing the truth of $\vdash P \Rightarrow \mathbf{wp}(c_1; c_2, Q)$ to be established directly, without the need for the intermediate assertion R.

3.4.1 Example

Specifying commands in terms of logical formulas, as in the *wp* calculus, allows the verification to be based on the properties required of the commands. Reasoning about commands specified using the **wp** function is similar to reasoning in any *wp* calculus (e.g. see Dijkstra, 1976 or Gries, 1981). The examples here will consider the features specific to the language $\mathcal{L}$. For the examples, assume $P, Q \in \mathcal{A}$,$c, c_1 \in \mathcal{C}_0$, $x, y, z \in$ *Names*, $v, v_1, v_2 \in$ *Values*, $e_1, e_2 \in \mathcal{E}$ and $l, l_1, l_2 \in$ *Labels*. Also assume x, y, z and the program counter *pc* are all distinct.

Assignments: The assignment command $x, y := v_1, v_2, l$ can be shown to establish postcondition $x =_a v_1$: $\vdash \mathit{true} \Rightarrow \mathbf{wp}((x, y := v_1, v_2, l), x =_a v_1)$. First, the precondition is strengthened with the assertion $x =_a v_1 \lhd (\mathit{pc}, l) \cdot (x, v_1) \cdot (y, v_2)$, which requires a proof of $\vdash x =_a v_1 \lhd (\mathit{pc}, l) \cdot (x, v_1) \cdot (y, v_2) \Rightarrow \mathit{true}$. This can be established using the substitution rules of Figure (3.3) and Lemma (3.1) (for the properties of the equivalence relation). The substitution is pushed into the equality (since $v_1 \in$ *Values*) to obtain $\vdash x \lhd (\mathit{pc}, l) \cdot (x, v_1) \cdot (y, v_2) =_a v_1$. Since x is distinct from both y and *pc*, this reduces to $\vdash v_1 =_a v_1$, which is trivially *true*. The specification to be proved is then $\vdash x =_a v_1 \lhd (\mathit{pc}, l) \cdot (x, v_1) \cdot (y, v_2) \Rightarrow \mathbf{wp}((x, y := v_1, v_2, l), x =_a v_1)$. This is immediate from the assignment rule (tl1) (the assignment list is always correct since the names are distinct).

Flow of control: The selection of commands for execution is determined by the labelling command and the program counter *pc*. For labelled command $l : c \in \mathcal{C}$ to satisfy the specification $\vdash P \Rightarrow \mathbf{wp}(l : c, Q)$, the precondition P must satisfy $\vdash P \Rightarrow \mathit{pc} =_a l$ (from the label rule tl2). For command c_1 to select $l : c$ as a successor, c_1 must assign l to the program counter. If c

Assignment: $$\frac{\vdash P \triangleleft (pc, l)\cdot al \Rightarrow \mathit{correct?}((pc, l)\cdot al)}{\vdash P \triangleleft ((pc, l)\cdot al) \Rightarrow \mathbf{wp}(:= (al, l), P)} \quad \text{(tl1)}$$

Label: $$\frac{\vdash P \Rightarrow pc =_a l \wedge \mathbf{wp}(c, Q)}{\vdash P \Rightarrow \mathbf{wp}(l : c, Q)} \quad \text{(tl2)}$$

Conditional: $$\frac{\begin{array}{l}\vdash P \wedge b \Rightarrow \mathbf{wp}(c_1, Q)\\ \vdash P \wedge \neg b \Rightarrow \mathbf{wp}(c_2, Q)\end{array}}{\vdash P \Rightarrow \mathbf{wp}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, Q)} \quad \text{(tl3)}$$

Strengthening: $$\frac{\vdash R \Rightarrow Q \quad \vdash P \Rightarrow \mathbf{wp}(c, R)}{\vdash P \Rightarrow \mathbf{wp}(c, Q)} \quad \text{(tl4)}$$

Weakening: $$\frac{\vdash P \Rightarrow R \quad \vdash R \Rightarrow \mathbf{wp}(c, Q)}{\vdash P \Rightarrow \mathbf{wp}(c, Q)} \quad \text{(tl5)}$$

where $c, c_1, c_2 \in \mathcal{C}_0$, $l \in \mathcal{E}_l$, $b \in \mathcal{E}_b$, $al \in \mathit{Alist}$ and $P, Q, R \in \mathcal{A}$

Figure 3.5: Proof Rules for Commands of $\mathcal{L}$

satisfies $\vdash P \Rightarrow \mathbf{wp}(c, Q)$ and Q is the precondition for $l : c$ then $\vdash Q \Rightarrow pc =_a l$. This must be established using the assignment rule (tl1), since *pc* is a name. If c_1 is $:= (al, l)$ then the proof is of the assertions $\vdash P \Rightarrow Q \triangleleft (pc, l)\cdot al$ (for the strengthening rule tl4) and $\vdash Q \Rightarrow Q \triangleleft (pc, l)\cdot al$ (for the assignment rule tl1).

Derived commands: Reasoning about derived commands is based on their definition. The command **goto** l can always be shown to satisfy the specification $\vdash P \triangleleft (pc, l) \Rightarrow \mathbf{wp}(\mathbf{goto}\ l, P)$. The truth of this follows by definition of **goto**, which reduces the specification to $\vdash P \triangleleft (pc, l) \Rightarrow \mathbf{wp}(:= (\mathit{nil}, l), P)$, and from the assignment rule (tl1). The command **abort** satisfies the specification $\vdash \mathit{false} \Rightarrow \mathbf{wp}(\mathbf{abort}, Q)$, for any postcondition $Q \in \mathcal{A}$. By definition $\mathbf{abort} = (pc, pc := \mathbf{true}, \mathbf{false}, \mathit{undef}(\mathcal{E}_l))$. Since the assignment list is never correct in any state, the command **abort** can never terminate.

Instructions: Processor instructions are specified in terms of $\mathcal{L}$ commands. The Alpha AXP instruction `addl` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ is modelled as the command $\mathbf{r}_3 := \mathbf{r}_1 +_a \mathbf{r}_2, \mathbf{loc}(pc +_a 1)$. Using the assignment rule (tl1), the instruction can be shown to satisfy the specification: $\vdash P \Rightarrow \mathbf{wp}(\mathbf{r}_3 := \mathbf{r}_1 +_a \mathbf{r}_2, \mathbf{loc}(pc +_a 1), \mathbf{r}_3 =_a \mathbf{r}_1 +_a \mathbf{r}_2 \wedge pc =_a \mathbf{loc}(pc +_a 1))$, for any $P \in \mathcal{A}$. The specification of the Alpha AXP jump instruction `jmp` $\mathbf{r}, v$ can also be derived from the assignment rule. The instruction is modelled by the $\mathcal{L}$ command $\mathbf{r} := pc, \mathbf{loc}(v)$ and specified: $\vdash P \wedge pc =_a l \Rightarrow \mathbf{wp}((\mathbf{r} := pc, \mathbf{loc}(v)), \mathbf{r} =_a l \wedge pc =_a \mathbf{loc}(v))$, for any $P \in \mathcal{P}$. Note that because the value of the program counter changes, the constant label l must be used to fix the value of *pc* which is to be assigned to **r**.

Abstraction

To see the effect on verification of abstracting commands, assume the commands **goto** l and $(l : x, y := e_1, e_2, l_1)$ are executed in sequence. To show that the commands establish $y =_a e_2$ requires a proof of the two assertions:

$$\vdash \mathit{true} \Rightarrow \mathbf{wp}(\mathbf{goto}\, l, pc =_a l)$$
$$\vdash pc =_a l \Rightarrow \mathbf{wp}((l : x, y := e_1, e_2, l_1), y =_a e_2)$$

Abstracting from the commands replaces these two assertions with a single specification. Let $c = (\mathbf{goto}\, l; (l : x, y := e_1, e_2, l_1))$. By definition, this is the command **if** $pc \lhd (pc, l) =_a l$ **then** $x, y := e_1, e_2, l_1$ **else goto** l. Because $pc \lhd (pc, l) =_a l$ is **true** in any state, command c can be simplified to $x, y := e_1, e_2, l_1$. The specification to be proved is therefore: $\vdash \mathit{true} \Rightarrow \mathbf{wp}((x, y := e_1, e_2, l_1), y =_a e_2)$, which is straightforward from the assignment rule (tl1).

For a second example, consider the following sequence of Alpha AXP instructions:

$$\begin{aligned} &l_1 : \texttt{addl}\ \mathbf{r3}, \mathbf{r2}, \mathbf{r3} \\ &l_2 : \texttt{cmpeq}\ \mathbf{r3}, \mathbf{r4}, \mathbf{r0} \\ &l_3 : \texttt{beq}\ \mathbf{r0}, v \end{aligned}$$

The labels l_1, l_2, l_3 are assumed to satisfy $l_2 \equiv \mathbf{loc}(l_1 +_a 1)$ and $l_3 \equiv \mathbf{loc}(l_2 +_a 1)$. The sequence is verified by replacing each instruction with its equivalent command of $\mathcal{L}$. To show that the sequence passes control to label $\mathbf{loc}(v)$, if execution begins in a state satisfying $\mathbf{r3} -_a \mathbf{r2} =_a \mathbf{r4}$, requires a proof of the assertions:

$$\begin{aligned} &\vdash pc =_a l_1 \wedge \mathbf{r3} -_a \mathbf{r2} =_a \mathbf{r4} \\ &\quad \Rightarrow \mathbf{wp}((l_1 : \mathbf{r3} := \mathbf{r3} +_a \mathbf{r2}, \mathbf{loc}(pc +_a 1)), pc =_a l_2 \wedge \mathbf{r3} =_a \mathbf{r4}) \\ &\vdash pc =_a l_2 \wedge \mathbf{r3} =_a \mathbf{r4} \\ &\quad \Rightarrow \mathbf{wp}((l_2 : \mathbf{if}\ \mathbf{r3} =_a \mathbf{r4}\ \mathbf{then}\ \mathbf{r0} := 1, \mathbf{loc}(pc +_a 1) \\ &\qquad\qquad \mathbf{else}\ \mathbf{r0} := 0, \mathbf{loc}(pc +_a 1)), pc =_a l_3 \wedge \neg \mathbf{r0} =_a 1) \\ &\vdash pc =_a l_3 \wedge \mathbf{r0} =_a 1 \\ &\quad \Rightarrow \mathbf{wp}(l_3 : \mathbf{if}\ \mathbf{r0} =_a 1\ \mathbf{then}\ \mathbf{goto}\ \mathbf{loc}(v)\ \mathbf{else}\ \mathbf{goto}\ \mathbf{loc}(pc +_a 1), pc =_a \mathbf{loc}(v)) \end{aligned}$$

The proof of each assertion is straightforward. However, the proof of the sequence can be simplified by abstracting from the $\mathcal{L}$ commands.

Let c_1, c_2, c_3 be the $\mathcal{L}$ commands modelling the instructions labelled l_1, l_2, l_3 respectively. The abstraction c of the sequence is obtained by the composition of c_1 and c_2, followed by composition with c_3: $c = (c_1; c_2); c_3$. After simplification, this is the command:

$$\begin{aligned} l_1 : \mathbf{if}\ &\mathbf{r3} +_a \mathbf{r2} =_a \mathbf{r4}\ \mathbf{then}\ \mathbf{r3}, \mathbf{r0} := \mathbf{r3} +_a \mathbf{r2}, 1, \mathbf{loc}(v) \\ &\qquad \mathbf{else}\ \mathbf{r3}, \mathbf{r0} := \mathbf{r3} +_a \mathbf{r2}, 0, \mathbf{loc}(l_3 +_a 1) \end{aligned}$$

The specification of the sequence can then be verified as a specification of the command:

$$\begin{aligned}
&\vdash pc =_a l_1 \wedge \mathbf{r3} =_a \mathbf{r4}\\
&\quad\Rightarrow \mathbf{wp}(l_1 : \mathbf{if}\ \mathbf{r3} +_a \mathbf{r2} =_a \mathbf{r4}\\
&\qquad\qquad\quad \mathbf{then}\ \mathbf{r3}, \mathbf{r0} := \mathbf{r3} +_a \mathbf{r2}, 1, \mathbf{loc}(v)\\
&\qquad\qquad\quad \mathbf{else}\ \mathbf{r3}, \mathbf{r0} := \mathbf{r3} +_a \mathbf{r2}, 0, \mathbf{loc}(l_3 +_a 1), pc =_a \mathbf{loc}(v))
\end{aligned}$$

The proof of this specification is straightforward from the precondition and the proof rules. The specifications to be established in the proof based on the individual commands were mainly concerned with propagating properties through the sequence of commands. For example, whether *pc* was assigned $\mathbf{loc}(v)$ in command c_3 depended on the result of the test in command c_2. Because many of these properties could be determined from the syntax of commands, constructing and simplifying an abstraction of the sequence leads to a simpler proof. The abstraction describes all the properties established during the execution of the sequence. The final proof is therefore concerned only with the pre- and postcondition of the sequence and not with the intermediate assertions established by commands in the sequence.

3.5 Conclusion

To model and abstract object code in terms of $\mathcal{L}$, the commands of $\mathcal{L}$ must be expressive enough to model any processor instruction and to support the abstraction of commands. To verify object code modelled in terms of $\mathcal{L}$ also requires the ability to define proof rules of a program logic which can be applied to the commands of $\mathcal{L}$. The principal difficulty when considering processor instructions is the undecidability introduced by pointers and computed jumps. A processor instruction carries out data operations and makes possibly conditional assignments to one or more program variables, which may be identified by pointers. These assignments are simultaneous, requiring a method of detecting unexecutable commands in the presence of pointers. To define proof rules for the commands requires a substitution operator which takes into account the aliasing problem. The abstraction of commands must consider both the aliasing problem, when considering lists of assignments, and computed jumps, when determining whether two commands are executed in sequence.

To model the behaviour of processor instructions, the language $\mathcal{L}$ contains a conditional, a labelling and an assignment command. The data operations of a processor are modelled as expressions of $\mathcal{L}$, which are defined in terms of constants and functions. These include the name expressions, which are a more general model of pointers than those commonly used (e.g. Dijkstra, 1976, Manna & Waldinger, 1981). The name expressions of $\mathcal{L}$ provide flexibility when modelling processor memory operations and allow pointers and simple (constant) variables to be treated equally. The assignment command of $\mathcal{L}$ is a simultaneous assignment and is expressive enough to describe any state transformation. This allows any processor instruction to be modelled by a single $\mathcal{L}$ command. The approach used to detect unexecutable assignments is similar to that of Cartwright & Oppen (1981). However, Cartwright & Oppen (1981) restricted the expressions which could be assigned to variables to exclude substitutions; these restrictions are not required for the commands of $\mathcal{L}$. Instructions implement the execution model of a processor, by selecting instructions for execution. The execution model of $\mathcal{L}$ is based on the use of a program counter *pc* to select a command for execution. Each command of $\mathcal{L}$ assigns a value to the program counter and this value can be the result of an expressions. The language $\mathcal{L}$ is therefore a flow-graph language which includes computed jumps.

The abstraction of commands is based on manipulating the syntax of $\mathcal{L}$ commands to construct a new command. For this approach, sequential composition was defined as a function on commands of $\mathcal{L}$. This differs from the usual treatment of sequential composition as a primitive syntactic construct which represents the combination of commands. The definition of sequential composition here extended the rules of Hoare et al. (1987) to the commands of $\mathcal{L}$, a language which includes pointers and computed jumps. To abstract from assignment commands required both substitution and the combination of assignment lists in the presence of name expressions. The approach used separates the syntactic constructs, representing the operations of substitution and combination, from the semantics of these constructs, which carry out the operations in a given state. This overcomes the undecidability of pointers and computed jumps by allowing the result of the operations to be determined as part of a correctness proof. The syntactic constructs allow the abstraction of assignment commands to be described (and constructed) from the text of the commands. The abstraction of computed jumps uses the conditional command and the program counter *pc* to guard the execution of commands. This ensures that the flow of control from one command to another is accurately described in the abstraction of the two commands. Consequently, sequential composition can be applied to any two commands: the result will always be an abstraction of the commands.

To specify and reason about commands of $\mathcal{L}$, a *wp* predicate transformer was defined to construct an assertion on a state. Proof rules are defined using the specifications formed by this *wp* function, following the approach of Dijkstra (1976). The majority of the proof rules of Figure (3.5) are standard rules for commands (e.g. Gries, 1981). However, the proof rule for the assignment command of $\mathcal{L}$ required the use of the substitution operator of $\mathcal{L}$, which can be applied in the presence of name expressions. Reasoning about substitution expressions based on the syntax of expressions is possible using the rules of Figure (3.3). The language $\mathcal{L}$ differs from the flow-graph languages usually considered in verification (see e.g. Clint & Hoare, 1972, Loeckx & Sieber, 1987 or Francez, 1992) in that the program counter *pc* is a program variable. This allows its use when specifying $\mathcal{L}$ commands and simplifies the proof rules (for the labelling and assignment commands) which describe the execution model of $\mathcal{L}$.

The main contribution of this chapter is the development of the commands of $\mathcal{L}$ which can be used to model processor instructions and the definition of a method of abstracting arbitrary $\mathcal{L}$ commands. The language $\mathcal{L}$ is independent of any processor: no distinction is made between instructions of the same processor and instructions of different processors. This is possible because processor instructions have a similar semantics. Modelling instructions in terms of $\mathcal{L}$ provides the ability to simplify verification by abstracting from the commands of a program. The abstraction of commands is constructed from the text of the commands, an approach which allows for efficient mechanisation. This reduces the manual work needed to verify a program by providing a means for the use of automated tools to abstract and simplify sequences of commands in the program.

Chapter 4

Programs

An object code program is verified by showing that states which are produced during its execution have the properties required by the program specification. The execution of an object code program is based on the execution of the program's instructions. Both the selection of instructions for execution and the states produced during execution are determined by the actions of the program instructions; the object code program only determines the instructions which are available for execution. Object code is verified in terms of its model as a program of the language $\mathcal{L}$, to allow the methods of verification and abstraction to be independent of the processor language. The translation from a processor language to the language $\mathcal{L}$ is straightforward: a program of $\mathcal{L}$ which models object code is simply the set of $\mathcal{L}$ commands modelling the instructions.

A program of $\mathcal{L}$ is verified by reasoning about the individual program commands and the verification is simplified by abstracting from the program. The main difficulty with programs of $\mathcal{L}$ is the abstraction of programs, which will be constructed by applying program transformations. To use these transformations in verification they must be shown to preserve the correctness of a program. The correctness of a transformation is established by reasoning about the changes made by the transformation to the program behaviour. The techniques used in verification to reason about program transformation generally assume that the programming language imposes a syntactic structure on programs which can be used to define transformations. These languages also allow comparisons to be made between the behaviour of programs without the need to consider the behaviour of individual program commands, simplifying reasoning about transformations. Programs of $\mathcal{L}$ do not have a syntactic structure with which to define transformations and the behaviour of $\mathcal{L}$ programs must be compared by considering the effect of each program command individually. To define and reason about transformations on programs of $\mathcal{L}$ therefore requires an extension of the methods used in program verification.

The abstraction of $\mathcal{L}$ programs will be based on the sequential composition operator of Chapter 3. The simplest method of abstraction is to apply sequential composition to manually chosen program commands. Any program can be abstracted in this way but the size of object code programs means that this approach is not enough to reduce the work needed to verify a program. A second approach is to use the flow of control through a program to mechanically find the sequences of commands which are to be abstracted by sequential composition. This approach can be described in terms of program transformations which can be efficiently automated, the techniques required are common in code optimisation. However, it requires a method for showing that the program transformations are correct. A framework will be described in which the correctness of a program transformation can be established. The framework is based on a method for using the flow-graph of a program to impose a syntactic structure on the program. This structure can then be used to define and reason about program transformations.

To verify $\mathcal{L}$ programs, a program logic suitable for reasoning about the liveness properties of a program will be described. This is based on the assertion language $\mathcal{A}$ of Chapter 3, extended with a specification operator for programs. Proof rules for this specification operator allow the programs of $\mathcal{L}$ to be verified using the method of intermittent assertions (Manna, 1974; Burstall, 1974). The properties of program commands can be specified and verified using the **wp** function and proof rules of Chapter 3. The proof rules for programs will also support the abstraction of a program to simplify its verification and an approach to using program abstraction when verifying programs will be described.

The chapter begins with the definition in **Section 4.1** of the syntax and semantics of $\mathcal{L}$ programs. This includes a refinement relation between programs of $\mathcal{L}$ and a method for program abstraction based on individually chosen commands. **Section 4.2** describes the framework used to define and reason about the transformations for program abstraction. Transformations are defined on a *region* of a program: a group of commands with a structure defined by their flow-graph. The transformations on regions, used to abstract programs, are defined and their properties described in **Section 4.3**. The specification and verification of $\mathcal{L}$ programs is described in **Section 4.4**. The method used to verify a program will be described and a small program verified as an example.

4.1 Programs of $\mathcal{L}$

An object code program has a finite number of instructions, each of which is stored in a memory location identified by a unique address (the label of the instruction). Instructions can be added to an object program by storing an instruction at an address which does not identify any other instruction. These properties are preserved by programs of $\mathcal{L}$, which are finite sets of uniquely labelled $\mathcal{L}$ commands. This is enough to model a program of a processor language since a command of $\mathcal{L}$ can model an instruction. Program commands are indexed by the labels: if a program p contains command $l : c$ then the label l uniquely identifies that command and is enough to obtain command $l : c$ from p.

Definition 4.1 *Programs*

A finite set p of labelled commands is a program of $\mathcal{L}$ iff *program?*(p) is *true*.

$$\begin{array}{l}\mathit{program?} : \mathit{FiniteSet}(\mathcal{C}) \to \mathit{boolean}\\ \mathit{program?}(a) \stackrel{\text{def}}{=} \forall(c, c_1 \in a) : \mathit{label}(c) = \mathit{label}(c_1) \Rightarrow c = c_1\end{array}$$

The set $\mathcal{P}$ contains all programs of $\mathcal{L}$.

$$\mathcal{P} \stackrel{\text{def}}{=} \{p : \mathit{FiniteSet}(\mathcal{C}) \mid \mathit{program?}(p)\}$$

If there is a command c in program p with label l then c is the command of p at l.

$$\begin{array}{l} \mathit{at} : (\mathcal{P} \times \mathit{Labels}) \to \mathcal{C} \\ \mathit{at}(p, l) \stackrel{\text{def}}{=} \epsilon\{c : \mathcal{C} \mid c \in p \wedge l = \mathit{label}(c)\} \end{array}$$

□

Function *at* obtains a command identified by a label from a program: the command labelled l in a program p is $\mathit{at}(p, l)$. Note that every subset of a program is also program; the subset of a program p will also be called a *sub-program* of p. In particular, the empty set is a program and a sub-program of every program.

A program of $\mathcal{L}$ does not have a structure nor does it identify an initial command: execution can begin with any command and a program can contain commands which cannot be or are never executed. The command which begins the program must be identified by the precondition of the program, which must state the initial value of the program counter. Moreover, the order in which commands of the program are executed is independent of any ordering of the labels.

Example 4.1 Assume $l_1, l_2, l_3 \in \mathit{Labels}$ such that $l_1 + 1 = l_2$ and $l_2 + 1 = l_3$. The labels of the commands in program $\{l_1 : \textbf{goto}\ l_3, l_2 : \textbf{goto}\ l_1, l_3 : \textbf{goto}\ l_2\}$ are ordered $l_1 < l_2 < l_3$ while the commands of the program are executed in the sequence l_1, l_3, l_2. The set $\{l_1 : \textbf{goto}\ l_1, l_2 : \textbf{goto}\ l_2\}$ is also a program since $l_1 \neq l_2$ but only one of its commands will be executed. □

A program can be extended with a command c to form a program $p \cup \{c\}$ provided that no command in p shares a label with c. A program can also be constructed by the combination of two programs p_1 and p_2. The program p' constructed by combining p_1 with p_2 contains all commands of p_2 and the commands of p_1 which do not share a label with a command of p_2. When there are commands $c_1 \in p_1$ and $c_2 \in p_2$ which share a label, $\mathit{label}(c_1) = \mathit{label}(c_2)$, only command c_2 occurs in program p'.

Definition 4.2 *Program construction*

If there is no command in a program p having the label $\mathit{label}(c)$ then $p + c$ is the *addition* of command c to p, otherwise it is p.

$$\begin{array}{l} \_ + \_ : (\mathcal{P} \times \mathcal{C}) \to \mathcal{P} \\ p + c \stackrel{\text{def}}{=} \begin{cases} p \cup \{c\} & \text{if } \forall (c_1 \in p) : \mathit{label}(c_1) \neq \mathit{label}(c) \\ p & \text{otherwise} \end{cases} \end{array}$$

The *combination* of programs $p_1, p_2 \in \mathcal{P}$ is the union of p_2 with the subset of p_1 containing commands whose labels are distinct from those of p_2.

$$\begin{array}{l} \_ \uplus \_ : (\mathcal{P} \times \mathcal{P}) \to \mathcal{P} \\ p_1 \uplus p_2 \stackrel{\text{def}}{=} p_2 \cup \{c_1 \in p_1 \mid \forall c_2 \in p_2 : \mathit{label}(c_1) \neq \mathit{label}(c_2)\} \end{array}$$

□

Since the empty set is a program, a program can be constructed by the addition of commands to the empty set. The definition of addition ensures that a program p is only extended with a command $(l : c)$ if no command in p is labelled with l. The combination of programs p_1 and p_2 is equivalent to the addition of the individual commands of p_1 to p_2. Typically, the combination operator will be applied when p_2 is a program derived from a subset of p_1.

Example 4.2 Let l_1, l_2, l_3 be distinct labels and assume program $p = \{l_1 : c_1, l_2 : c_2, l_3 : c_3\}$.

Program p can be constructed from the empty set by the addition, in any order, of its commands: $p = \{\} + l_2 : c_2 + l_3 : c_3 + l_1 : c_1$. Furthermore, program p can also be constructed from any subset of p by the addition of all of its commands: $p = \{l_1 : c_1, l_3 : c_3\} + l_2 : c_2 + l_3 : c_3 + l_1 : c_1$.

Let p' be the program derived from p by the sequential composition of $l_1 : c_1$ and $l_3 : c_3$ and the removal of $l_3; c_3$: $p' = \{(l_1 : c_1; l_3 : c_3), l_2 : c_2\}$. Note that the label of $(l_1 : c_1; l_3 : c_3)$ is l_1.

The combination of p and p', $p \uplus p'$, is the program which contains the commands $at(p', l_1)$ and $at(p, l_3)$: $p \uplus p' = \{(l_1 : c_1; l_3 : c_3), l_2 : c_2, l_3 : c_3\}$. □

Induction on Programs

The set of programs, $\mathcal{P}$, is closed under addition: every program can be formed from the empty set and the addition of a finite number of commands. This defines an induction scheme for programs. A second, for strong induction, is derived from the Strong Finite Induction scheme (see page 8) on the set of commands which form a program.

Theorem 4.1 *Induction schemes*

For any property Φ, with type $\mathcal{P} \to boolean$,

1. *Induction:*

$$\frac{\Phi(\{\}) \quad \forall (p : \mathcal{P}) : \Phi(p) \Rightarrow \forall (c : \mathcal{C}) : \Phi(p + c)}{\forall (p : \mathcal{P}) : \Phi(p)}$$

2. *Strong induction:*

$$\frac{\forall (p : \mathcal{P}) : (\forall (p' : \mathcal{P}) : p' \subset p \Rightarrow \Phi(p')) \Rightarrow \Phi(p)}{\forall (p : \mathcal{P}) : \Phi(p)}$$

The induction schemes can be used to reason about the syntactic properties of a program. In general, these are the properties of the set of commands making up the program.

4.1.1 Semantics of Programs

A program is executed by the repeated selection and execution of its commands. Execution begins in an initial state s in which the value of the program counter identifies a command. This command is executed and, if it terminates, produces a new state t. The value of the program counter in state t identifies the next command to be executed and the selection and execution of commands is then repeated. A *behaviour* σ is an infinite sequence of states and σ is a behaviour of a program p iff σ can be produced as the result of executing p beginning in the initial state $\sigma(0)$.

Definition 4.3 *Semantics of programs*

A behaviour is an infinite sequence of states.

$$\mathit{Behaviour} \stackrel{\text{def}}{=} \mathit{seq}(\mathit{State})$$

Interpretation function $\mathcal{I}_p$ applied to program p and behaviour σ is *true* iff σ is a behaviour of p.

$$\begin{array}{l}\mathcal{I}_p : \mathcal{P} \to \mathit{Behaviour} \to \mathit{boolean} \\ \mathcal{I}_p(p)(\sigma) \stackrel{\text{def}}{=} \forall(n : \mathbb{N}) : \exists(c \in p) : \mathcal{I}_c(c)(\sigma(n), \sigma(n+1))\end{array}$$

□

Sequences of states are commonly used to model the behaviour of sequential and parallel programs (Loeckx and Sieber, 1987; Cousot, 1990; Manna and Pnueli, 1991). They are also used to define the semantics of temporal logics, used for program specification and verification (Manna and Pnueli, 1981; Lamport, 1994). Both liveness and safety properties can be defined in terms of program behaviours (Manna & Pnueli, 1991). *Liveness properties* of the program p require that a state in the program behaviour satisfies some assertion $P \in \mathcal{A}$: if $\mathcal{I}_p(p)(\sigma)$ then $\exists(i : \mathbb{N}) : P(\sigma(i))$. *Safety properties* of the program require that every state in the behaviour of program p satisfies some assertion $P \in \mathcal{A}$: if $\mathcal{I}_p(p)(\sigma)$ then $\forall(i : \mathbb{N}) : P(\sigma(i))$.

Every state in a program behaviour is the result of executing commands of the program. Because the commands of $\mathcal{L}$ are deterministic, a behaviour σ of program p is defined by the initial state $\sigma(0)$ and the commands of p. For $i > 0$, state $\sigma(i)$ is said to be *produced* by program p from the initial state $\sigma(0)$. Since a behaviour σ of a program is an infinite sequence, every sub-sequence σ_i, $i \geq 0$, is also a behaviour of the program.

Corollary 4.1 *For any program p, behaviour σ and $i \in \mathbb{N}$, if σ is a behaviour of p then so is every sub-sequence of σ: $\mathcal{I}_p(p)(\sigma) \Rightarrow \mathcal{I}_p(p)(\sigma_n)$.*

Proof. From definition of $\mathcal{I}_p(p)(\sigma)$, for every $j \in \mathbb{N}$ there is a command $c \in p$ such that $\mathcal{I}_c(c)(\sigma(j), \sigma(j+1))$. From the definition of the suffix of a behaviour, $\sigma_n(i) = \sigma(n+i)$ and the proof of $\mathcal{I}_p(p)(\sigma_n)$ follows from $\mathcal{I}_p(p)(\sigma)$ with $j = n + i$. □

A program p terminates when a state is produced in which no command of p can begin execution. Because a behaviour is an infinite sequence of states, there is no behaviour σ and terminating program p such that $\mathcal{I}_p(p)(\sigma)$. However, a terminating program can be transformed to a non-terminating program by the addition of commands to form an infinite loop. In general, no distinction will be made between (non-terminating) programs for which there is a behaviour and (terminating) programs for which there is no behaviour. Whether a program terminates is not decidable from the syntax of the program commands and when deriving an abstraction of a program only the syntax will be considered.

Example 4.3 Let l_1, l_2, l_3 be distinct labels and b a boolean expression, $b \in \mathcal{E}_b$. If program p has the commands l_1 : **goto** l_2 and l_2 : **if** b **then goto** l_3 **else goto** l_1. then p terminates if $b =$ **true** since there is no command in p with label l_3. Program p can be extended to a non-terminating program p' by the addition of l_3 : **goto** l_3, $p' = p + l_3$: **goto** l_3.

Let $b =$ **true** and σ be a sequence of states in which $\mathcal{I}_l(pc)(\sigma(0)) = l_1$, $\mathcal{I}_l(pc)(\sigma(1)) = l_2$ and $\mathcal{I}_l(pc)(\sigma(2)) = l_3$. If for every $n > 2$, $\sigma(n) = \sigma(2)$ then σ is a behaviour of program p'. The first command of the program is labelled l_1 and relates $\sigma(0)$ to $\sigma(1)$ and the second command is labelled l_2 relating states $\sigma(1)$ and $\sigma(2)$. After the second command terminates, the only command that is selected is l_3 which repeatedly selects itself. □

Properties of Programs

Programs of the language $\mathcal{L}$ are sequential: only one command of a program can be selected in any state. If σ is a behaviour of program p and l the value of the program counter in state $\sigma(0)$, the command at l in p is executed in $\sigma(0)$, $\mathcal{I}_c(at(p, \mathcal{I}_e(pc)(\sigma(0))))(\sigma(0), \sigma(1))$. A program p terminates in a state s if no command of p is enabled in s; state s is *final* for p. If there is a command c of program p which is enabled in s and c halts then program p also halts in s. No other command of p can be executed in s, since each command is uniquely labelled, and no successor to command c can be selected.

Definition 4.4 *Final states and program failure*

A state s is final for a set a of commands iff no command $c \in a$ is enabled in s.

$$\begin{aligned}&\textit{final?} : \textit{Set}(\mathcal{C}) \to \textit{State} \to \textit{boolean}\\&\textit{final?}(a)(s) \stackrel{\text{def}}{=} \forall(c : \mathcal{C}) : c \in a \Rightarrow \neg\textit{enabled}(c)(s)\end{aligned}$$

Program p halts in state s iff is there is a command $c \in p$ which halts in s.

$$\begin{aligned}&\textit{halt?} : \mathcal{P} \to \textit{State} \to \textit{boolean}\\&\textit{halt?}(p)(s) \stackrel{\text{def}}{=} \exists(c \in p) : \textit{halt?}(c)(s)\end{aligned}$$

□

Programs $p, p' \in \mathcal{P}$ are equivalent, $\mathcal{I}_p(p)(\sigma) = \mathcal{I}_p(p')(\sigma)$ for $\sigma \in \textit{Behaviour}$, iff programs p and p' produce the same states in the same order. For every command $c \in p$ which can be executed there is an equivalent command $c' \in p'$. Conversely, transforming a program p by replacing commands of p with equivalent commands will result in a program p' which is equivalent to p.

Lemma 4.1 *For any programs $p, p' \in \mathcal{P}$ and behaviour σ,*

$$\frac{\forall(s,t:\mathit{State}):\exists(c \in p):\mathcal{I}_c(c)(s,t) \Leftrightarrow \exists(c' \in p'):\mathcal{I}_c(c')(s,t)}{\mathcal{I}_p(p)(\sigma) = \mathcal{I}_p(p')(\sigma)}$$

A useful transformation using the property of Lemma (4.1) is the systematic replacement of the program counter *pc* with the label of the command in which it appears.

Example 4.4 Let l_1, l_2, l_3 be distinct labels and c_1 be some command. The program $p = \{l_1 : c_1, l_2 : \textbf{abort}\}$ halts for all states in which $l_2 : \textbf{abort}$ is enabled and also for all states s, t such that $\mathcal{I}_c(l_1 : c_1)(s,t)$ and $\mathit{enabled}(l_2 : \textbf{abort})(t)$.

The state s in which *pc* has the value l_3 is final for program p: no command in p is labelled l_3.

The program $\{l : \textbf{goto}\, \mathit{pc}\}$ is equivalent to the program $\{l : \textbf{goto}\, l\}$ since the command l : **goto** *pc* is equivalent to the command $l : \textbf{goto}\, l$.

The program $\{l : \textbf{if true then goto}\, \mathit{pc}\, \textbf{else abort}\}$ is equivalent to the program $\{l : \textbf{goto}\, l\}$ since the command l : **if true then goto** *pc* **else abort** is equivalent to l : **goto** *pc*. □

4.1.2 Transition Relation: *leads-to*

To verify the liveness properties of a program it is only necessary to show that eventually the program produces a state t satisfying a postcondition from a state s satisfying a precondition. A program relates the state in which it begins execution to the states which are produced during the execution. If execution of program p begins in state s and produces state t then s *leads to* t through p and the program defines a *transition relation* between the two states. Transition relations are common in program verification and analysis (e.g. Manna, 1974; Cousot, 1981; Gordon, 1994b). The simplest transition relation, called *leads-to*, can be used to reason about the liveness properties of a program by relating the initial state s to a state t produced by the program. The *leads-to* relation between states is the transitive closure, restricted to a program p, of the interpretation function $\mathcal{I}_c$ on the commands of p.

Definition 4.5 *Leads to*

The *leads-to* relation from state s to state t through the set a of commands is written $s \overset{a}{\leadsto} t$ and is inductively defined:

$$\frac{c \in a \quad \mathcal{I}_c(c)(s,t)}{s \overset{a}{\leadsto} t} \qquad \frac{s \overset{a}{\leadsto} u \quad u \overset{a}{\leadsto} t}{s \overset{a}{\leadsto} t}$$

For any command $c \in \mathcal{C}_0$ and states $s, t \in \mathit{State}$, if $\mathcal{I}_c(c)(s,t)$ then s *leads to* t through command c. The interpretation of a command $\mathcal{I}_c(c)(s,t)$ will also be written $s \overset{c}{\leadsto} t$. □

The *leads-to* relation extends the interpretation function of commands to consider the cumulative effect of commands of a program p. If p beginning in a state s leads to a state t then the commands of p establish a relationship between the two states. The *leads-to* relation also allows the liveness properties of a subset of a program to be related to the program. If program p produces a state t from t then any program of which p is a subset will also produce t from s.

Lemma 4.2 *For sets $a, b \in Set(\mathcal{C})$ of commands and states $s, t \in State$,*

$$\frac{a \subseteq b \quad s \overset{a}{\leadsto} t}{s \overset{b}{\leadsto} t}$$

Proof. Straightforward, by induction and from the definition. □

The *leads-to* relation determines the states that are produced during the execution of a program, without requiring the intermediate states produced by the program to be considered. If program p beginning in state s *leads to* state t then state t will eventually appear in a behaviour of the program.

Theorem 4.2 *For program p, states s and t, behaviour σ and $n, m \in \mathbb{N}$,*

$$\frac{\mathcal{I}_p(p)(\sigma) \quad s = \sigma(0) \quad s \overset{p}{\leadsto} t}{\exists m : m > 0 \wedge t = \sigma(m)}$$

As a consequence of Theorem (4.2), to verify a liveness property of a program it is only necessary to consider the states related by *leads-to*. The property, P, is established by showing that the program beginning in state s eventually produces a state t satisfying P. Any behaviour σ of the program such that there is a $\sigma(i) = s$, $i \geq 0$, will also have a state $\sigma(j) = t$, $j > i$, satisfying property P.

Example 4.5 Let p be a program with commands $c_1, c_2, c_3 \in \mathcal{C}$ and let s_1, s_2, s_3, s_4 be states such that $s_1 \overset{c_1}{\leadsto} s_2 \overset{c_2}{\leadsto} s_3 \overset{c_3}{\leadsto} s_4 \overset{c_1}{\leadsto} s_2$.

State s_1 leads to each of the states s_2, s_3 and s_4 through program p. Command c_1 is executed once to produce s_2 from s_1. Since state s_2 leads to s_2, commands c_2, c_3 and c_4 can be executed any number of times. Any behaviour σ of p such that $\sigma(i) = s_1$, $i \geq 0$, will have states $\sigma(i+1) = s_2, \sigma(i+2) = s_3, \sigma(i+3) = s_4$ and $\sigma(i+4) = s_2$. □

4.1.3 Refinement of Programs

A program p is refined by program p', written $p \sqsubseteq p'$, if every state produced by p is also produced by p'. Refinement between programs allows program p to be substituted for p' in a proof of correctness. Program p is an abstraction of p' and, to show that p' produces a state t satisfying an assertion, it is enough to show that t can be produced by program p. The replacement of p' with p cannot lead to an incorrect proof: program p' can produce more states than the abstraction p but attempting to show that p produces a state it does not can only lead to failure of the proof.

A refinement relation is needed to show that a program p is abstracted by a second program p', $p' \sqsubseteq p$. (Here, the abstraction p' would be obtained as the result of transforming p in some way.) The refinement relation $\sqsubseteq$ between programs of $\mathcal{L}$ is defined using the relation *leads-to* to compare the states produced by programs.

Definition 4.6 *Refinement*

Program p_1 is refined by p_2, written $p_1 \sqsubseteq p_2$, if whenever p_1 beginning in state s leads to t then p_2 beginning in state s also leads to state t.

$$\begin{array}{l} \_ \sqsubseteq \_ : (\mathcal{P} \times \mathcal{P}) \to \mathit{boolean} \\ p_1 \sqsubseteq p_2 \stackrel{\text{def}}{=} \forall (s, t : \mathit{State}) : s \stackrel{p_1}{\leadsto} t \Rightarrow s \stackrel{p_2}{\leadsto} t \end{array}$$

□

This refinement relation is weaker than those for structured languages (Back & von Wright, 1989), where refinement is based on the semantics of programs. Assume $\mathcal{I}$ is the interpretation of a structured program such that $\mathcal{I}(p)(s, t)$ is *true* iff structured program p beginning in state s terminates in state t. Let $\sqsubseteq_{SP}$ be the refinement relation between structured programs satisfying, for structured programs, p and p', $p \sqsubseteq_{SP} p' \Leftrightarrow \mathcal{I}(p)(s, t) \Rightarrow \mathcal{I}(p')(s, t)$. Because this compares the states in which the programs begin and end, this refinement relation does not depend on the number of commands in the programs p and p'. This relation can also be used to establish the equivalence of structured programs: if $p \sqsubseteq_{SP} p'$ and $p' \sqsubseteq_{SP} p$ then $\mathcal{I}(p)(s, t) = \mathcal{I}(p')(s, t)$ for all $s, t \in \mathit{State}$. This is not true for the refinement relation $\sqsubseteq$ of $\mathcal{L}$ (Definition 4.6). Assume $\mathcal{L}$ programs $p_1, p_2 \in \mathcal{P}$ satisfy $p_1 \sqsubseteq p_2$ and $p_2 \sqsubseteq p_1$. It is not the case that $\mathcal{I}_p(p_1)(\sigma) = \mathcal{I}_p(p_2)(\sigma)$, for behaviour σ. Programs p_1 and p_2 can produce a different number of intermediate states or produce states in a different order, depending the commands used to produce the states.

A refinement relation $\sqsubseteq_B$ based on the behaviour of a program of $\mathcal{L}$ would satisfy $p_1 \sqsubseteq_B p_2 \Leftrightarrow \forall \sigma : \mathcal{I}_p(p_1)(\sigma) \Rightarrow \mathcal{I}_p(p_2)(\sigma)$. This would have the property $p_1 \sqsubseteq_B p_2 \wedge p_2 \sqsubseteq_B p_1 \Rightarrow \mathcal{I}_p(p_1)(\sigma) = \mathcal{I}_p(p_2)(\sigma)$. However, this form of refinement is too restrictive since it requires p_2 to produce the same states as p_1 (to ensure that their behaviours are equal). For each command $c_1 \in p_1$ this would require a command $c_2 \in p_2$ such that $\mathcal{I}_c(c_1)(s, t) \Rightarrow \mathcal{I}_c(c_2)(s, t)$. This forbids the replacement of a sequence of commands in p_2 with a single command in p_1, limiting the abstraction of programs. Refinement can be based on a program behaviour, by showing that a behaviour σ of a program p_1 can be transformed (by some function f) to be a behaviour of the refinement p_2 (Abadi & Lamport, 1991). However, this approach complicates reasoning about the abstraction of a program since the changes made to the program behaviour must be considered (see Lamport, 1994, for a logic using this approach to refinement).

Refinement between programs is intended to compare the states produced by a program, rather than the commands needed to produce those states. For example, refinement is often used to justify compilation rules which replace a single command with a sequence of simpler commands (Hoare et al., 1993; Bowen and He Jifeng, 1994). Definition (4.6) is based on this approach to refinement between programs. The use of the *leads-to* relation allows the commands in the programs to be ignored in favour of the states produced by the program. This leads to a refinement relation which is sufficient for reasoning about the liveness properties of programs. If programs p_1 is refined by p_2 then any state produced by p_1 must also be produced by p_2 and will appear in the behaviours of both p_1 and p_2 (Theorem 4.2). This simplifies the replacement of a program with its abstraction during a verification proof.

Example 4.6 Assume program p is specified by assertions $P, Q \in \mathcal{A}$ such that P specifies the state in which program execution begins and Q is a postcondition to be established by the program. Program p eventually establishes Q from P if for every state s such that $P(s)$, there is a state t such that $s \overset{p}{\leadsto} t$ and $Q(t)$.

Assume $p' \sqsubseteq p$ and $\exists t : s \overset{p'}{\leadsto} t \wedge Q(t)$. Since $p' \sqsubseteq p$, $\exists t : s \overset{p}{\leadsto} t \wedge Q(t)$ is *true*. Also assume that σ is a behaviour of p, $\mathcal{I}_p(p)(\sigma)$ such that $\sigma(0) = s$. From Theorem (4.2), there is a $j > 0$ such that $\sigma(j) = t$ and $Q(\sigma(j))$. □

Properties of Refinement

Refinement has a number of basic properties, for verification the most useful is transitivity. If p_1 is an abstraction of p_2 and p_2 is an abstraction of program p then verifying p_1 will also verify p.

Theorem 4.3 *For programs $p_1, p_2, p_3 \in \mathcal{P}$,*

$$\frac{p_1 \sqsubseteq p_2 \quad p_2 \sqsubseteq p_3}{p_1 \sqsubseteq p_3}$$

Proof. Straightforward, from definition of refinement and transitivity of *leads-to*. □

The refinement relation allows the replacement of a program p in a proof of correctness with an abstraction of p. However, the abstraction of p must be chosen with care since any program is a refinement of the program which always fails.

Theorem 4.4 *For any programs $p_1, p_2 \in \mathcal{P}$ and states s, t:*

$$\frac{\forall s, t : \neg(s \overset{p_1}{\leadsto} t)}{p_1 \sqsubseteq p_2}$$

Proof. Straightforward, by definition of $\sqsubseteq$. □

A trivial method of program abstraction is to replace a program p with the empty set since, by Theorem (4.4), the ordering $\{\} \sqsubseteq p$ can always be established. For verification, such abstractions are not useful, a program which always fails cannot be used to verify the liveness property of a program with commands. Instead, the abstraction of a program must be constructed so that the abstraction can be shown to have the properties of p which are of interest.

4.1.4 Simple Program Abstraction

A simple method for abstracting from a program p is to apply sequential composition to commands of p. The correctness of this method is based on three properties of sequential composition and refinement. The first, that combining the sequential composition of program commands with the program forms an abstraction. The second, that the abstraction of any subset of a program p is an abstraction of p. The last, that combining a program p with its abstraction forms an abstraction of p.

Theorem 4.5 *Abstraction of programs*

For program $p, p_1, p_2 \in \mathcal{P}$ and commands $c_1, c_2 \in \mathcal{C}$

1. *Composition of commands:* $$\frac{c_1 \in p \quad c_2 \in p}{(p \uplus \{c_1; c_2\}) \sqsubseteq p}$$
2. *Refinement and sub-programs:* $$\frac{p_1 \sqsubseteq p_2 \quad p_2 \subseteq p}{p_1 \sqsubseteq p}$$
3. *Programs and abstraction:* $$\frac{p_1 \sqsubseteq p_2}{(p_1 \uplus p_2) \sqsubseteq p_2}$$

The properties of Theorem (4.5) allow an abstraction to be constructed for any program p of $\mathcal{L}$ by manipulating commands or subsets of p. The method is to choose two commands c_1, c_2 of p and then apply sequential composition, $c_1; c_2$. This forms the singleton set $\{(c_1; c_2)\}$, which is an abstraction of $\{c_1, c_2\}$ (as a consequence of Theorem 3.6). The singleton set is combined with program p to form the abstraction $p \uplus \{(c_1; c_2)\} \sqsubseteq p$. Note that $(p \uplus \{c_1; c_2\})$ is equivalent to $(p - \{c\}) \cup \{c_1; c_2\}$, replacing command c_1 with $c_1; c_2$. Because c_2 may be the target of a computed jump, it is neither replaced nor removed (the target of a computed jump is undecidable).

This method can be repeated any number of times; because refinement is transitive, the result will always be an abstraction of the original program. Because sequential composition always abstracts from its arguments (Theorem 3.5 and Theorem 3.6), there is no restriction on the choice of commands c_1 and c_2 The choice would normally be made to simplify the verification and by following the order in which the commands are executed.

4.2 Program Transformation

Program abstraction using the method of Section (4.1.4) requires the manual choice of the program commands to be abstracted. Because of the size of an object code program, manually choosing the commands is too time-consuming to be practical. A more efficient approach is to automate the program abstraction by implementing a program transformation T which abstracts from any program p to which it is applied: $T(p) \sqsubseteq p$. The transformation must ensure that verifying $T(p)$ is simpler, at worst no more difficult, than verifying p.

The sequential composition operator can be used by a program transformation to abstract from program commands. The main difficulty is therefore the selection of the program commands to be abstracted. Commands are selected for abstraction in the order in which they may be executed. This is the flow of control through a program, which can be determined using the techniques of code optimisation (Hecht, 1977; Aho et al., 1986). These construct a flow-graph of the program from the syntax of the program commands. In a flow-graph, there is an edge from command c_1 to command c_2 iff control can pass from c_1 to c_2, this will be written $c_1 \longmapsto c_2$. The choice of commands c_1, c_2 of a program will require, at least, that c_1 passes control to c_2: $c_1 \longmapsto c_2$; the proof methods for verification impose additional constraints.

Consider a simple transformation T_S, based on the properties of Theorem (4.5), which considers only the flow of control from one command to another. Assume p is a program to be transformed by T_S. The abstraction $T_S(p)$ is obtained by first choosing any two commands $c_1, c_2 \in p$ such that $c_1 \longmapsto c_2$ and then forming the abstraction of p as the program $T_S(p) = p \uplus \{c_1; c_2\}$. The result of this transformation is not necessarily useful for verification since it can hide properties needed to verify the program. In particular, it fails to preserve the loops in a program (which must be verified by induction on the values of variables; Floyd, 1967). To apply the transformation to a program p, each cut-point in a loop must first be identified manually and removed from the sub-program of p which is transformed.

For example, assume $c_1 = (l_1 : x := 1, l_2)$, $c_2 = (l_2, x := 0, l_2)$ and let program p be $\{c_1, c_2\}$. Assume that program execution begins with c_1 and that the postcondition to be established by the program requires control to pass to command c_2 with $x >_a 0$. The transformation results in program $T_S(p) = \{(c_1; c_2), c_2\}$. By composing the commands c_1 and c_2, the first execution of the command c_2 is hidden by the execution of $c_1; c_2$. When control passes to c_2, (the only command labelled l_2), the value of x will be 0 and will remain 0. The postcondition of the program p cannot be established from the abstraction $T_S(p)$. This can be avoided by manually identifying c_2 as a cut-point in a loop and removing it from the program. However, loops in object code can be made up of a large number of instructions and it is not practical to manually identify all the cut-points of loops which must be preserved.

To construct an abstraction which is useful for verification, a transformation must identify and preserve the cut-points of a program which are part of loops in the program. The cut-points of a program are determined by the loops in the program (Floyd, 1967). To find loops in a program requires a more sophisticated analysis of the program's flow-graph than is possible by considering only two commands since a loop can be made up of any number of commands.

Program Optimisation

Techniques for code optimisation can be used as a basis for program abstraction. Code optimisation is based on analysing a program's flow-graph to determine the changes which can be made to the program. The flow-graph analysis can be used to identify the loops in a program which are needed by the proof methods for program correctness. However, optimising transformations can require techniques which cannot be used in program abstraction. For example, a *node copying*

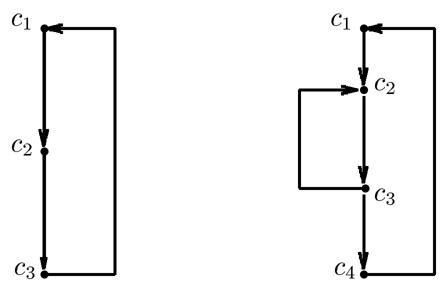

Figure 4.1: Reducible Loops

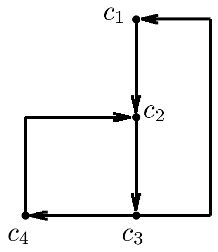

Figure 4.2: An Irreducible Loop

transformation is often used to restructure the flow-graph of a program, making it more suitable for the analysis techniques (Hecht, 1977). This constructs a refinement $T(p)$ of a program p, $p \sqsubseteq T(p)$, by adding one or more commands to p. Since the new commands of $T(p)$ can begin in a state in which p halts (they do not share labels with any command of p) the program $T(p)$ is not an abstraction of p. Such techniques make it difficult to show that a transformation constructs an abstraction of the program, satisfying $T(p) \sqsubseteq p$.

The methods used in code optimisation to analyse a flow-graph can also limit the application of a transformation. These techniques often assume that all loops in a program are *reducible* (Aho et al., 1986): any two loops are either entirely distinct or one occurs entirely within the other (see Figure 4.1). This is *false* for flow-graph programs, such as object code, which can contain *irreducible loops* (also called *interleaving loops*, Loeckx & Sieber, 1987). These consist of two loops neither of which is entirely contained within the other (see Figure 4.2). Irreducible loops can be transformed (by node-copying) to reducible loops (Aho et al., 1986), but this complicates the transformation of a program. Methods of analysing programs with irreducible loops have been proposed (e.g. Sreedhar et al., 1996) although the interpretation of a loop can vary between techniques (Wolfe, 1991). These methods can be used with transformation T_S, to select the program commands to be abstracted. However, transformation T_s abstracts from only two commands in a single application and is less efficient than a transformation which abstracts from a group of commands in one application.

Using the techniques of code optimisation to abstract from a program also makes the correctness of a transformation difficult to prove. Code optimisation techniques are conservative, exploiting relatively simple properties of the program text. The correctness of an optimising transformation is established by showing that the properties of the text of the transformed program are those of the original. The correctness of an abstracting transformation, for verification, must be established from the states produced by the program and its abstraction. The methods used in code optimisation are not suitable for establishing the correctness of an abstracting transformation since the states produced by the program are not considered.

Abstraction by Program Transformation

The method used here to abstract from programs is based on transforming *regions* of a program where a region is a subset of the program which has a structure defined by its flow-graph. The abstraction of a region will follow the approach used by the proof methods for verification (Floyd, 1967; Burstall, 1974): the cut-points of the region will be found and the sequences of commands between each cut-point will be abstracted by sequential composition. The abstraction of a program is obtained by combining the transformed region with the original program. The use of regions of a program means that a transformation can consider a group of program commands, rather than only two commands as is the case with the method of Section 4.1.4 and transformation T_s. The region transformations will be shown to be correct, as will the result of combining a transformed region with the original program.

The definition of regions is based on the flow of control through a program. A region transformation can be defined by primitive recursion (Kleene, 1952), ensuring that a transformation always terminates. The regions will be given a semantics and a refinement relation will be defined. This allows the behaviour of a transformed region to be compared with the original region. Reasoning about the semantic properties of a region is based on an analysis of a *trace* through the region. Traces are used to show that transformations are correct by comparing the states produced at the cut-points of a region with those produced at the cut-points of the transformed region. This provides the framework in which to define the transformations for program abstraction; these transformations will be described in the Section 4.3.

4.2.1 Control Flow

The flow of control through a program p during the program execution is the order in which the commands of p are executed. The flow of control is determined by the commands of p and the initial state s in which program execution begins. Since the commands can include computed jumps, the actual flow of control during a program execution is undecidable. An alternative, which is used in program analysis and is based on the syntax of program commands, describes the possible flow of control through the program (Hecht, 1977)

The flow of control through a program p is defined by a successor relationship between commands. If it is possible for command c_1 to select command c_2 then c_1 *reaches* c_2. Command c_1 can select, and reaches, command c_2 if there is a state in which the expression assigned to the program counter *pc* by command c_1 is equivalent to the label of c_2.

Definition 4.7 *Reaches*

The *reaches* relation, $\longmapsto$, between commands has type $(\mathcal{C}_0 \times \mathcal{C}) \to$ *boolean* and is defined by recursion over the commands.

$$(:= al, l_e) \longmapsto c \stackrel{\text{def}}{=} \exists(s : \mathit{State}) : l_e \equiv_s \mathit{label}(c)$$
$$(\textbf{if } b \textbf{ then } c_t \textbf{ else } c_f) \longmapsto c \stackrel{\text{def}}{=} c_t \longmapsto c \vee c_f \longmapsto c$$
$$(l : c_1) \longmapsto c \stackrel{\text{def}}{=} c_1 \longmapsto c$$

If $c_1 \longmapsto c_2$ then c_1 is said to *directly reach* c_2 and c_2 is an *immediate successor* of c_1. □

The *reaches* relation between commands of a program p defines the edges in the flow-graph of p. Because the *reaches* relation compares the successor expressions with the labels of commands, a command which reaches c also reaches any command sharing a label with c.

Corollary 4.2 *For commands c_1, c_2 and c_3, if $c_1 \longmapsto c_2$ and c_2 and c_3 have the same label, $\mathit{label}(c_2) = \mathit{label}(c_3)$, then $c_1 \longmapsto c_3$.*

Proof. Straightforward, by induction on the command c_1. □

The definition of the *reaches* relation allows the possible flow of control through a program to be determined from the syntax of the commands. If command c_1 assigns expression e to the program counter and e is not strongly equivalent to the label of c_2, $e \not\equiv \mathit{label}(c_2)$ then c_1 does not reach c_2. However, if e is a basic label, $e \in \mathit{Labels}$, syntactically equal to the label of c_2, $e = \mathit{label}(c_2)$ or if e is strongly equivalent to such a label , then c_1 must select c_2.

Example 4.7 Let command c_1 be the assignment command with assignment list al. Command c_1 reaches command $c_2 \in \mathcal{C}$ whenever the successor expression of c_1 contains a variable. Assume $x \in \mathit{Names}$, $l_1 \in \mathit{Labels}$ and $l_e \in \mathcal{E}_l$ such that a name occurs in l_e:

$$:= (al, x) \longmapsto c_2 \qquad := (al, pc) \longmapsto c_2 \qquad := (al, l_e) \longmapsto c_2$$

If c_1 assigns only $\mathit{undef}(\mathcal{E}_l)$ to the program, $c_1 = (:= (al, \mathit{undef}(\mathcal{E}_l)))$, then c_1 cannot reach any command. If the successor expression of c_1 is a basic label the relation is decidable: for $l \in \mathit{Labels}$, $(:= (al, l) \longmapsto c_2)$ iff $l = \mathit{label}(c_2)$. □

Sequential composition preserves the reaches relation between commands. The composition of commands c_1 and c_2 results in a labelled, conditional command in which the false branch is c_1: if c_1 reaches c_3 then so will $c_1; c_2$. The result of $c_1; c_2$ will have the same label as c_1 and if command c_3 reaches c_1 then it will also reach $c_1; c_2$. Conversely, if $c_1; c_2$ reaches c_3 then either c_1 or c_2 will reach c_3.

Theorem 4.6 *Reaches and composition*

For commands $c_1, c_2 \in \mathcal{C}_0$ and $c_3 \in \mathcal{C}$,

1. *Composition:* *(Left)* $\dfrac{c_1 \longmapsto c_2}{c_1; c_3 \longmapsto c_2}$ *(Right)* $\dfrac{c_1 \longmapsto c_2}{c_1 \longmapsto c_2; c_3}$

2. *Reverse:* $\dfrac{c_1; c_2 \longmapsto c_3}{c_1 \longmapsto c_3 \vee c_2 \longmapsto c_3}$

Reaches **Through a Program**

The reaches relation between commands determines whether a command can select another for execution. When considering the flow of control in a program p, it is necessary to determine whether control can eventually pass from a command $c_1 \in p$ to a command $c_2 \in p$ through any number of intermediate commands. Command c_1 reaches c_2 through program p, written $c_1 \overset{p}{\longmapsto} c_2$ iff there is a sequence of commands of p, beginning with c_1 and ending with c_2, such that each command can pass control to the next command in the sequence. The *reaches* relation through a program is defined as the transitive closure of the reaches relation between commands.

Definition 4.8 *Reaches through a program*

A command c_1 reaches a command c_2 through a set $a \in Set(\mathcal{C})$ iff the transitive closure of the reaches relation restricted to a is *true*.

$$\frac{c_1, c_2 \in a \quad c_1 \longmapsto c_2}{c_1 \overset{a}{\longmapsto} c_2} \qquad \frac{c_1 \overset{a}{\longmapsto} c_2 \quad c_2 \overset{a}{\longmapsto} c_3}{c_1 \overset{a}{\longmapsto} c_3}$$

If $c_1 \overset{a}{\longmapsto} c_2$ then c_1 is said to reach c_2 through a. □

The set through which a command reaches another need not be a program of $\mathcal{L}$. It is not necessary for every command to be uniquely labelled, only for there to be a sequence of commands through which control could pass from c_1 to c_2.

In general, when a command c_1 is said to reach c_2, it will be assumed that it does so through some set of commands. A set a through which c_1 reaches c_2 contains commands which are required to establish $c_1 \overset{a}{\longmapsto} c_2$. A command $c' \in a$ is *necessary* for c_1 to reach c_2 if c_1 does not reach c_2 through $a - \{c'\}$. A property of the *reaches* relation is that $c_1 \overset{a}{\longmapsto} c_2$ iff there is a path through a from c_1 to c_2 (see Section D.6 of the appendix). The set through which a command reaches another can always be extended. If command c_1 reaches command c_2 through set a then it will do so through any superset of a.

Theorem 4.7 *Extending the* reaches *set*

For commands $c_1, c_2 \in \mathcal{C}$ and sets $a, b \in Set(\mathcal{C})$,

$$\frac{a \subseteq b \quad c_1 \overset{a}{\longmapsto} c_2}{c_1 \overset{b}{\longmapsto} c_2}$$

Proof. Straightforward, by induction on $\overset{a}{\longmapsto}$. □

Every command c in a program p begins a flow-graph made up of the commands of p which can be reached from c. However, there is no requirement that a command in a program can be reached from another command. A program can have commands which will never be selected for execution or which depend on the command which is selected in the initial state.

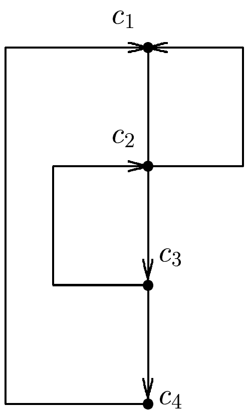

Figure 4.3: Flow-graph of Example (4.9)

Example 4.8 Let l_1 and l_2 be distinct labels, the program $\{l_1 : \textbf{goto}\, l_1, l_2 : \textbf{goto}\, l_2\}$ has two distinct flow-graphs. If execution begins with the command labelled l_1 then the command labelled l_2 will never be selected and the flow-graph contains the single command $l_1 : \textbf{goto}\, l_2$. The flow-graph beginning with the command labelled l_2 contains the single command $l_2 : \textbf{goto}\, l_2$. $\Box$

The reaches relations describe the possible flow of control through a program while a transition relation describes the actual flow of control. It follows that the *leads-to* relation is stronger than the *reaches*. If command c_1 of a program p is enabled in state s, command $c_2 \in p$ is enabled in state t and $s \overset{p}{\leadsto} t$ then eventually control will pass from c_1 to c_2 and c_1 reaches c_2 in p.

Theorem 4.8 reaches *and* leads-to

For program $p \in \mathcal{P}$, commands $c_1, c_2 \in \mathcal{C}$ and states $s, t \in State$,

1. *Immediate successors.*

$$\frac{\exists s, t : s \overset{c_1}{\leadsto} t \wedge \mathit{enabled}(c_2)(t)}{c_1 \longmapsto c_2}$$

2. *Reaches through a program.*

$$\frac{s \overset{P}{\leadsto} t \quad \mathit{enabled}(c_1)(s) \quad \mathit{enabled}(c_2)(t) \quad c_1, c_2 \in p}{c_1 \overset{p}{\longmapsto} c_2}$$

Since the relation *reaches* is weaker than the relation *leads-to*, it is safe to use *reaches* when analysing the flow of control through a program. That a command c_1 does pass control to a command c_2 is undecidable. That c_1 may pass control to c_2 can be established syntactically and also by showing that execution of c_1 does eventually lead to the selection of c_2.

Example 4.9 Let p be the program containing the commands

$$\begin{array}{l} l_1 : \textbf{goto } l_2 \\ l_2 : \textbf{if } b_1 \textbf{ then goto } l_1 \textbf{ else goto } l_3 \\ l_3 : \textbf{if } b_2 \textbf{ then goto } l_2 \textbf{ else goto } l_4 \\ l_4 : \textbf{goto } l_1 \end{array}$$

Let $c_1, \ldots, c_4$ be the commands of p labelled $l_1, \ldots, l_4$ respectively; the flow-graph of p is given in Figure (4.3). Command c_2 is the only immediate successor of c_1 and has c_1 and c_3 as immediate successors. As a consequence, command c_1 reaches c_1 through p, forming a loop in the flow-graph. Command c_1 also reaches c_3 through the path $c_1 \longmapsto c_2 \longmapsto c_3$.

The immediate successors of command c_3 are c_2 and c_4. Because command c_1 reaches c_3, c_1 also reaches c_4. Command c_4 has c_1 as an immediate successor and this forms a second loop through the flow-graph from command c_1 to itself. Because c_1 cannot reach c_3 except through command c_2, c_2 is necessary for both $c_1 \overset{p}{\longmapsto} c_3$ and $c_1 \overset{p}{\longmapsto} c_4$. □

4.2.2 Regions of a Program

The selection for execution of a command c of a program partitions the program into those commands to which c may pass control and those to which c cannot pass control. Command c begins a *region* of the program, containing only those commands which can be reached from c through the program. The region has an identified initial command, c, and a structure defined by the flow-graph formed by the *reaches* relation between commands. This structure can be used to define transformations which abstract from regions.

Definition 4.9 *Regions*

A *region* r is a pair (l, p) where p is a program and l the label of a command $c \in p$ beginning the region. Every other command in p is reachable from c through p.

$$\begin{array}{l} \textit{region?} : (\textit{Labels} \times \mathcal{P}) \to \textit{boolean} \\ \textit{region?}(l, p) \stackrel{\text{def}}{=} \exists(c \in p) : \textit{label}(c) = l \land \forall(c_1 \in p) : c = c_1 \lor c \overset{p}{\longmapsto} c_1 \end{array}$$

$\mathcal{R}$ is the set of all regions.

$$\begin{array}{l} \mathcal{R} : \textit{Set}(\textit{Labels} \times \mathcal{P}) \\ \mathcal{R} \stackrel{\text{def}}{=} \{(l, p) \mid \textit{region?}(l, p)\} \end{array}$$

□

A region (l, p) can be considered a form of program with an initial command, identified by the label l. Equally, it can be considered a (compound) command which begins execution when the program counter has the value of the region's label l. Transformations on a region can be defined using the control flow through the region to determine the order in which commands of the region may be executed. A similar approach is used in program analysis in which programs are partitioned into *basic blocks* or *intervals* (see Muchnick and Jones, 1981; Aho et al., 1986). The analysis of the program follows the order in which the commands of the basic block or interval may be executed.

Example 4.10 For the program p of Example (4.9), assume that c_1 is the first command to selected. The largest region of p beginning at c_1, (l_1, p) contains all commands which can be reached by c_1 and identifies the first command (c_1) by the label l_1.

A region can be formed from any subset of p. The region $(l_2, \{c_2, c_3\})$ begins at c_2 and contains the loop formed by c_2 and c_3. Each command also forms a region: $(l_4, \{c_4\})$ is a region. □

Region Operators

A region is made up of a label l and a program p: the region is said to be labelled with l and the *body* of the region is p. A region is constructed either from a single command or from the subset of a program. If p is a program and command $c \in p$ is labelled l then the region, r, of program p beginning with c is labelled l and the body of r is the subset of p which is reachable from c. When a region is constructed from a single command c, the label of the region is the label of c and the body is the singleton set $\{c\}$.

Definition 4.10 *Region constructors*

If p is a program and l is the label of a command in p, the result of *region*(l, p) is the region of p beginning at l.

$$\begin{array}{l}
\mathit{region} : (\mathit{Labels} \times \mathcal{P}) \to \mathcal{R} \\
\mathit{region}(l, p) \stackrel{\text{def}}{=} (l, p') \\
\text{where } p' \stackrel{\text{def}}{=} \{c \in p \mid \exists (c_1 \in p) : l = \mathit{label}(c_1) \wedge c_1 = c \vee c_1 \stackrel{p}{\longmapsto} c\}
\end{array}$$

A *unit* region is constructed, by function *unit*, from a single command. For region r, *unit?*(r) is *true* iff r is a unit region.

$$\begin{array}{ll}
\mathit{unit} : \mathcal{C} \to \mathcal{R} & \mathit{unit?} : \mathcal{R} \to \mathit{boolean} \\
\mathit{unit}(c) \stackrel{\text{def}}{=} (\mathit{label}(c), \{c\}) & \mathit{unit?}(r) \stackrel{\text{def}}{=} \exists (c : \mathcal{C}) : r = \mathit{unit}(c)
\end{array}$$

□

The label of a region identifies the command which begins the region, this command is the *head* of the region. Syntactic comparison of two regions is by the subset relation, extended to the body of a region, or by the equality of the label-program pairs. The *reaches* relation and the *leads-to* relation for regions are defined on the body of a region.

Definition 4.11 *Region components and operators*

For region $r = (l, p)$, the *label* of r is l and the *body* of r is p. The head of region r is the command in the body of r whose label is that of the region.

$$\begin{array}{lll} \mathit{label} : \mathcal{R} \to \mathit{Labels} & \mathit{body} : \mathcal{R} \to \mathcal{P} & \mathit{head} : \mathcal{R} \to \mathcal{C} \\ \mathit{label}(l, p) \stackrel{\text{def}}{=} l & \mathit{body}(l, p) \stackrel{\text{def}}{=} p & \mathit{head}(r) \stackrel{\text{def}}{=} \mathit{at}(\mathit{body}(r), \mathit{label}(r)) \end{array}$$

Command c is a member of a region r iff it is a member of the body of r: $c \in r \stackrel{\text{def}}{=} c \in \mathit{body}(r)$. A region r_1 is a *subset*, or *sub-region*, of region r_2 iff the body of r_1 is a subset of the body of r_2. Region r_1 is a *proper subset* of r_2 iff the body of r_1 is a proper subset of the body of r_2.

$$\begin{array}{ll} \_ \subseteq \_ : (\mathcal{R} \times \mathcal{R}) \to \mathit{boolean} & \_ \subset \_ : (\mathcal{R} \times \mathcal{R}) \to \mathit{boolean} \\ r_1 \subseteq r_2 \stackrel{\text{def}}{=} \mathit{body}(r_1) \subseteq \mathit{body}(r_2) & r_1 \subset r_2 \stackrel{\text{def}}{=} \mathit{body}(r_1) \subset \mathit{body}(r_2) \end{array}$$

Regions r_1 and r_2 are syntactically equal iff both the labels of r_1 and r_2 are the same and both contain the same commands.

$$\begin{array}{l} \_ = \_ : (\mathcal{R} \times \mathcal{R}) \to \mathit{boolean} \\ r_1 = r_2 \stackrel{\text{def}}{=} \mathit{label}(r_1) = \mathit{label}(r_2) \wedge \mathit{body}(r_1) = \mathit{body}(r_2) \end{array}$$

A state s leads to state t through region r if s leads to t through the body of r. Command c_1 reaches command c_2 through region r if c_1 reaches c_2 through the body of r.

$$s \stackrel{r}{\leadsto} t \stackrel{\text{def}}{=} s \stackrel{\mathit{body}(r)}{\leadsto} t \qquad c_1 \stackrel{r}{\longmapsto} c_2 \stackrel{\text{def}}{=} c_1 \stackrel{\mathit{body}(r)}{\longmapsto} c_2$$

□

Since no region can have an empty body, a unit region is the smallest possible region. The function *region* constructs the largest region of a program p. Assuming that there is a command $c \in p$ labelled with l, the region $\mathit{region}(l, p)$ contains all commands of p which can be reached by command c.

Corollary 4.3 *From program $p \in \mathcal{P}$, commands $c, c_1 \in \mathcal{C}$ and $l \in \mathit{Labels}$,*

1. *The body of a region constructed from a program p and label l by the function region is always a subset of p, $\mathit{body}(\mathit{region}(l, p)) \subseteq p$.*
2. *If $c \in p$ then $c \in \mathit{region}(\mathit{label}(c), p)$ and $\mathit{head}(\mathit{region}(\mathit{label}(c), p)) = c$.*
3. *If $c \stackrel{p}{\longmapsto} c_1$ then $c_1 \in \mathit{region}(\mathit{label}(c), P)$.*
4. *If $c \in p$ and $c_1 \in \mathit{region}(\mathit{label}(c), p)$ then either $c = c_1$ or $c \stackrel{p}{\longmapsto} c_1$.*

Proof. Straightforward, from definitions. □

The body of a region is a program and the proper subset relation, $\subset$, on regions is well-founded. There is therefore a strong induction scheme on regions, derived from the induction scheme for programs of Theorem (4.1).

Theorem 4.9 *Induction on regions*

For any property Φ, with type $\mathcal{R} \to boolean$,

$$\frac{\forall (r : \mathcal{R}) : (\forall (r' : \mathcal{R}) : r' \subset r \Rightarrow \Phi(r')) \Rightarrow \Phi(r)}{\forall (r : \mathcal{R}) : \Phi(r)}$$

To prove a property of region r using the induction scheme of Theorem (4.9) requires a method for extracting a proper sub-region from r. The sub-regions which begin with the immediate successors of the head of r and constructed from the $\mathit{body}(r) - \{\mathit{head}(r)\}$ are proper sub-regions of r.

Definition 4.12 *Immediate sub-regions of a region*

The result of *rest* applied to region r is the set of *immediate sub-regions* of r.

$$\begin{array}{l} \mathit{rest} : \mathcal{R} \to \mathit{FiniteSet}(\mathcal{R}) \\ \mathit{rest}(r) \stackrel{\text{def}}{=} \{r' : \mathcal{R} \mid \exists (c : \mathcal{C}) : c \in p \wedge \mathit{head}(r) \longmapsto c \wedge r' = \mathit{region}(\mathit{label}(c), p)\} \\ \text{where } p \stackrel{\text{def}}{=} \mathit{body}(r) - \{\mathit{head}(r)\} \end{array}$$

□

The set of regions in $\mathit{rest}(r)$ are uniquely labelled since each region is constructed from a command which is uniquely labelled. In addition, the sub-regions of a region r which are members of $\mathit{rest}(r)$ are also proper subsets of r.

Corollary 4.4 *For regions r and r', $r' \in \mathit{rest}(r) \Rightarrow r' \subset (r)$.*

Proof. Immediate, from definition and Corollary (4.3). □

If r' is a sub-region of region r such that $r' \in \mathit{rest}(r)$ then the head of r' is an immediate successor of the head of r, $\mathit{head}(r) \longmapsto \mathit{head}(r')$. This can be used, with the induction scheme of Theorem (4.9), to relate the property assumed of the sub-region r' with the property to be proved of r. Because there is a path from the head of a region r to any other command $c \in r$ (when $c \neq \mathit{head}(r)$), either $\mathit{unit?}(r)$ or there is a region $r' \in \mathit{rest}(r)$. This provides the cases for proofs by induction on a region. The sub-regions of a region r in $\mathit{rest}(r)$ can also be used to define a transformation T by primitive recursion on a region. If the result of $T(r)$ is defined in terms of $T(r')$, for $r' \in \mathit{rest}(r)$, then the recursion is well-founded, and will eventually terminate, since r' is a proper subset of r (Kleene, 1952).

Loops in Regions

The method used to abstract from a region depends on identifying the loops in the region. The presence of loops in a program, such as the body of a region, is determined using the flow of control through the program. A program p contains a loop if there is a command $c \in p$ which can reach itself through p, $c \xmapsto{p} c$. Conversely, a program p is loop-free iff there is no command $c \in p$ such that $c \xmapsto{p} c$.

Definition 4.13 A program p is loop-free if no command in p can reach itself.

$$\begin{aligned}&\mathit{loopfree?} : \mathit{Set}(\mathcal{C}) \to \mathit{boolean}\\ &\mathit{loopfree?}(A) \stackrel{\text{def}}{=} \forall (c \in A) : \neg(c \xmapsto{A} c)\end{aligned}$$

□

A program p contains a loop iff *loopfree?*(p) is *false*: there is at least one command c such that $c \xmapsto{p} c$. A command c_2 of p is *necessary* for a loop in p if $p - \{c_2\}$ is loop-free. e.g. If $c \in p$ and $c \xmapsto{p} c$ but not $c \xmapsto{p-\{c_2\}} c$ then c_2 is necessary for the loop containing c. A cut-point for the loop containing c is a command, c_1, such that $c_1 \xmapsto{p} c$. The cut-point c_1 is part of the loop, $c_1 \xmapsto{p} c_1$, and can be reached by c, $c \xmapsto{p} c_1$.

Classes of region can be defined by the number and type of commands necessary for loops in a region. In a *loop-free* region there is no command which can reach itself and therefore no command is necessary for a loop. In a *single loop*, the region contains at least one loop and the head of the region is necessary for every loop in the region.

Definition 4.14 *Classes of region*

A region is *loop-free* iff its body does not contain a loop.

$$\begin{aligned}&\mathit{loopfree?} : \mathcal{R} \to \mathit{boolean}\\ &\mathit{loopfree?}(r) \stackrel{\text{def}}{=} \mathit{loopfree?}(\mathit{body}(r))\end{aligned}$$

A region is a *single loop* if the body of the region excluding the head is loop-free.

$$\begin{aligned}&\mathit{single?} : \mathcal{R} \to \mathit{boolean}\\ &\mathit{single?}(r) \stackrel{\text{def}}{=} \mathit{loopfree?}(\mathit{body}(r) - \{\mathit{head}(r)\})\end{aligned}$$

□

The *general loops* are regions which are neither loop-free nor a single loop. These contain at least one loop which is independent of the head of the region, $\neg\mathit{loopfree?}(\mathit{body}(r) - \{\mathit{head}(r)\})$. There is an ordering between the classes of region: the class of general loops includes all regions and the single loops include all loop-free regions.

$$\mathit{loopfree?}(r) \Rightarrow \mathit{single?}(r)$$

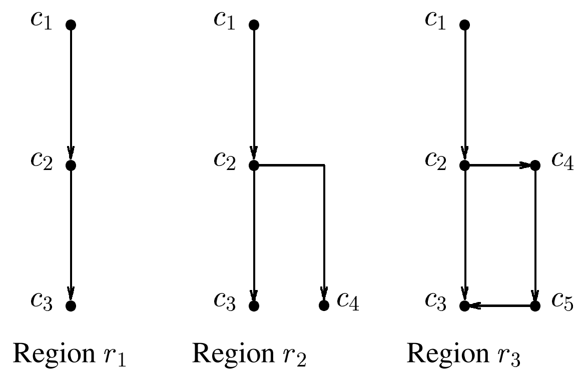

Figure 4.4: Loop-Free Regions

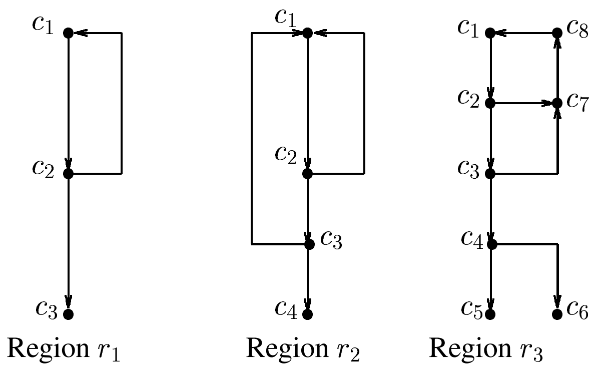

Figure 4.5: Single Regions

A single region is simpler than a general region, for abstraction or verification, since the only command necessary for all loops is the head of the region. The only cut-point needed for a single loop region r is therefore the head of r.

The methods of Floyd (1967) and Burstall (1974) for program verification can be described in terms of general and single regions. In both, a program p is broken down into sequences of commands between cut-points and there is an identified command which begins the program. The program p forms a general region and each sequence of commands forms a single loop region, beginning with a cut-point and constructed from the subset of p of which excludes all other cut-points. To verify that the general region satisfies the program specification, each of the single regions is shown to satisfy an intermediate specification.

Example 4.11 The head of each region of Figure (4.4), Figure (4.5) and Figure (4.6) is the command c_1. The regions of Figure (4.4) are loop-free: there is no command that can reach itself through the region. The regions of Figure (4.5) are single loops. No command in any region can reach itself without passing through the head of the region. The regions of Figure (4.6) are

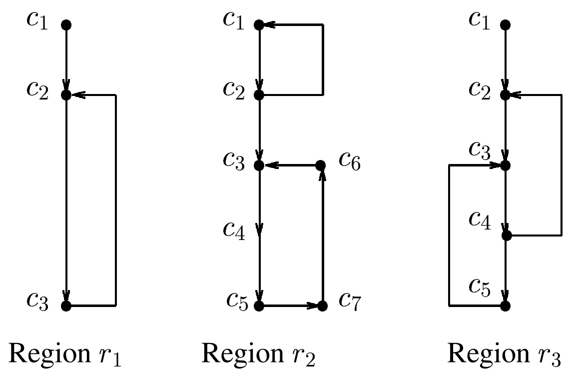

Figure 4.6: General Regions

neither loop-free nor single loops. Each region has a command which can reach itself without passing through the head of the region. □

4.2.3 Semantics of Regions

A region is a component of a program which is executed when control passes to the region's head. The region is *enabled* when the head of the region is selected for execution. Continuing execution of the program requires the execution of commands of the region. As each command of the region is selected and executed it produces a new state. The region terminates, passing control out of the region when a state is produced in which no command of the region is selected; this state is *final* for the region. If a command of region r fails in a state s or region r, beginning in s, produces a state in which a command of the region fails, then the region *halts* in state s.

Definition 4.15 *Enabled, final and halt*

Region r is *enabled* in a state s if the head of r is enabled in s. A state is final for region r if it is final for the body of r.

$$\begin{array}{ll} \textit{enabled} : \mathcal{R} \to \textit{State} \to \textit{boolean} & \textit{final?} : \mathcal{R} \to \textit{State} \to \textit{boolean} \\ \textit{enabled}(r)(s) \stackrel{\text{def}}{=} \mathcal{I}_e(\textit{pc})(s) = \textit{label}(r) & \textit{final?}(r)(s) \stackrel{\text{def}}{=} \textit{final?}(\textit{body}(r))(s) \end{array}$$

Region r halts in state s if the $\textit{body}(r)$ halts in s or in a state produced by r from s.

$$\begin{array}{l} \textit{halt?} : \mathcal{R} \to \textit{State} \to \textit{boolean} \\ \textit{halt?}(r)(s) \stackrel{\text{def}}{=} \textit{halt?}(\textit{body}(r))(s) \lor \exists t : s \stackrel{r}{\leadsto} t \land \textit{halt?}(\textit{body}(r))(t) \end{array}$$

□

A region r of a program p will normally be considered separately from p: any changes made to r by a transformation T will have no affect on p. The abstraction of p is obtained by combining the body of the transformed region with p: $p \uplus \mathit{body}(T(r))$. To ensure that this preserves the correctness of p, the result of transforming the region r of p must preserve the correctness of r. This requires a semantics for the regions to allow the effect of a transformation on the region to be compared with the original behaviour of the region.

A region can be interpreted as a (possibly terminating) program or as a compound command. The interpretation of r as a program considers all states produced by r. The interpretation of a region as a compound command is stronger. Region r relates states s and t if r is enabled in s, r produces t and t is final for r. This describes the external behaviour of regions: states s and t are the states in which r begin and ends. The internal behaviour of the region, made up of the intermediate states needed to produce t from s, is hidden. This is consistent with the interpretation of commands of $\mathcal{L}$ and will be used for the semantics of regions.

Definition 4.16 *Semantics of regions*

A region r begins in state s and produces state t if r is enabled in s, produces t and t is final for r. The semantics of the regions is defined by interpretation function $\mathcal{I}_r$.

$$\mathcal{I}_r : \mathcal{R} \to (\mathit{State} \times \mathit{State}) \to \mathit{boolean}$$
$$\mathcal{I}_r(r)(s,t) \stackrel{\text{def}}{=} \mathit{enabled}(r)(s) \wedge s \stackrel{r}{\leadsto} t \wedge \mathit{final?}(r)(t)$$

□

States which are produced by a region r but which are not final for r are intermediate states of the execution of r and are hidden by the interpretation, $\mathcal{I}_r$, of regions. Unlike a command, a region r which does not terminate does not necessarily halt. The region may contain a loop which never terminates but continuously produces (hidden) intermediate states. The interpretation of region r is *false* for all pairs of states: $\mathcal{I}_r$ considers only the external states of a region.

Refinement of Regions

To show that a transformation constructs an abstraction of a region, it must be possible to show that the original region is a refinement of the transformed region. Two forms of refinement can be defined between regions. When the regions are a considered as compound commands, region r' is refined by region r if $\mathcal{I}_r(r')(s,t) \Rightarrow \mathcal{I}_r(r)(s,t)$, for states s,t. This is the refinement relation of structured programs (Back & von Wright, 1989). The second form of refinement considers a region as a program: the refinement relation is interpreted as program refinement between the region bodies, $\mathit{body}(r) \sqsubseteq \mathit{body}(r')$. This compares the internal behaviour of the regions and ensures that the abstracting a region of a program p results in an abstraction of p. However, this interpretation does not require the regions to terminate in the same state. This is too weak since it would allow a region to contain a non-terminating loop which is not present in its abstraction.

Both forms of refinement will be used: the refinement relation, $\_ \sqsubseteq \_$, between regions interprets a region as both a command and a program. Region r_1 is an abstraction of region r_2 iff both $\mathcal{I}_r(r_1)(s,t) \Rightarrow \mathcal{I}_r(r_2)(s,t)$ (for all $s,t \in \mathit{State}$) and $\mathit{body}(r_1) \sqsubseteq \mathit{body}(r_2)$.

Definition 4.17 *Refinement*

Region r_1 is refined by region r_2, $r_1 \sqsubseteq r_2$, iff r_1 and r_2 have the same label, the body of r_1 is refined by the body of r_2 and every final state produced by r_1 is also final for r_2.

$$\_ \sqsubseteq \_ : (\mathcal{R} \times \mathcal{R}) \rightarrow \mathit{boolean}$$
$$r_1 \sqsubseteq r_2 \stackrel{\text{def}}{=} \begin{cases} \mathit{label}(r_1) = \mathit{label}(r_2) \wedge \mathit{body}(r_1) \sqsubseteq \mathit{body}(r_2) \\ \wedge \forall (s, t : \mathit{State}) : s \stackrel{r_1}{\leadsto} t \wedge \mathit{final?}(r_1)(t) \Rightarrow \mathit{final?}(r_2)(t) \end{cases}$$

□

This definition of refinement is stronger than is needed for verification, for which only the external behaviour of regions is strictly required. However, it exposes the internal behaviour of regions which would be hidden if only the interpretation of regions, $\mathcal{I}_r$, was considered. This allows the effect of a transformation on the internal behaviour (and therefore the loops) of a region to be considered. A region r' abstracts a region r, $r \sqsubseteq r'$, only if all loops in r are present in r' and r' begins and ends whenever r' begins and ends.

Refinement between regions is a partial order (a region r_1 refines itself) and has the basic property of transitivity.

Corollary 4.5 *Refinement between regions is transitive. For regions $r_1, r_2, r_3 \in \mathcal{R}$,*

$$\frac{r_1 \sqsubseteq r_2 \quad r_2 \sqsubseteq r_3}{r_1 \sqsubseteq r_3}$$

Refinement between regions is stronger than refinement between programs. If regions r_1 and r_2 refine each other than they are equivalent.

Theorem 4.10 *Equivalence*

For regions $r_1, r_2 \in \mathcal{R}$ and states $s, t \in \mathit{State}$,

$$\frac{r_1 \sqsubseteq r_2 \quad r_2 \sqsubseteq r_1}{\mathcal{I}_r(r_1)(s,t) = \mathcal{I}_r(r_2)(s,t)}$$

Proof. From the definition of $\sqsubseteq$, $\mathit{label}(r_1) = \mathit{label}(r_2)$ and r_1 is enabled in s iff r_2 is enabled in s. From $\mathit{body}(r_1) \sqsubseteq \mathit{body}(r_2)$ and $\mathit{body}(r_2) \sqsubseteq \mathit{body}(r_1)$, $s \stackrel{r_1}{\leadsto} t$ iff $s \stackrel{r_2}{\leadsto} t$. Since $s \stackrel{r_1}{\leadsto} t$ and $s \stackrel{r_2}{\leadsto} t$, t is final for r_1 iff it is final for r_2. □

A region r_2 which is semantically equivalent to region r_1 does not necessarily refine r_1. Refinement of regions requires that the body of r_2 is a refinement of the body of r_1, $\mathit{body}(r_1) \sqsubseteq \mathit{body}(r_2)$. Semantic equivalence requires only that interpretation of the two regions is the same, $\mathcal{I}_r(r_1)(s,t) = \mathcal{I}_r(r_2)(s,t)$, and does not consider the intermediate states which can be produced by the regions. As with programs, a region which always fails can be refined by any other region.

Theorem 4.11 *For any regions r_1, r_2 and state s,*

$$\frac{\forall s : \textit{halt?}(r_1)(s)}{r_1 \sqsubseteq r_2}$$

A consequence of Theorem (4.11) is that if the transformation of region r results in a region r' which always fails then r' is a valid abstraction of r. A transformation T can be shown to construct a useful abstraction of a region r if the result of $T(r)$ fails in a state s only if the region r fails in s.

Example 4.12 Assume $x \in$ *Names* and distinct labels $l, l_1, l_2 \in$ *Labels* and states $s, t \in$ *State*. Let r_1 be the region *unit*$(l : (x := 2, l_2))$ and r_2 be the region $(l, \{l : (x := 1, l_1), l_1 : (x := x + 1, l_2))\}$.

The regions r_1 and r_2 are semantically equivalent: $\mathcal{I}_r(r_1)(s, t) \Leftrightarrow \mathcal{I}_r(r_2)(s, t)$. Both are enabled in the same states, end in the same states (in which the program counter has the value l_2) and assign the value 2 to the name x.

Region r_1 is not a refinement of r_2 since the body of r_1 is not a refinement of the body of r_2. Assume that $l_1 : (x := x + 1, l_2)$ is enabled in s' and $s' \overset{r_2}{\leadsto} t$. The only command in r_1 is not enabled in s' and therefore $s' \overset{r_1}{\leadsto} t$ is *false*. □

4.2.4 Transition Relation: Traces

The refinement relations for programs and for regions do not consider the size of programs and regions being compared. A region r_1 can be refined by region r_2, $r_1 \sqsubseteq r_2$, even if r_1 contains more commands than r_2. This is because refinement is based on the *leads-to* relation, which considers the states produced by a program but not how the states are produced. This makes establishing a refinement relationship between programs (and therefore regions) difficult. For region r_1 to be refined by region r_2, every state produced by r_1 must be produced by r_2. Region r_1 can produce a state by executing one or more loops, which may be made up of any number of commands. The *leads-to* relation considers either a single command or all commands of the region. To show that a loop in r_2 eventually produces a state requires the states produced by each iteration of the loop to be considered. This can require an arbitrary number of commands (making up the body of the loop) to be considered. The *leads-to* relation does not allow induction on a loop, only on a program. It cannot, therefore, be used to compare the behaviour of arbitrary regions (or programs).

For example, assume T is a transformation which must abstract from any region r. To show $T(r) \sqsubseteq r$, it is necessary to show that for any states s, t, $s \overset{T(r)}{\leadsto} t \Rightarrow s \overset{r}{\leadsto} t$. Any number of loops either in $T(r)$ or r may be executed to produce state t from s. Assume there is a single command $c \in T(r)$ such that $s \overset{c}{\leadsto} t$, the proof of abstraction must show that $s \overset{c}{\leadsto} t \Rightarrow s \overset{r}{\leadsto} t$. Command c may form a loop in $T(r)$, while the equivalent loop in r may require several commands in r. The *leads-to* relation does not allow these commands to be considered: it is only possible to show that a loop in r produces t from s if the loop in r is also made up of a single command.

The behaviour of regions can be compared by considering the states produced by loop-free sequences of commands in the regions (this is the basis of the methods of Floyd, 1967, and Burstall, 1974). Informally, a statement that $s \overset{r}{\leadsto} t$ (for region r) is broken down into the loop-free sequences of commands, called *traces*, needed to produce s from t. A trace describes the states produced by a loop-free sequence of commands, such as those which make up a loop. The behaviour of a loop in the region can then be established from the states produced by the traces through the loop.

A *trace* through a program is a transition relation between states produced by one or more commands of the program, each of which is executed at most once. The subset of p containing the commands necessary to establish a trace from s to t is the *trace set* of p from s to t.

Definition 4.18 *Traces*

There is a trace from state s to state t through program p iff predicate $\mathit{trace}(p, s, t)$ is *true*. The predicate *trace* has type $(\mathcal{P} \times \mathit{State} \times \mathit{State}) \to \mathit{boolean}$ and is inductively defined:

$$\frac{c \in p \quad s \overset{c}{\leadsto} t}{\mathit{trace}(p, s, t)} \qquad \frac{c \in p \quad s \overset{c}{\leadsto} u \quad \mathit{trace}(p - \{c\}, u, t)}{\mathit{trace}(p, s, t)}$$

The trace set $\mathit{tset}(p, s, t)$ is the subset of commands of program p necessary to establish a trace from s to t.

$$\mathit{tset} : (\mathcal{P} \times \mathit{State} \times \mathit{State}) \to \mathit{Set}(\mathcal{C})$$
$$\mathit{tset}(p, s, t) \overset{\text{def}}{=} \begin{cases} \{c \in p \mid s \overset{c}{\leadsto} t\} \\ \cup\{c \in p \mid \exists(u : \mathit{State}) : s \overset{c}{\leadsto} u \wedge \mathit{trace}(p - \{c\}, u, t)\} \\ \cup\{c \in p \mid \exists(c_1 \in p, u : \mathit{State}) : s \overset{c_1}{\leadsto} u \wedge c \in \mathit{tset}(p - \{c_1\}, u, t\} \end{cases}$$

□

There is a trace from state s to state t through program p iff there is a trace through the trace set $\mathit{tset}(p, s, t)$.

Lemma 4.3 *For states $s, t \in \mathit{State}$ and program $p \in \mathcal{P}$,*

$$\mathit{trace}(p, s, t) \Leftrightarrow \mathit{trace}(\mathit{tset}(p, s, t), s, t)$$

If there is a trace though program p from state s to state t and command $c \in p$ is enabled in state t, then either c is used to establish the trace from s to t, $c \in \mathit{tset}(p, s, t)$; c halts in state t or c can extend the trace to a state u, $t \overset{c}{\leadsto} u$. If command c is necessary to establish the trace from s to t, $c \in \mathit{tset}(p, s, t)$, then the trace cannot be extended and there is a *maximal trace* from state s to state t through p. A *restricted maximal trace* is a maximal trace through program p which begins with any command in p but which does not execute any other command in a given set a.

Definition 4.19 *Maximal traces*

For program p and states s, t, a trace is maximal iff t is final for $p - \textit{tset}(p, s, t)$.

$$\begin{aligned} &\textit{mtrace} : (\mathcal{P} \times \textit{State} \times \textit{State}) \to \textit{boolean} \\ &\textit{mtrace}(p, s, t) \stackrel{\text{def}}{=} \textit{trace}(p, s, t) \wedge \textit{final?}(p - \textit{tset}(p, s, t))(t) \end{aligned}$$

There is a maximal trace restricted in set a from state s to state t through a program p iff there is a maximal trace from s to t through $(p - a) \cup \{c\}$ where $c \in p$ is enabled in s.

$$\begin{aligned} &\textit{rmtrace} : \textit{Set}(\mathcal{C}) \to (\mathcal{P} \times \textit{State} \times \textit{State}) \to \textit{boolean} \\ &\textit{rmtrace}(a)(p, s, t) \stackrel{\text{def}}{=} \exists c \in p : \textit{enabled}(c)(s) \wedge \textit{mtrace}((p - a) \cup \{c\}, s, t) \end{aligned}$$

□

A maximal trace, *mtrace*, describes the states produced by executing the largest number of commands of a program without repeating any command. Maximal trace relations allow the semantic properties of a program to be established by reasoning about the longest (loop-free) paths through the flow-graph of the program. Typically, these paths consist of the commands between the cut-points in a program. This is similar to the use of paths in program verification and in analysis to reason about the changes made to the program variables by iteration of a loop (e.g Floyd, 1967; Tarjan, 1981; Aho et al., 1986).

Example 4.13 Let p be a program with commands $c_1, c_2, c_3 \in \mathcal{C}$ and let s_1, s_2, s_3, s_4 be states such that $s_1 \stackrel{c_1}{\leadsto} s_2 \stackrel{c_2}{\leadsto} s_3 \stackrel{c_3}{\leadsto} s_4 \stackrel{c_1}{\leadsto} s_2$.

There is a trace from state s_1 to state s_4 through program p, $\textit{trace}(p, s_1, s_2)$, with trace set $\{c_1, c_2, c_3\}$. There are also traces from s_1 to s_2, from s_2 to s_2 and from s_1 to s_3.

There is a maximal trace from s_1 to s_4: c_1 begins the trace in state s_1 and is enabled in state s_4. The other maximal traces are from s_2 to s_2, from s_3 to s_3 and from s_4 to s_4.

Let $a = \{c_2\}$. There is a restricted maximal trace from s_1 to s_2 through p, $\textit{rmtrace}(a)(p, s_1, s_2)$. There is also a restricted maximal trace from s_2 to s_2, $\textit{rmtrace}(a)(p, s_2, s_2)$. □

Properties of Traces

The trace relations are used to establish the semantic properties of region transformations. The formal description of the properties of the traces are given in Appendix D (as part of the correctness proofs for region transformations). The properties needed when reasoning about region transformations relate the behaviour of a region with the states established by maximal traces. Two properties of the maximal trace, *mtrace*, are important: the first relates a maximal trace with commands beginning a loop. Assume program p and states s, t such that $\textit{mtrace}(p, s, t)$ and a command $c \in p$ is enabled in t. Because there is a maximal trace from s to t, command c begins a loop in p. Command c must have been executed in the trace from s to t and c is the first command to be re-selected for execution. The trace *mtrace*(p, s, t) therefore describes the behaviour of p up to the first repetition of a command.

A second property is an equivalence between the semantics of regions and the transitive closure of a maximal trace. If the head of r is enabled in s and t is final for r, then the interpretation of r is equivalent to the transitive closure of the maximal trace through r:

$$\mathcal{I}_r(r)(s,t) = \textit{enabled}(r)(s) \wedge \textit{mtrace}^{+}(\textit{body}(r), s, t) \wedge \textit{final?}(r)(t) \tag{4.1}$$

Equation (4.1) allows the behaviour of a region r to be broken down to maximal traces through r, each of which ends with a command beginning a loop. Informally, this allows a transformation T on a region r to be shown to satisfy $\mathcal{I}_r(T(r))(s,t) \Rightarrow \mathcal{I}_r(r)(s,t))$ by showing that *mtrace*$(\textit{body}(T(r)), s', t') \Rightarrow$ *mtrace*$(\textit{body}(r), s', t')$ (for any $s, s', t, t' \in$ *State*).

To show refinement between regions, $T(r) \sqsubseteq r$, it is also necessary to compare the internal behaviour of the regions, to establish *body*$(T(r)) \sqsubseteq$ *body*(r). This uses an equivalence between the reflexive transitive closure of a maximal trace and the *leads-to* relation: for any $r \in \mathcal{R}, s, t, u \in$ *State*, $s \overset{r}{\leadsto} t$ iff *mtrace*$^{*}(\textit{body}(r), s, u) \wedge$ *trace*$(\textit{body}(r), u, t)$. Region $T(r)$ can therefore be shown to abstract from region r, *body*$(T(r)) \sqsubseteq$ *body*(r), by establishing the truth of *mtrace*$(\textit{body}(T(r)), s, t) \Rightarrow$ *mtrace*$(\textit{body}(r), s, t)$ and *trace*$(\textit{body}(T(r), s, t) \Rightarrow$ *trace*$(\textit{body}(r), s, t)$, for any $s, t \in$ *State*. When the region r is a single loop, *single?*(r), the head of r begins and ends each maximal trace (since *head*(r) is the only command needed for each loop in r). The semantic properties of the single region r can then be established by induction on the maximal traces and the traces through r beginning with the head of r.

When r is a general region, a loop can begin at any command of r and these commands are hidden by *mtrace* (which requires the commands to executed at least once before they are detected). The restricted maximal trace describes a maximal trace up to but not including identified commands. If a is a set of cut-points chosen for a region r then *rmtrace*$(a)(\textit{body}(r), s, t)$ is a maximal trace which may begin with a cut-point but cannot otherwise use any cut-point. When all commands beginning a loop are chosen as cut-points, the properties *rmtrace* are similar to the the *mtrace* relation. If a is the set of command in a region r which begin loops in r then:

$$\frac{c \in a \quad \textit{enabled}(c)(s)}{\textit{mtrace}^{+}(\textit{body}(r), s, t) = \textit{rmtrace}^{+}(\textit{body}(r), s, t)} \tag{4.2}$$

Equation (4.2) and Equation (4.1) allow the behaviour of any region to be broken down into a series of loop-free sequences. The comparison of regions can therefore be based on the behaviour of the loops in the regions rather than on the states that each region produces. The properties of a trace (describing a sequence of commands through a program) can be established by induction on the relation *trace*. The properties of loops in the region can be established by induction on the transitive closure of *mtrace*. This is consistent with the approach used by Floyd (1967) to verify a program by partitioning the program into a set of loop-free sequences of commands, allowing properties of the loops to be established by induction on the sequences of commands.

4.3 Program Abstraction

The abstraction of a program p is formed by abstracting a region r of p. The region r would be constructed during the verification of program p, beginning with some program command of interest, such as a program cut-point or a command identified by an intermediate assertion. The result of abstracting region r will be a region r' satisfying $r' \sqsubseteq r$. The abstraction of p is the result of combining the commands of r' with the program p, $\mathit{body}(r') \uplus p \sqsubseteq p$. This uses the property of Theorem (4.5) together with the fact that abstracting a region of a program results in an abstraction of the program.

Theorem 4.12 *For* $r', r \in \mathcal{R}$*,* $p \in \mathcal{P}$ *and* $l \in \mathit{Labels}$*,*

$$\frac{r' \sqsubseteq r \quad \mathit{body}(r) \subseteq p}{\mathit{body}(r') \sqsubseteq p}$$

Proof. From the definition of $r' \sqsubseteq r$, $\mathit{body}(r') \sqsubseteq \mathit{body}(r)$ and the proof is immediate from Theorem (4.5). □

Basing program abstraction on a region allows a transformation to be defined on the region's structure, determined by the flow of control through the region. The region also has an identified initial command (the head of the region) which can be used to analyse the flow of control through the region, to detect the loops in the region.

4.3.1 Abstracting from Regions

The method used here to abstract from regions is based on the application of two transformations. The first, a *path transformation* T_1, abstracts from single regions (satisfying predicate *single?*). The second, a *general transformation* T_2, is defined in terms of T_1 and can be applied to any region r; the result of $T_2(r)$ is an abstraction of r. The method of abstracting from r is similar to the approach used in the proof methods for program verification (Floyd, 1967; Burstall, 1974): the abstraction of a region is formed by abstracting the sequences of commands between cut-points in the region.

The two transformations, T_1 and T_2, abstract a region r as follows: the general transformation, T_2, finds the loops in r and forms the set of cut-points, $\mathit{cuts}(r)$. Each command in $\mathit{cuts}(r)$ is either the head of r or begins a loop in r. The sequences of commands between cut-points are formed as a set of regions, $\mathit{loops}(r)$. Each region $r_1 \in \mathit{loops}(r)$ is a single region, $\mathit{single?}(r)$, which begins with a cut-point of r and excludes all other cut-points. The region r_1 contains all sequences of commands from a cut-point up to but excluding any other cut-point. Each of these sequences describes a path through the flow-graph of the region r. The path transformation T_1 is applied to each region in $\mathit{loops}(r)$, resulting in a set of commands. Each of these commands is the abstraction of sequences of commands between cut-points of r. The result of $T_2(r)$ is the region constructed from this set of commands.

The description of the transformations will begin with the definition of path transformation T_1 and a description of its properties. This will be followed by the definition of the general transformation T_2 and a description of its properties. The required properties of the transformations are that both abstract from their argument. Additional properties of the transformations strengthen the refinement relation between a region and its abstraction. Finally, factors limiting the use of the general transformation to abstract from regions will be discussed. The proofs of the properties for both transformations are given in Appendix D.

4.3.2 Path Transformation of a Region

The path transformation T_1 constructs an abstraction of a single loop region r by applying sequential composition to the commands of r in the order in which they may be executed. The transformation is defined by recursion on r. When r is a unit region, there are no commands in r other than *head*(r) and the result of $T_1(r)$ is *head*(r). When r is not a unit region, the commands which follow the head of r form the regions of *rest*(r). The transformation T_1 is applied to each of the regions in *rest*(r) to obtain a set a of commands. The head of r is then composed with each command in the set a. This results in a command c which is equivalent to a path in r and the abstraction of r is the unit region *unit*(c).

To compose the head of the region with a set of commands, the sequential composition operator is generalised to allow its application to a command and a finite set of commands.

Definition 4.20 *Composition over a set*

There is a function *choose* of type *Set*$(T) \to T$ satisfying, for any type T and set S : *Set*(T):

$$\frac{S \neq \{\}}{\mathit{choose}(S) \in S}$$

The composition of a command c with a finite set a of commands is defined by recursion over a.

$$\begin{array}{l}
_\,;\,_ \; : (\mathcal{C} \times \mathit{FiniteSet}(\mathcal{C})) \to \mathcal{C} \\
c; a \stackrel{\text{def}}{=} \begin{cases} c & \text{if } a = \{\} \\ (c; a - \{c'\}); c' & \text{otherwise} \end{cases} \\
\text{where } c' = \mathit{choose}(a)
\end{array}$$

□

Composition of c with a set of commands A, $(c; A)$, has properties similar to the sequential composition operator between commands, provided that no command in set A is selected for execution by another command in the set.

Example 4.14 Let c_1, c_2 and c_3 be commands such that $\neg(c_2 \longmapsto c_3)$, $\neg(c_3 \longmapsto c_2)$ and *label*$(c_2) \neq$ *label*(c_3). The result of composing c_1 with $\{c_2, c_3\}$, $c_1; \{c_2, c_3\}$, is either $c_1; c_2; c_3$ or $c_1; c_3; c_2$. For $s, t \in$ *State*, $\mathcal{I}_c(c_1; \{c_2, c_3\})(s, t)$ is *true* iff there is a state u such that $\mathcal{I}_c(c_1)(s, u)$

and either $\mathcal{I}_c(c_2)(u, t)$ or $\mathcal{I}_c(c_3)(u, t)$ or neither c_2 nor c_3 are enabled in u and $u = t$ (Theorem 3.7). Because *label*$(c_2) \neq$ *label*(c_3), the result of $\mathcal{I}_c(c_1; \{c_2, c_3\})(s, t)$ is equivalent to one of $\mathcal{I}_c(c_1)(s, t), \mathcal{I}_c(c_1; c_2)(s, t)$ or $\mathcal{I}_c(c_1; c_3)(s, t)$. There cannot be a state u such that $\mathcal{I}_c(c_1; c_2)(s, u)$ and $\mathcal{I}_c(c_3)(u, t)$, since this would contradict $\neg(c_2 \longmapsto c_3)$. □

The path transformation T_1 uses the structure of a region, defined by the *reaches* relation, to select and compose the region's commands. The result of applying T_1 to a region r, $T_1(r)$, is the command which results from the composition of the head of r with the set of commands formed by applying T_1 to each region in *rest*(r).

Definition 4.21 *Path transformation*

The path transformation, T_1, of a region r is defined by recursion over the region.

$$T_1 : \mathcal{R} \to \mathcal{C}$$
$$T_1(r) \stackrel{\text{def}}{=} \begin{cases} \mathit{head}(r) & \text{if } \mathit{unit?}(r) \\ \mathit{head}(r); (T_1(\mathit{rest}(r))) & \text{otherwise} \end{cases}$$

□

The selection of commands by transformation T_1 is determined by the flow of control through the region. If control passes to the head of a region r then any command c_1 which may be selected by *head*(r) will form a region $r_1 \in$ *rest*(r) and *head*$(r) \longmapsto$ *head*(r_1). The region r_1 will contain every command which is reachable from *head*(r_1) in r and, if *single?*(r), will also be loop-free (since *head*$(r) \notin$ *body*(r')). The application of T_1 to r' will repeat this process: composing the commands which are reachable from *head*(r') in the order in which they may be executed. The result is a single command which abstracts from every path through the region beginning with the head of the region.

Example 4.15 The abstraction of region r_1 of Figure (4.4) by the path transformation T_1 is as follows. The flow of control through r_1 is: $c_1 \longmapsto c_2 \longmapsto c_3$. The sub-region $r_1' \in$ *rest*(r) is formed from c_2: $r' =$ *region*$(l_2, \{c_2, c_3\})$. A second sub-region $r'' \in$ *rest*(r') is formed: $r'' =$ *region*$(l_3, \{c_3\})$. Applying T_1 to r'' results in $T_1(r'') = c_3$. Applying T_1 to r' results in $T_1(r') = (c_2; c_3)$. The result of $T_1(r)$ is therefore $T_1(r) = c_1; (c_2; c_3)$.

Consider region r_2 of Figure (4.4). The effect of applying T_1 to r_2 is to construct the sub-region beginning at command c_2, $r_2' =$ *region*$(l_2, \{c_2, c_3, c_4\})$. The set *rest*(r_2') is $\{$*unit*$(c_3),$ *unit*$(c_4)\}$ and the result of $T_1($*rest*$(r_2'))$ is the set $\{c_3, c_4\}$. The result of $T_1(r_2)$ is therefore $c_1; (c_2; \{c_3, c_4\})$.

The result of applying the transformation to a single loop is similar. Consider region r_1 of Figure (4.5). The application of T_1 to r_1 constructs the region beginning at c_2, $r_1' =$ *region*$(l_2, \{c_2, c_3\})$. The region r_1' does not include c_1, the head of region r_1 and the result of the transformation is the command $c_1; (c_2; c_3)$. □

Properties of the Path Transformation

The result of applying the path transformation T_1 to a single region r is a command c. This command describes the iteration-free behaviour of the region r but is not an abstraction of r. The command is equivalent to a maximal trace through r beginning with the head of r: $\mathcal{I}_c(T_1(r))(s,t) = \mathit{enabled}(r)(s) \wedge \mathit{mtrace}(\mathit{body}(r), s, t)$. The single region r, $\mathit{single?}(r)$ can contain any number of loops (all of which must begin with the head of r). The states produced by r can therefore result from the repeated iteration of loops through r. Since $T_1(r)$ is a command, it can be executed at most once to produce a state t from state s. To allow the repeated execution of $T_1(r)$, a unit region must be constructed, $\mathit{unit}(T_1(r))$. This allows the command to be executed any number of times to produce t from s: $\mathcal{I}_r(\mathit{unit}(T_1(r)))(s,t)$. This unit region, $\mathit{unit}(T_1(r))$, is the abstraction of r.

Theorem 4.13 *Path refines abstraction*

For any region r, which is a single loop, $\mathit{unit}(T_1(r))$ is an abstraction of r.

$$\frac{\mathit{single?}(r)}{\mathit{unit}(T_1(r)) \sqsubseteq r}$$

As well as being an abstraction of a single loop r, $\mathit{unit}(T_1(r))$ is semantically equivalent to r. If r is loop-free then executing r is equivalent to executing $T_1(r)$, no sequence of commands is executed more than once. If r contains a loop then the loop begins with the head of r and an execution of r is made up of one or more iterations of the loop. $T_1(r)$ is equivalent to a single iteration of the loop and $\mathit{unit}(T_1(r))$ is equivalent to the repeated iteration of the loop.

Theorem 4.14 *Path is equivalent to abstraction*

For region r and states s, t,

$$\frac{\mathit{single?}(r)}{\mathcal{I}_r(\mathit{unit}(T_1(r)))(s,t) = \mathcal{I}_r(r)(s,t)}$$

Both Theorem (4.13) and Theorem (4.14) are based on the property that any state produced by $\mathit{unit}(T_1(r))$ is produced by one or more maximal traces through region r: if $r' = \mathit{unit}(T_1(r))$ then $s \overset{r'}{\leadsto} t$ iff $\mathit{mtrace}^+(\mathit{body}(r), s, t)$. These two theorems establish the relationship between the regions when both terminate.

The path transformation constructs a proper abstraction of a region r by preserving the failures of r: region r halts in a state s in which it is enabled iff $\mathit{unit}(T_1(r))$ halts in s.

Theorem 4.15 *Path transformation preserves failure*

For region r and state s

$$\frac{\mathit{single?}(r) \quad \mathit{enabled}(r)(s)}{\mathit{halt?}(\mathit{unit}(T_1(r)))(s) \Leftrightarrow \mathit{halt?}(r)(s)}$$

Theorem (4.13) states that if the abstraction $\mathit{unit}(T_1(r))$ of region r establishes a property then so will r. Theorem (4.14) states the reverse: if r beginning in a state s produces a final state t in which some assertion is *true*, then so will $\mathit{unit}(T_1(r))$. A consequence of Theorem (4.13) and Theorem (4.14) is that a specification can be proved from $\mathcal{I}_r(r)(s,t)$ iff it can be proved from $\mathcal{I}_r(\mathit{unit}(T_1(r)))(s,t)$. Although such specifications describe a subset of the properties which may be proved of a region, they include the fact that a region terminates. Theorem (4.15) ensures that the abstraction is not based on Theorem (4.11): the transformed region fails in a state s iff the original region r also fails in s.

4.3.3 General Transformation of a Region

The general transformation, T_2, of an arbitrary region r breaks down r, which can contain any number of loops, to a set of sub-regions, $\mathit{loops}(r)$ each of which begins with a cut-point of r. The abstraction $T_2(r)$ is obtained by applying T_1 to each of the regions in $\mathit{loops}(r)$. The principal operations carried out by transformation T_2 find the cut-points of a region, by finding the loops in a region, and form a region from the result of applying the path transformation T_1.

Extracting Loops of a Region

The cut-points of a region r are the head of r and the commands which begin a loop in r. A command c of r begins a loop in r if either c is the head of r and $c \xmapsto{r} c$, or there is a path $a \subseteq \mathit{body}(r)$ from $\mathit{head}(r)$ to c ($\mathit{head}(r) \xmapsto{a} c$), a path $b \subseteq \mathit{body}(r)$ from c to c ($c \xmapsto{b} c$) and the only command common to both a and b is c ($a \cap b = \{c\}$).

Definition 4.22 *Loop heads*

For region r and command c, predicate $\mathit{lphead?}(c,r)$ is true iff c is the head of a loop in r.

$$\begin{array}{l}\mathit{lphead?} : (\mathcal{C} \times \mathcal{R}) \to \mathit{boolean} \\ \mathit{lphead?}(c,r) \stackrel{\text{def}}{=} \left\{\begin{array}{l} c = \mathit{head}(r) \wedge c \xmapsto{r} c \\ \vee(\exists a : a \subseteq \mathit{body}(r) \wedge \mathit{head}(r) \xmapsto{a} c \wedge c \xmapsto{b} c)\end{array}\right. \\ \text{where } b = (\mathit{body}(r) - a) \cup \{c\}\end{array}$$

For any region r, the set $\mathit{lpheads}(r)$ contains all commands beginning a loop in r.

$$\begin{array}{l}\mathit{lpheads} : \mathcal{R} \to \mathcal{P} \\ \mathit{lpheads}(r) \stackrel{\text{def}}{=} \{c \in r \mid \mathit{lphead?}(c,r)\}\end{array}$$

The cut-points of region r are the head of r and the commands which begin a loop in r.

$$\begin{array}{l}\mathit{cuts} : \mathcal{R} \to \mathcal{P} \\ \mathit{cuts}(r) \stackrel{\text{def}}{=} \mathit{lpheads}(r) + \mathit{head}(r)\end{array}$$

□

Note that *lpheads*(r) + *head*(r) is equivalent to *lpheads*$(r) \cup \{$*head*$(r)\}$: if *head*$(r) \notin$ *lpheads*(r) then no other command in *lpheads*(r) can share a label with *head*(r). The predicate *lphead?* identifies the commands beginning loops in r; these, together with the head of r, are the cut-points of the region (see Theorem D.6 of Appendix D). The method used here to find the commands beginning loops is more general than those used in program optimisation (Aho et al., 1986; Wolfe, 1991) since it can be applied to any region, regardless of the structure of the region's flow-graph.

When a cut-point c is identified, the body of the loop is made up of the commands which can be reached from c without passing through the head of the region or through any other cut-point.

Definition 4.23 *Loop bodies*

The body of a loop beginning at c in a region r is constructed from a cut-point c and the subset of r which excludes all other cut-points.

$$\begin{aligned}&\mathit{lpbody} : (\mathcal{C} \times \mathcal{R}) \to \mathcal{P}\\ &\mathit{lpbody}(c, r) \stackrel{\text{def}}{=} (\mathit{body}(r) - \mathit{lpheads}(r)) + c\end{aligned}$$

For region r, *loops*(r) is the set of sub-regions of r which begin with a cut-point of r and which exclude all other cut-points.

$$\begin{aligned}&\mathit{loops} : \mathcal{R} \to \mathit{FiniteSet}(\mathcal{R})\\ &\mathit{loops}(r) \stackrel{\text{def}}{=} \{\mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r)) \mid c \in \mathit{cuts}(r)\}\end{aligned}$$

□

The sub-regions constructed from the cut-points of r are single loops. If r' is a loop in r, $r' \in$ *loops*(r), then any sub-region of r' which contains a loop would also contain a cut-point c of r. If c was not the head of r' then, by the definition of *loops* and of *lpbody*, c could not be a member of r'.

Theorem 4.16 *For regions $r, r' \in \mathcal{R}$,*

$$\frac{r' \in \mathit{loops}(r)}{\mathit{single?}(r')}$$

The property of Theorem (4.16) ensures that the result of applying transformation T_1 to each $r_1 \in$ *loops*(r) is an abstraction of r_1. Transformation T_1 preserves the labels of the regions and the commands in *cuts*(r) share labels with commands in the set $T_1($*loops*$(r))$. The labels of the cut-points are preserved to ensure that any loop in r is also preserved by the abstractions constructed from $T_1($*loops*$(r))$.

The cut-points in *cuts*(r) are for use in the transformation of region r and are independent of any cut-points used for verification. If a proof begins with a program p and the property to be established is that eventually control reaches a command c, then c is a cut-point for the verification. If an abstraction p' is constructed from p, by constructing and transforming a region r, the command c may not appear in p' since it is not necessarily a cut-point for the transformation. However, the region can be constructed to exclude c and the abstraction p' will then include c.

Example 4.16 In the single loop r_3 of Figure (4.5), the only cut-point is the head of the region. Although there is a command other then *head*(r_3) which reaches itself through r_3, *head*(r_3) is necessary for each such loop. The only cut-point of r_3 is *head*(r_3), *cuts*$(r_3) = \{$*head*$(r_3)\}$, *lpbody*$($*head*$(r_3), r_3) =$ *body*(r_3) and *loops*$(r_3) = \{r_3\}$.

In a general loop, there may be one or more loops which are independent of the head of the region. In region r_1 of Figure (4.6), command c_2 is a cut-point of the region. There is a path $a = \{c_1, c_2\}$ such that *head*$(r_1) \xmapsto{a} c_2$, and a subset $p = \{c_2, c_3\}$ such that $c_2 \xmapsto{p} c_2$. The only command which occurs in both a and p is c_2. By contrast, command c_3 is not a cut-point. Although there is a subset $p' = \{c_3, c_2\}$ such that $c_3 \xmapsto{p} c_3$, there is no set a such that $a \cap p = \{c_3\}$ and *head*$(r_3) \xmapsto{a} c_3$. The cut-points of r_1 are $\{$*head*$(r_1), c_2\}$. The set of sub-regions *loops*(r) contains the region *unit*$($*head*$(r_1))$ and the region *region*$(l_2, \{c_2, c_3\})$ only.

In region r_3 of Figure (4.6), there are three cut-points, *cuts*$(r_3) = \{$*head*$(r_3), c_2, c_3\}$. For c_2, there are sets $a = \{c_1, c_2\}$ and $p = \{c_2, c_3, c_4\}$ such that *head*$(r_3) \xmapsto{a} c_2$, $c_2 \xmapsto{p} c_2$ and $a \cap p = \{c_2\}$. For c_3, the sets are $a = \{c_1, c_2, c_3\}$ and $p = \{c_3, c_4, c_5\}$. The loops in r_3 are *unit*$($*head*$(r_3))$, $(l_2, \{c_2, c_3\})$ and $(l_3, \{c_3, c_4, c_5\})$. Note that a path from c_2 to c_2 must pass through the region beginning with c_3. □

Transforming a Region

Once the cut-points of a region r are found and used to construct the sub-regions of *loops*(r), the general transformation T_2 constructs the abstraction of r. First the path transformation T_1 is applied to each sub-region in *loops*(r) to obtain a set of a commands from which a region is constructed. This region is the result of the transformation, $T_2(r)$.

Definition 4.24 *General transformation*

The body of the abstraction of a region r is constructed from the set *gtbody*(r) obtained by applying T_1 to the sub-regions of r beginning with a cut-point of r.

$$\begin{aligned} &\textit{gtbody} : \mathcal{R} \to \mathcal{P} \\ &\textit{gtbody}(r) \stackrel{\text{def}}{=} \{T_1(r_1) \mid r_1 \in \textit{loops}(r)\} \end{aligned}$$

The general transformation of a region r is the region beginning with the label of r and constructed from *gtbody*(r).

$$\begin{aligned} &T_2 : \mathcal{R} \to \mathcal{R} \\ &T_2(r) \stackrel{\text{def}}{=} \textit{region}(\textit{label}(r), \textit{gtbody}(r)) \end{aligned}$$

□

For any region r, the result of $T_2(r)$ is region in which each command is an abstraction of a sequence of commands from *head*(r) to a cut-point of r, from a cut-point to *head*(r) or from a cut-point to another cut-point. Each command in $T_2(r)$ is formed by applying T_1 to a region r' in *loops*(r) and, by Theorem (4.16), region r' is a single loop, *single?*(r'). The interpretation of a command $c \in r$, $\mathcal{I}_c(c)(s,t)$, is therefore equivalent to a maximal trace through r, beginning with a cut-point of r and excluding all other cut-points. This is described as the restricted maximal trace, *rmtrace*$($*lpheads*$(r))($*body*$(r), s, t)$. The properties of the general transformation are based on the relationship between the commands of $T_2(r)$ and the region r, described in terms of the restricted maximal traces. In particular, if $r' = T_2(r)$ and *enabled*$(r')(s)$ then $s \overset{r'}{\leadsto} t$ iff *rmtrace*$^+($*lpheads*$(r))($*body*$(r), s, t)$ (see Lemma D.42 and Lemma D.49 of Appendix D).

Transformation T_2 is a generalisation of the path transformation to arbitrary regions. The properties of T_2 are similar to those of T_1 but do not impose constraints on the region to be transformed. The most important property is that the general transformation T_2 constructs an abstraction of any region r.

Theorem 4.17 *Abstraction*

Transformation T_2 constructs an abstraction of any region r:

$$T_2(r) \sqsubseteq r$$

As with the path transformation, the result of applying T_2 to region r is a region which is semantically equivalent to r.

Theorem 4.18 *Equivalence*

For any region r and states s, t:

$$\mathcal{I}_r(T_2(r))(s,t) = \mathcal{I}_r(r)(s,t)$$

The transformation T_2 also preserves the failures of a region. Region r fails in a state s in which it is enabled iff $T_2(r)$ also fails in s.

Theorem 4.19 *General transformation preserves failure*

For any region r and state s,

$$\frac{\textit{enabled}(r)(s)}{\textit{halt?}(T_2(r))(s) \Leftrightarrow \textit{halt?}(r)(s)}$$

Transformation T_2 is a generalisation of T_1 and the result of applying the general transformation T_2 to a single region r is the region *unit*$(T_1(r))$.

Theorem 4.20 *Transformation of single regions*

For any region r,

$$\frac{single?(r)}{T_2(r) = unit(T_1(r))}$$

Theorem (4.17) and Theorem (4.18) allow the general transformation T_2 to be applied to any region r, the result will be both an abstraction of and equivalent to r. Theorem (4.19) ensures that the abstraction is not trivial, region $T_2(r)$ fails only when r fails. A consequence of Theorem (4.18) is that it is only necessary to apply T_2 to a region r; if r is is a single region then the result will be as if the path transformation had been applied.

Applying the General Transformation

The result of applying the general transformation T_2 to a region r of a program p will be used to abstract from p. Whether the verification of p is simplified by its abstraction therefore depends on the changes which can be made by the general transformation to the region r. There are practical limits to the effect of applying the transformation T_2 to a region; in some cases, applying T_2 to a region r will not make any changes, $T_2(r) = r$. For example, assume a command c of region r selects a successor by a label expression, which depends on some variable. Because c can reach any command of r, it is possible, in the worst case, to form a loop beginning with any command in r. Consequently, every command $c \in r$ will be a cut-point of r and no changes will be made to r, $T_2(r) = r$, since the cut-points of r are preserved by T_2.

This limitation is caused by the need to determine the control flow through a program from the syntax of the commands. For many processor languages, the majority of instructions will select a successor using a basic label (in *Labels*) or can be transformed to such a program, using Lemma (4.1). In such processor languages, an abstraction of a program can be obtained by constructing regions only from the commands which use a basic label to select a successor. Transforming these regions will, generally, reduce the number of commands. The commands which select a successor by a function on a name can then be considered separately, by a step in the proof to demonstrate that the command has some property. Alternatively, the commands can be abstracted individually (using Theorem 4.5) to form abstractions of p.

Processor languages in which each instruction selects a successor by a label expression (not in *Labels*) must be considered individually. Each such language will have an execution model which determines how an object program is executed. This model will generally support the definition of a *reaches* relation with which to determine the flow of control through a program (such relations are required for program optimisation). An alternative is to define transformations which form regions suitable for abstraction using T_2. For example, a transformation based on *constant propagation* (Aho et al., 1986) would attempt to calculate the labels of each command in a region, possibly using information obtained from the program specification. The result of such a transformation would be a region in which each command identified its successor by a constant label. Such a region would be suitable for abstraction by the general transformation.

Example 4.17 Assume commands $c_1, c_2, \ldots, c_5 \in \mathcal{C}$ such that $c_1 \longmapsto c_2 \longmapsto c_3 \longmapsto c_4 \longmapsto c_5$ and $c_4 \longmapsto c_2$. Also assume function $f \in \mathcal{F}_l$, name $x \in \mathcal{E}_n$, label $l \in$ *Labels* and command l : **goto** $f(x)$ such that $c_5 \longmapsto l$: **goto** $f(x)$. Let p be the program $\{c_1, c_2, c_3, c_4, c_5, l : \textbf{goto}\ f(x)\}$ and assume c_1 is the first command to be executed. Let r be the largest region of p beginning with command c.

$$r = (\mathit{label}(c_1), \{c_1, c_2, c_3, c_4, c_5, l : \textbf{goto}\ f(x)\})$$

Since l : **goto** $f(x) \longmapsto c$ for every command $c \in r$, every command in r is a cut-point of r. For example, there is a path a from the head of region, c_1 to c_3, $a = \{c_1, c_2, c_3\}$ and a path b from c_3 to c_3, $b = \{c_3, c_4, c_5, l : \textbf{goto}\ f(x)\}$ and the only command common to a and b is c_3. The transformation T_2 therefore constructs the set of unit regions:

$$\{\mathit{unit}(c_1), \mathit{unit}(c_2), \mathit{unit}(c_3), \mathit{unit}(c_4), \mathit{unit}(c_5), \mathit{unit}(l : \textbf{goto}\ f(x))\}$$

The application of T_1 to each unit region *unit*(c) is c and the result of the transformation is the region $r, T_2(r) = r$.

An alternative approach, which results in an abstraction p' of p, is to exclude the command l : **goto** $f(x)$ from the region. Let r be the region beginning with c_1 but excluding l : **goto** $f(x)$.

$$r = (\mathit{label}(c_1), \{c_1, c_2, c_3, c_4, c_5\})$$

The only command beginning a loop in r is c_2 and $T_2(r)$ applies the path transformation T_1 to the sub-regions:

$$r_1 = (\mathit{label}(c_1), \{c_1\}) \quad r_2 = (\mathit{label}(c_2), \{c_2, c_3, c_4, c_5\})$$

The transformation T_1 results in the commands $T_1(r_1) = c_1$ and $T_1(r_2) = c_2; c_3; c_4; c_5$. The abstraction p' is obtained by combining the program $\{T_1(r_1), T_1(r_1)\}$ with p. The command l : **goto** $f(x)$ can then be combined with any command of p'. Since $c_5 \longmapsto l$: **goto** $f(x)$ and c_5 occurs in r_2, the most useful abstraction is likely to result from $T_1(r_2); l$: **goto** $f(x)$. □

4.4 Proof Rules for Programs

The verification of a program of $\mathcal{L}$ is carried out in a program logic by applying proof rules to programs and commands. The logic which will be used here extends the assertion language $\mathcal{A}$ (defined in Chapter 3), with an operator for specifying the liveness properties of programs. This operator and its proof rules will allow a program to be verified using the method of intermittent assertions. It will also allow reasoning about the refinement relation between programs, to simplify the verification of a program. In particular, it will support the replacement of a program with its abstraction during a proof of correctness.

More expressive logics can be defined which also allow the safety properties of program to be considered. Examples of such logics include those of (Manna & Pnueli, 1981) and (Lamport, Programs: $$\frac{c \in p \quad \vdash p \Rightarrow \mathbf{wp}(c, Q)}{\vdash [P]p[Q]} \qquad \text{(tl6)}$$

Refinement: $$\frac{p_1 \sqsubseteq p_2 \quad \vdash [P]p_1[Q]}{\vdash [P]p_2[Q]} \qquad \text{(tl7)}$$

Transitivity: $$\frac{\vdash [P]p[R] \quad \vdash [R]p[Q]}{\vdash [P]p[Q]} \qquad \text{(tl8)}$$

Induction: $$\frac{i, j, n \in \mathbb{N} \quad \dfrac{j < i \quad \vdash [F(j)]p[Q]}{\vdash [F(i)]p[Q]}}{\vdash [F(n)]p[Q]} \qquad \text{(tl9)}$$

Weakening (programs): $$\frac{\vdash P \Rightarrow R \quad \vdash [R]p[Q]}{\vdash [P]p[Q]} \qquad \text{(tl10)}$$

Strengthening (programs) $$\frac{\vdash R \Rightarrow Q \quad \vdash [P]p[R]}{\vdash [P]p[Q]} \qquad \text{(tl11)}$$

where $c \in \mathcal{C}$, $p, p_1, p_2 \in \mathcal{P}$, $P, Q, R \in \mathcal{A}$ and $F \in (\mathbb{N} \to \mathcal{A})$

Figure 4.7: Proof Rules for the Programs

1994). Formulas for these logics specify the properties of a program behaviour (in *Behaviour*). However, reasoning about refinement in these logics is complicated (see Abadi & Lamport, 1991, or Lamport, 1994). The logic defined here will be sufficient to verify the liveness properties of sequential programs. The specification operator for programs will be based on the *leads-to* relation. The liveness properties established in the logic can therefore be established in a logic based on program behaviours (Theorem 4.2).

Specification of Programs

A program p is specified by an assertion (in $\mathcal{A}$), written $[P]p[Q]$ and constructed from precondition $P \in \mathcal{A}$ and postcondition $Q \in \mathcal{A}$. Each specification states that if the execution of program p begins in a state satisfying P then p will eventually produce a state satisfying Q.

Definition 4.25 *Program specification operator*

For assertions $P, Q \in \mathcal{A}$ and program $p \in \mathcal{P}$, the specification operator constructs a triple

$[P]p[Q]$ as an assertion on a state.

$$[\_]\_[\_] : (\mathcal{A} \times \mathcal{P} \times \mathcal{A}) \to \mathcal{A}$$
$$[P]p[Q] \stackrel{\text{def}}{=} \lambda(s : State) : P(s) \Rightarrow \exists(t : State) : s \stackrel{p}{\leadsto} t \wedge Q(t)$$

□

The program specification operator describes the total correctness of a program with respect to a specification. If $P \in \mathcal{A}$ is the precondition and $Q \in \mathcal{A}$ the postcondition of program p then the $\vdash [P]p[Q]$ asserts that p beginning in any state satisfying P will eventually establish Q.

The program logic for verifying programs of $\mathcal{L}$ is made up of the proof rules for commands (Figure 3.5 of Chapter 3) and the proof rules for programs of Figure (4.7). These proof rules are similar to rules defined by Francez (1992) for the intermittent assertions. Rule (tl6) describes the effect of a program command: if $P \Rightarrow wp(c, Q)$ then c will terminate in a state satisfying Q; therefore program p will establish Q. The refinement rule (tl7) states that any postcondition established by program p will also be established by a refinement p' of p and rule (tl8) is a restatement of the transitivity of *leads-to*. Rule (tl9) defines the induction scheme for program specifications. The proofs for the rules are straightforward from the definitions. The proof of the induction rule (tl9) is immediate from induction on the natural numbers. Proof rules for regions are not required since a region is primarily a means for constructing abstractions of a program. The refinement rule (tl7) is the main mechanism for simplifying a program: it allows the replacement of a program p with its abstraction p'. This abstraction can be constructed by abstracting from any subset of program p (Theorem 4.5).

4.4.1 Verifying Programs

A program p satisfies a specification made up of precondition $P \in \mathcal{A}$ and postcondition $Q \in \mathcal{A}$ if $\vdash [P]p[Q]$ is *true*. To verify the program, it must be shown that beginning in a state satisfying P, program p will eventually establish Q. Using the proof rules for programs, commands of the program are individually shown to establish assertions from which the program specification can be established. This is the reverse of the top-down approach used in the program logics for structured languages (Hoare, 1969; Dijkstra, 1976) in which the program specification is broken down into assertions to be established by commands.

The verification of a program using the proof rules of Figure (4.7) is based on the method of intermittent assertions (Manna, 1974; Burstall, 1974). This is a consequence of the definition of the specification operator, which requires that a program eventually establishes a postcondition. The transitivity rule (tl8) allows the specification of a program p, $\vdash [P]p[Q]$, to be established from a series of intermediate specifications: $\vdash [P]p[A_1], \ldots, \vdash [A_i]p[A_{i+1}], \ldots, \vdash [A_n]p[Q]$, where $P, Q, A_1, \ldots, A_n \in \mathcal{A}$. The rule for programs (tl6), allows each intermediate specification to be established from a sequence of commands: $\vdash A_i \Rightarrow \mathbf{wp}(c_1, B_1), \ldots, \vdash B_n \Rightarrow \mathbf{wp}(c_n, A_{i+1})$ where $c_1, \ldots, c_n \in p$ and $B_1, \ldots, B_n \in \mathcal{A}$. The choice of intermediate assertions, $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$, is determined by the proof method as are the commands $c_1, \ldots, c_n$ (which will normally be a sequence beginning with a program cut-point).

The refinement rule (tl8) allows the verification of a program to be simplified by abstracting from the program. The program can be abstracted at any point in the verification. For example, before beginning a proof, replacing $\vdash [P]p[Q]$ with $\vdash [P]p'[Q]$ where $p' \sqsubseteq p$, or when establishing an intermediate specification, replacing $\vdash [A_i]p[A_{i+1}]$ with $\vdash [A_i]p'[A_{i+1}]$. The abstraction of the program can be constructed either by applying sequential composition to individually selected program commands or by applying transformation T_1 or T_2 to a region of the program.

The approach to verifying programs which will be used here is based on constructing an abstraction of a program before beginning the proof of correctness. A program $p \in \mathcal{P}$ with precondition $P \in \mathcal{A}$ and postcondition $Q \in \mathcal{A}$ will be shown to satisfy the specification $P \in \mathcal{A}$, $\vdash [P]p[Q]$, in the following steps:

1. The first command of the program (which must be identified by precondition P) will be used to construct a region r of p. Region r will be made up of all commands in p which identify a successor by a basic label.
2. Transformation T_2 will be applied to r to construct the abstraction $T_2(r)$.
3. The abstraction p_1 of p will be formed by combining the region $T_2(r)$ with p: $p_1 = p \uplus \mathit{body}(T_2(r))$.
4. Sequential composition will be selectively applied to the commands of p_1, to abstract from commands which were excluded from region r (because of a successor expression depending on a variable). The result will be an abstraction p_2 of p. If there are no suitable commands in p_1 then $p_2 = p_1$.
5. Program p_2 will verified using the method of intermittent assertions: $\vdash [P]p_2[Q]$.

These steps simplify a program before verification based on the method of intermittent assertions. The correctness of the steps is therefore a consequence of the correctness of the proof method and the transformations used to abstract from a program. Proof tools to simplify the verification would automate the construction and transformation of a region (steps 1 and 2) and would simplify the result of sequential composition (step 4).

Example

For an example of the verification and abstraction of a program, assume $x, y, z \in \mathit{Names}$ are distinct and program $p \in \mathcal{P}$ has the following commands:

$$\begin{array}{l}
l_1 : x := 10, l_2 \\
l_2 : y := 0, l_3 \\
l_3 : \textbf{if } x =_a 0 \textbf{ then goto } l_6 \textbf{ else goto } l_4 \\
l_4 : x := x -_a 1, l_5 \\
l_5 : y := y +_a 1, l_3 \\
l_6 : \textbf{goto loc}(z)
\end{array}$$

Program p is intended to illustrate the abstraction of a program and does not carry out any useful function. The program will be shown to satisfy a simple specification which requires the program, beginning at the command labelled l_1, to eventually pass control to the label stored in variable z.

The precondition for the program, $\mathit{Pre}(l)$, states that execution begins at l_1 and the value of z is some label l. The postcondition, $\mathit{Post}(l)$, requires control to pass to the label stored in z.

$$\begin{array}{ll} \mathit{Pre} : \mathit{label} \to \mathcal{A} & \mathit{Post} : \mathit{label} \to \mathcal{A} \\ \mathit{Pre}(l) \stackrel{\text{def}}{=} z =_a l \wedge \mathit{pc} = l_1 & \mathit{Post}(l) \stackrel{\text{def}}{=} z =_a l \wedge \mathit{pc} =_a l \end{array}$$

The specification of the program is, $\vdash [\mathit{Pre}(l)]p[\mathit{Post}(l)]$ (for any $l \in \mathit{Labels}$). This requires the name z to have the same value when the program begins and ends. The commands of p labelled $l_1, \ldots, l_6$ will be referred to as $c_1, \ldots, c_6$ (i.e. $c_i = \mathit{at}(p, l_i)$).

The flow-graph of p is made up of a path from c_1 to label c_2, a loop, between c_3 and c_5, and a path from c_3 to c_6. First an abstraction p_1 of p is constructed from the region r beginning at c_1 and containing all commands except c_6 (which is a computed jump), $r = \mathit{region}(l_1, p - \{c_6\})$. The general transformation is applied to r; the cut-points of r are c_1 (the head of r) and c_3 (beginning the loop in r): $\mathit{cuts}(r) = \{c_1, c_3\}$. Two regions, r_1 and r_2, are constructed from r beginning at each cut-point and excluding the other cut-point, these form the loops of r:

$$\begin{array}{l} r_1 = \mathit{region}(l_1, \{c_1, c_2\}) \\ r_2 = \mathit{region}(l_3, \{c_3, c_4, c_5\}) \\ \mathit{loops}(r) = \{r_1, r_2\} \end{array}$$

The path transformation is applied to each region in $\mathit{loops}(r)$ to form the set of commands $\mathit{gtbody}(r)$, containing the abstractions of the single regions. The result of $T_2(r)$ is the region constructed from this set:

$$\begin{array}{l} \mathit{gtbody}(r) = \{T_1(r_1), T_1(r_2)\} \\ T_2(r) = \mathit{region}(l_1, \mathit{gtbody}(r)) \end{array}$$

Note that since composition preserves the reaches relation, $T_1(r_1) \longmapsto T_1(r_2)$, the body of $T_2(r)$ is $\mathit{gtbody}(r)$. The abstraction $p_1 \in \mathcal{P}$ of p is formed by combining the body of $T_2(r)$ with p: $p_1 = p \uplus \{T_1(r_1), T_1(r_2)\}$. By Theorem (4.12), this satisfies the refinement ordering $p_1 \sqsubseteq p$.

An abstraction p_2 of p_1 is then constructed by the sequential composition of $T_1(r_2)$ with command c_6, which was excluded from region r. Program p_2 is formed as $p_1 \uplus \{(T_1(r_2); c_6)\}$. This also satisfies the refinement ordering $p_2 \sqsubseteq p_1$ (and therefore $p_2 \sqsubseteq p$). The program p is verified by showing that p_2 is correct: $\vdash [\mathit{Pre}(l)]p_2[\mathit{Post}(l)]$, for any $l \in \mathit{Labels}$. Only two commands of p_2 are required in the proof of correctness, $T_1(r_1)$ and $(T_1(r_2); c_6)$. These will be referred to as C_1 and C_2, $C_1 = T_1(r_1)$ and $C_2 = T_1(r_2); c_6$. The two commands, after simplification, are:

$$\begin{array}{l} C_1 = l_1 : x, y := 10, 0, l_3 \\ C_2 = l_3 : \textbf{if } x =_a 0 \textbf{ then } x, y := x -_a 1, y +_a 1, \mathbf{loc}(z) \\ \qquad\qquad\qquad\quad \textbf{else } x, y := x -_a 1, y +_a 1, l_3 \end{array}$$

The simplifications can be carried out mechanically using the text of the commands.

The proof of correctness is straightforward and only the main steps will be described. Because command C_2 forms a loop, $C_2 \longmapsto C_2$, the proof requires induction on the value of x. This uses an invariant $\mathit{Inv}(v, l)$ for the loop, where v is the value of x and l the label stored in z:

$$\begin{aligned}&\mathit{Inv} : (\mathbb{N} \times \mathit{Labels}) \to \mathcal{A}\\ &\mathit{Inv}(v, l) \stackrel{\text{def}}{=} v = x \wedge l = z\end{aligned}$$

The steps of the proof are as follows, for any $l \in \mathit{Labels}$:

1. Precondition establishes invariant, $\vdash [\mathit{Pre}(l)]p_2[pc =_a l_3 \wedge \mathit{Inv}(10, l)]$.

   This can be established directly from C_1, with a proof of $\vdash (pc =_a l_1 \wedge \mathit{Pre}(l)) \Rightarrow \mathbf{wp}(C_1, pc =_a l_3 \wedge \mathit{Inv}(10, l))$. Note that C_1 is selected by the precondition $\mathit{Pre}(l)$ ($pc =_a l_1$) and ends in a state in which C_2 is enabled ($pc =_a l_3$).

2. Invariant establishes postcondition, $\vdash [pc =_a l_3 \wedge \mathit{Inv}(v, l)]p_2[pc =_a l]$ (for any $v \in \mathbb{N}$).

   The proof is by induction on v, the value of name x, using the induction rule (tl9). Since $pc =_a l_3$, command C_2 is selected. Furthermore, only command C_2 needs to be considered to establish the intermediate specification. There are two cases to consider, when $v = 0$ and when $v > 0$.

   (a) Base case, $\vdash [pc =_a l_3 \wedge \mathit{Inv}(0, l)]p_2[pc =_a \mathbf{loc}(z)]$.
   This can be established from the rule for programs (tl1) and from the truth of $\vdash pc =_a l_3 \wedge \mathit{Inv}(0, l) \Rightarrow \mathbf{wp}(C_2, pc =_a \mathbf{loc}(z))$.

   (b) Inductive case,$v > 0$ and $\vdash [pc =_a l_3 \wedge \mathit{Inv}(v, l)]p_2[pc =_a \mathbf{loc}(z)]$.
   This is established from the inductive hypothesis (see rule tl9) by establishing the specification $\vdash [pc =_a l_3 \wedge \mathit{Inv}(v, l)]p_2[pc =_a l_3 \wedge \mathit{Inv}(v - 1, l)]$. As with the base case, this can be established from the rule for programs (tl1) and the property of command C_2: $\vdash pc =_a l_3 \wedge \mathit{Inv}(v, l) \Rightarrow \mathbf{wp}(C_2, pc =_a l_3 \wedge \mathit{Inv}(v - 1, l))$.

These steps are used to show the correctness of program p_2 by combining the intermediate assertions. From $\vdash [\mathit{Pre}(l)]p_2[pc =_a l_3 \wedge \mathit{Inv}(10, l)]$ and $\vdash [pc =_a l_3 \wedge \mathit{Inv}(v, l)]p_2[pc =_a \mathbf{loc}(z)]$ (for any v), the transitivity rule (tl8) establishes $\vdash [\mathit{Pre}(l)]p_2[pc =_a \mathbf{loc}(z)]$. The refinement rule (tl7) and $p_2 \sqsubseteq p$ (and the definition of *Post*), can then be used to prove the correctness of p: $\vdash [\mathit{Pre}(l)]p[\mathit{Post}(l)]$.

4.5 Conclusion

Object code is more general than the programs of structured languages, which are usually considered in program verification. A structured program is a compound command (Loeckx & Sieber,

1987) and can be transformed or verified as a single command. To reason about object code, the program must be considered a collection of individual instructions, which can be executed in any order. This means that object code must be verified by reasoning about the individual instructions. For refinement and abstraction, it also means that all states produced by a program must be considered, not only the states in which a program begins and ends. To transform an object code program it is therefore necessary to consider the behaviour of the program instructions individually. This requires techniques more similar to those used in code optimisation to manipulate groups of instructions than those used in program verification and refinement to manipulate structured programs.

The language $\mathcal{L}$ provides a means for verifying and abstracting the object code programs of arbitrary processors. Since any processor instruction can be modelled by a command of $\mathcal{L}$, any object code program (of any processor language) can be modelled as a program of $\mathcal{L}$. This allows object code programs to be verified in a single program logic, defined for the language $\mathcal{L}$. Because an instruction can be modelled by a single $\mathcal{L}$ command, the difficulty of verifying object code is not increased by the translation to a program of $\mathcal{L}$. This approach considers object code as a program, which can be verified using the methods of Floyd (1967) and Burstall (1974). It differs significantly from other approaches to verifying object code which require that the program instructions are embedded in a structured program (Back et al., 1994) or which consider instructions as data for a processor simulation (Yuan Yu, 1992; Boyer & Moore, 1997). Both these approaches complicate reasoning about and transforming object code programs since either the data or the execution model of object code is simulated in terms of a less expressive language.

Abstraction is intended to reduce the number of program commands which must be considered during verification. Two methods were described for the abstraction of $\mathcal{L}$ programs. The first applies sequential composition to manually chosen program commands. This method can be applied to any program of $\mathcal{L}$ (and therefore any object code program) but is difficult to mechanise. The second method is based on applying program transformations which analyse the program flow-graph to identify the sequences of commands to be abstracted by sequential composition. This approach is similar to techniques used in code optimisation. However, the methods used to define and reason about code optimising transformations cannot be used to define abstracting transformations. Instead, a framework for transforming programs of $\mathcal{L}$ was developed based on regions of a program. This included methods for reasoning about the syntactic and semantic properties of regions, allowing a transformation to be defined and shown to abstract from regions. This framework was used to define and reason about transformations T_1 and T_2 which abstract from regions.

The path transformation T_1 is a straightforward application of sequential composition to sequences of commands in a region. The general transformation T_2 is more complicated since it must analyse the flow-graph of the region. The analysis used is more general than those of code optimisation: it is not restricted by the structure of the regions flow-graph. The result of applying T_2 to a region r is an abstraction of r which preserves the liveness properties of r: any specification which can be established by r can be established by its abstraction. Abstracting a program p by abstracting a region of the program therefore results in an abstraction with the liveness properties of p. Because the abstraction of a program of $\mathcal{L}$ is also a program of $\mathcal{L}$, the methods for abstraction can be repeatedly applied. This is used in the verification method described in Section 4.4. It is not possible using other approaches to abstraction, such as symbolic execution (King, 1971), which require that a program and its abstraction are treated separately. Constructing abstractions as $\mathcal{L}$ programs allows the same methods for verification and abstraction to be applied to a program and its abstraction.

The main contribution of this chapter is the ability to model object code as programs of $\mathcal{L}$ and to abstract and verify these programs. The use of techniques based on the flow-graph of a program to abstract from the program is new for verification. Since the requirements of the transformations are the same as for code optimisation (principally requiring the ability to analyse the flow-graph of a program), mechanising the transformations is straightforward. Furthermore, since the methods for abstraction can be applied to any program of $\mathcal{L}$, they can be used to abstract the object code programs of any processor language. This allows any object code program to be verified in a program logic and also allows the manual work needed to verify the program to be reduced by applying automated tools to simplify the program.

Chapter 5

Examples: Processor Languages

There are two approaches to considering object code verification. The first is in terms of the processor language: the problem is to develop methods for reasoning about the instructions and programs of a processor language. The second is in terms of the object code as a program, without considering the processor language: given methods for reasoning about programs of a processor language, the problem is to exploit particular features of object code to simplify their verification. The main emphasis of this thesis is on the problem of reasoning about programs of processor languages since this is needed to treat object code as a program. This chapter will consider the verification of object code for particular processor languages, by considering the translation from a processor language to the language $\mathcal{L}$. The object code programs considered in this chapter will be verified, in terms of their model in $\mathcal{L}$, using the method described in Section 4.4. The approach used in this thesis to simplify verification is based on program abstraction, which does not exploit features of object code. The treatment of object code as programs with particular features will be considered in the next chapter.

A processor architecture defines the instructions which can occur in an object code program and also determines the data operations and the variables which may be used by an instruction. To model an object code program as a program of $\mathcal{L}$, the language $\mathcal{L}$ must be able to describe the behaviour of each processor instruction. This, in turn, requires the ability to describe the data operations of the processor as expressions of $\mathcal{L}$. Examples of the data operations and instructions of different processors will be considered and their definition in terms of $\mathcal{L}$ described. This will be based on a general data model which includes the expressions of $\mathcal{L}$ needed to model common data operations of processors. As an example of the use of this data model, a program of $\mathcal{L}$ to implement division will be defined and verified. The use made by this program of the general data model, defined in $\mathcal{L}$, reflects the use made by object code of the data operations of a processor.

A processor language is defined by the processor architecture which follows either a complex instruction set (CISC) design or a reduced instruction set (RISC) design (Hennessy & Patterson, 1990). As an example of the use of the language $\mathcal{L}$ to model processor instructions, two processor architectures will be considered: the Motorola 68000 (Motorola, 1986), a CISC processor, and the PowerPC (Motorola Inc. and IBM Corp., 1997), a RISC processor. These two processors are of interest since their instructions are representative of the instructions found in many processor languages. If the language $\mathcal{L}$ can model arbitrary instructions then it should be able to model the instructions of the Motorola 68000 and the PowerPC processors. To illustrate the use of $\mathcal{L}$ for object code verification, programs will be defined and verified for both the Motorola 68000 and the PowerPC processors. The difficulty of modelling object code in the language $\mathcal{L}$ is determined by the complexity of the instructions and the data operations. These are influenced by the design of the processor architecture and can lead to a processor language having features which are difficult to model in $\mathcal{L}$. Some of these features and their model in terms of $\mathcal{L}$ will be described.

This chapter is structured as follows: **Section 5.1** describes the model, in $\mathcal{L}$, of data items and operations commonly used in processor languages. A program of $\mathcal{L}$ using this model to implement the division of natural numbers, will be defined and verified. This will be followed by the description of two processor languages, in terms of $\mathcal{L}$. The first, in **Section 5.2** is the language of the Motorola 68000 processor. This includes the verification of two object code programs, one of which is the Motorola 68000 implementation of the program for division. **Section 5.3** describes the PowerPC processor language. This also includes the verification of two object code programs, one of which is the PowerPC implementation of the division program. Some features of processor languages which complicate the model of object code programs will be described in **Section 5.4**.

The program logic defined in Chapter 4 is the basis for all verification proofs described in this chapter. The processor language semantics defined in this chapter are intended as examples of the use of the abstract language $\mathcal{L}$. The processor functions and instructions defined are generally chosen for their use in the example programs and the description of the processor languages is not intended to be complete.

5.1 General Data Model

A processor represents and manipulates data items as *bit-vectors* (Hayes, 1988). A *bit* is an element of the set $\{0, 1\}$. A bit-vector of size n is a finite sequence of n bits and can be considered a function of type $(\{x : \mathbb{N} \mid x < n\} \rightarrow \{0, 1\})$; the ith bit of bit-vector b is $b(i)$. A bit-vector b can be interpreted as natural number (in $\mathbb{N}$) by assigning to each bit a weight relative to its position in the sequence. The bit in position i of b has weight 2^i and the interpretation of the bit-vector is $\sum_{i<n} b(i) \times 2^i$. A bit-vector of size n can represent and be represented by a natural number in the range $0, \ldots, 2^n - 1$. This allows the data items of a processor language to be modelled by the set of natural numbers, $\mathbb{N}$.

The *least significant bit* of a bit-vector of size n is at position 0 and the *most significant bit* is at position $n - 1$. Three sizes of bit-vectors will be assumed: a *byte* is a bit-vector of size 8; a *word* is a bit-vector of size 16 and a *long-word* (or simply *long*) is a bit-vector of size 32. The terms used by the semantics of processor languages to refer to bit-vectors of size 16 and 32 vary. Where there is ambiguity, the term used by the processor semantics will also be given. Constants

Byte, *Word* and *Long* will identify the size of the byte, word and long-word bit-vectors:

$$\mathit{Byte} \stackrel{\text{def}}{=} 8 \quad \mathit{Word} \stackrel{\text{def}}{=} 16 \quad \mathit{Long} \stackrel{\text{def}}{=} 32$$

The model in $\mathcal{L}$ of a processor's data operations will be based on natural numbers, as in the examples of Chapter 3. The data model of Figure (3.1) will be assumed; in particular, the set of values and variables will be $\mathbb{N}$: $\mathit{Values} = \mathbb{N}$ and $\mathit{Vars} = \mathbb{N}$. The set of registers *Regs* will depend on the processor language but will include the program counter of $\mathcal{L}$: $pc \in \mathit{Regs}$. The value functions of Figure (3.1) as well as the label and name functions **loc** and **ref** will also be used. The expressions defined in Chapter 3 and used here include the arithmetic operators $x +_a y$, $x -_a y$, $x \times_a y$, $x \bmod_a y$ and x^y and the comparison relations $x =_a y$, $x <_a y$ and $x >_a y$.

5.1.1 Data Operations

The data operations provided by a processor manipulate and compare bit-vectors or implement arithmetic functions. The data operations are modelled by expressions of $\mathcal{L}$, defined in terms of arithmetic operations on natural numbers and functions manipulating bit-vectors. The functions of $\mathcal{L}$ (in addition to those of Chapter 3) needed to model the data operations are defined in Section A.1 of the appendix. Where an expression models an operation on a data type common in processor languages, the name of the data type may be used. For example, the term *bit-vector* may be used rather than the term *value expression*.

Bit-Vector Operations

To model processor operations on bit-vectors, the expressions of $\mathcal{L}$ must allow the manipulation of bits and bit-vectors. This can be defined in terms of accessor and constructor functions. For any $i \in \mathit{Values}$ and $a \in \mathcal{E}$, the expression $\mathbf{bit}(i)(a)$ is an accessor function which results in the ith bit of bit-vector a. The least significant bit of bit-vector a has index 0, $\mathbf{bit}(0)(a)$; the most significant bit of a long-word bit-vector a has index 31, $\mathbf{bit}(31)(a)$. Bit constructor **mkBit** applied to argument a results in 0 iff a is equivalent to **false** and is 1 otherwise.

Long-words, words and bytes are constructed from elements of the set *Values*: a byte is constructed from the application of **Byte**, a word is constructed by **Word** and a long-word by **Long**. These functions have type $(\mathcal{E} \to \mathcal{E})$ and definition:

$$\mathbf{Byte}(a) \stackrel{\text{def}}{=} a \bmod_a 2^8 \quad \mathbf{Word}(a) \stackrel{\text{def}}{=} a \bmod_a 2^{16} \quad \mathbf{Long}(a) \stackrel{\text{def}}{=} a \bmod_a 2^{32}$$

Let $a, b, c, d \in \mathcal{E}$. Expression $\mathbf{mkWord}(a, b) \in \mathcal{E}$ constructs a word in which the *most significant byte* is $\mathbf{Byte}(a)$ and the *least significant byte is* $\mathbf{Byte}(b)$. Expression $\mathbf{mkLong}(a, b, c, d) \in \mathcal{E}$ constructs a long-word in which the *most significant word* is $\mathbf{mkWord}(a, b)$ and the *least significant word* is $\mathbf{mkWord}(c, d)$.

$$\mathbf{mkWord}(a, b) \stackrel{\text{def}}{=} (\mathbf{Byte}(a) \times_a 2^8) +_a \mathbf{Byte}(b)$$
$$\mathbf{mkLong}(a, b, c, d) \stackrel{\text{def}}{=} (\mathbf{mkWord}(a, b) \times_a 2^{16}) +_a \mathbf{mkWord}(c, d)$$

Accessor functions extract bytes and words from bit-vectors. Function **B** applied to value i and a bit-vector a results in the ith byte of a. The least significant byte of bit-vector a is $\mathbf{B}(0)(a)$ and the most significant byte of a long-word a is $\mathbf{B}(3)(a)$. The accessor function **W** applied to value i and bit-vector a results in the ith word of a. The least significant word of a is $\mathbf{W}(0)(a)$ and the most significant word of a long-word a is $\mathbf{W}(1)(a)$.

$$\begin{array}{ll} \mathbf{B}(3)(\mathbf{mkLong}(a,b,c,d)) \equiv a & \mathbf{B}(0)(\mathbf{mkWord}(a,b)) \equiv b \\ \mathbf{W}(1)(\mathbf{mkLong}(a,b,c,d)) \equiv \mathbf{mkWord}(a,b) & \mathbf{W}(0)(\mathbf{Byte}(a)) \equiv \mathbf{Byte}(a) \end{array}$$

Bit-Vectors of size 32 are the largest that will be considered therefore accessor functions are not needed for long-words.

Processor languages provide *rotate* and *shift* operations to alter the position of the bits in a bit-vector (Wakerly, 1989). A rotate or shift left operator applied to bit-vector a results in a bit-vector b such that every bit of a in position i is equal to the bit of b in position $i+1$. A rotate or shift right operator applied to bit-vector a results in a bit-vector b such that every bit of a in position i is equal to the bit of b in position $i-1$. Shift and rotate operations differ in the treatment of the least and most significant bits of the argument. The rotate operations store the least or most significant bits of the arguments, typically in the result. The shift operations do not preserve these bits. Expressions of $\mathcal{L}$ for common shift and rotate operations on bit-vectors are described in Section A.1.3 of the appendix.

Arithmetic Operations

Arithmetic functions of processor languages are defined for bit-vectors of a given size. The general form of an arithmetic expression (of $\mathcal{E}$) on a bit-vector will be $f(sz)(a_1, \ldots, a_n)$ where f is the function name, $sz \in \mathit{Values}$ the size of the bit-vectors and $a_1, \ldots, a_n \in \mathcal{E}$ the arguments to the function. The result of applying the arithmetic functions can be represented in a bit-vector of size *sz*. An arithmetic function on bit-vectors interprets its arguments either as integers, the function is said to be *signed*, or as naturals, where the function is *unsigned*.

Integers are commonly represented in a processor language using the two's complement system (see Hayes, 1988 or Wakerly, 1989). A bit-vector a of size n is interpreted as an integer by assigning to the most significant bit, $a(n-1)$, the weight -2^{n-1}. The integer represented by the bit-vector a is given by $(-2^{n-1} \times b(n-1)) + \sum_{i<n-1} b(i) \times 2^i$ and the integers which can be represented by a bit-vector of size n are $\{x : \mathbb{Z} \mid -2^{n-1} \leq x \leq 2^{n-1}-1\}$. Note that since the bit-vectors are represented as the natural numbers, $\mathit{Values} = \mathbb{N}$, the $\mathcal{L}$ model described here represents an integer by an interpretation of a natural number.

The addition, subtraction and multiplication of bit-vectors x, y of size *sz* will be written $x +_{sz} y$, $x -_{sz} y$ and $x \times_{sz} y$ respectively. For example, the addition of bit-vector of size 32 (long-words) will be written $x +_{32} y$. The operations are defined in Section A.1 of the appendix and generalise the operations of Figure (3.2) to bit-vectors of a given size *sz*. However, addition and subtraction are defined using two's complement arithmetic for both signed and unsigned data. Signed addition has the same definition as unsigned addition; subtraction is signed and is defined by addition and the negation of an argument ($x - y = x + (-y)$, for $x, y \in \mathbb{Z}$).

The comparison of bit-vectors also takes the size of the bit-vectors into account. The equality between bit-vectors of size *sz* will be written $x =_{sz} y$ and the less-than relation is written $x <_{sz} y$. Other comparison functions can be defined in terms of the equality and the less-than operators. In particular, $x >_{sz} y$ will be written for the greater-than relation between bit-vectors of size *sz*. These relations have type $(\mathcal{E} \times \mathit{Values} \times \mathcal{E}) \rightarrow \mathcal{E}_b$ and definition:

$$\begin{aligned} x =_{sz} y &\stackrel{\text{def}}{=} (x \bmod 2^{sz}) =_a (y \bmod 2^{sz}) \\ x <_{sz} y &\stackrel{\text{def}}{=} (x \bmod 2^{sz}) <_a (y \bmod 2^{sz}) \\ x >_{sz} y &\stackrel{\text{def}}{=} y <_{sz} x \end{aligned}$$

The size of the bit-vector by which an integer is represented can be increased by *sign extension*. Bit-vector a of size n is extended to a bit-vector of size $m \geq n$ by assigning to the bits in positions n to m the value $\mathbf{bit}(a)(n -_a 1)$ (the sign of a). The interpretation as an integer of the resulting bit-vector is that of bit-vector a. This operation is modelled by the expression $\mathbf{ext}(n, m, a)$ which sign-extends bit-vector a of size n to size m. When the interpretation of a as an integer x is not negative, $x \geq 0$, $\mathbf{ext}(n, m, a) \equiv a$.

5.1.2 Memory Operations

Processor languages define addressing modes which determine how a program may access the machine memory. The memory model depend on the semantics of the processor but typically a processor stores a single byte in each location in memory; words and long-words are stored in two or four consecutive locations respectively and each memory location is identified by a long-word. The name functions used here will range over the memory variables *Vars* and are base on the name function **ref**, defined in Chapter 3. When describing processor instructions, the function **ref** will also be used as a generic description of a memory operation.

The data items stored in a number of memory locations can be combined to construct a single bit-vector. Assuming that each memory location stores a single byte, a long-word can be read from memory address $x \in \mathit{Vars}$ by accessing the four consecutive locations beginning with x. Value function **readl** applies the name function **ref** and the constructor **mkLong** to the arguments $a, a+1, a+2, a+3$. Function **readl** has type $\mathcal{E} \rightarrow \mathcal{E}$ and definition:

$$\mathbf{readl}(a) \stackrel{\text{def}}{=} \mathbf{mkLong}(\mathbf{ref}(a), \mathbf{ref}(a +_{32} 1), \mathbf{ref}(a +_{32} 2)\mathbf{ref}(a +_{32} 3))$$

To store a long-word in the memory locations beginning at address a, each of which can store a single byte, the long-word must be broken down to four bytes. The function **writel** applied to expressions $a, e \in \mathcal{E}$, constructs an assignment list in which the locations between address a and $a +_{32} 3$ are assigned byte 3 to byte 0 of long-word e.

$$\begin{aligned} \mathbf{writel}(a, e) = \ & (\mathbf{ref}(a), \mathbf{B}(3)(e)) \cdot (\mathbf{ref}(a +_{32} 1), \mathbf{B}(2)(e)) \cdot \\ & (\mathbf{ref}(a +_{32} 2), \mathbf{B}(1)(e)) \cdot (\mathbf{ref}(a +_{32} 3), \mathbf{B}(0)(e)) \end{aligned}$$

```
unsigned int div (n, d, r)
unsigned int n, d;
unsigned int *r;
{
    unsigned int c=0;
    while (n>d)
    {
        c=c+1;
        n=n-d;
    }
    *r=n;
    return c
}
```

Figure 5.1: Division: C Program

For any $a, e \in \mathcal{E}$, the memory variables updated by **writel**(a, e) are used to calculate the value of **readl**(a). Substituting **writel**(a, e) for **readl**(a) will always result in the expression e written to address a, $\textbf{readl}(a) \lhd (\textbf{writel}(a, e)) \equiv e$. Any substitution in **writel** can be applied directly to its arguments, $\textbf{writel}(a, e) \lhd (x, v) \equiv \textbf{writel}(a \lhd (x, v), e \lhd (x, v))$.

5.1.3 Example: Division of Natural Numbers

As an example of the use of data operations in object code, a program for the division of natural numbers will be defined and its verification described. The program is derived from the object code implementing the program of the language C (Kernighan & Ritchie, 1978), given in Figure (5.1). The C program is a function with parameters n, d and r which calculates $n \div d$ and stores n mod d in the variable identified by pointer r.

The program of Figure (5.1) was compiled to produce an object code program for the Alpha AXP processor (Digital Equipment Corporation, 1996). This was simplified (by removing unnecessary assignments) to the Alpha AXP program of Figure (5.2). The $\mathcal{L}$ program *idiv* of Figure (5.2) was derived from this object code program and assumes the set of registers satisfies $\{\textbf{r0}, \textbf{r1}, \ldots, \textbf{r30}\} \subseteq \textit{Regs}$. The $\mathcal{L}$ program *idiv* will be used as an example of the use of data operations and the verification of programs. It is not intended to be an accurate model of the Alpha AXP program. However, the only significant difference between a model of the Alpha AXP program and program *idiv* is in the command labelled l_{10}. The Alpha AXP stores long-words in four consecutive memory locations: an accurate model would have the command $:= (\textbf{writel}(\textbf{r2}, \textbf{r18}), l_{11})$ at label l_{10}.

The $\mathcal{L}$ program *idiv* begins at the command labelled l_1, with argument n stored in register **r16**, argument d stored in register **r17** and argument r in register **r18**. The label to which control

div..ng:	bis \$31, \$31, \$0	l_1 :	$\mathbf{r0} := 0, l_2$
	zapnot \$16, 15, \$2	l_2 :	$\mathbf{r2} := \mathbf{r16}, l_3$
	zapnot \$17, 15, \$3	l_3 :	$\mathbf{r3} := \mathbf{r17}, l_4$
	cmplue \$2, \$17, \$1	l_4 :	**if r2** $<_{32}$ **r17 then r1** $:= 1, l_5$ **else r1** $:= 0, l_5$
	bne \$1, \$35	l_5 :	**if not r1** $=_{32}$ 0 **then goto** l_6 **else goto** l_{10}
\$36:	addl \$0, 1, \$0	l_6 :	$\mathbf{r0} := \mathbf{r0} +_{32} 1, l_6$
	subl \$2, \$17, \$2	l_7 :	$\mathbf{r2} := \mathbf{r2} -_{32} \mathbf{r17}, l_8$
	cmpule \$2,\$3, \$1	l_8 :	**if r2** $<_{32}$ **r3 then r1** $:= 1, l_9$ **else r1** $:= 0, l_9$
	beq \$1, \$36	l_9 :	**if r1** $=_{32}$ 0 **then goto** l_6 **else goto** l_{10}
\$35:	stl \$2, 0(\$18)	l_{10} :	$\mathbf{ref}(\mathbf{r18}) := \mathbf{r2}, l_{11}$
	ret \$31, (\$26)	l_{11} :	**goto loc**$(\mathbf{r26})$

Alphа AXP Program $\mathcal{L}$ Program *idiv*

Figure 5.2: Division: $\mathcal{L}$ Program *idiv*

is to return, at the end of the program, is stored in register **r26**. The program begins by assigning 0 to register **r0** (which implements the C variable c) and the contents of registers **r16** and **r17** to registers **r2** and **r3** respectively. If **r2** is less than **r17** ($n < d$), control is passed to the instruction labelled l_6 otherwise control passes to the instruction labelled l_{10}.

The loop implementing the *while* statement of the C program begins at label l_6. Register **r0** is incremented by 1 ($c = c+1$); **r2** is decremented by the value of **r17** ($n = n - d$) and compared with **r17**. If **r2** is not less than **r3** ($n < d$), control passes to label l_6, beginning another iteration of the loop. If **r2** is less than **r17**, control passes to the instruction labelled l_{10}. This stores the value of **r2** in the memory location identified by **r18**, implementing the assignment $*r = n$. The program then ends, passing control to the instruction identified by register **r26**. The result of the program (the C variable c) is stored in register **r0**.

Specification of *idiv*

The specification of program *idiv* requires that for any natural number n and d, $d > 0$, the program terminates with the quotient assigned to register **r0** and the remainder stored in the memory location a, identified by register **r18**. When the program terminates, control must pass to the label l stored in register **r26**. For $n, d, a, l \in \mathbb{N}$, the precondition of program *idiv* is the assertion $\mathit{Pre}(n, d, a, l) \in \mathcal{A}$ and the postcondition is the assertion $\mathit{Post}(n, d, a, l) \in \mathcal{A}$:

$$\mathit{Pre}(n, d, a, l) \stackrel{\text{def}}{=} d >_a 0 \wedge n \geq_a 0 \wedge \mathbf{r16} =_{32} n \wedge \mathbf{r17} =_{32} d \wedge \mathbf{r18} =_{32} a \wedge \mathbf{r26} =_{32} l$$
$$\mathit{Post}(n, d, a, l) \stackrel{\text{def}}{=} n \geq_a 0 \wedge d >_a 0 \wedge \mathbf{r26} =_{32} l \wedge \mathbf{r18} =_{32} a \wedge n =_{32} (\mathbf{r0} \times_{32} d) +_{32} \mathbf{ref}(\mathbf{r18})$$

The program begins when the command labelled l_1 is selected for execution in a state satisfying $\mathit{Pre}(n, d, a, l)$. Eventually the program *idiv* must establish $\mathit{Post}(n, d, a, l)$ and pass control to the command labelled l:

$$\vdash [pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l)]\mathit{idiv}[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)] \quad (\text{for } n, d, a, l \in \mathbb{N})$$

Verification of *idiv*

Program *idiv* is verified by showing that it satisfies intermediate specifications at the program cut-points. The intermediate specifications are then used to establish the program specification. Since there is a loop at label l_6, the cut-points of the program are labelled l_1 (the first command), l_6 and the command labelled l (at which execution must end). The property of the loop in the program, at l_6, is verified by induction on the value of **r16**. The invariant for the loop (Floyd, 1967; Burstall, 1974) specifies the values of the quotient and the remainder at each iteration of the loop. The invariant, $\mathit{Inv}(q, n, d, a, l) \in \mathcal{A}$, is defined:

$$\mathit{Inv}(q, n, d, a, l) \stackrel{\text{def}}{=} \begin{cases} n \geq_a 0 \wedge d >_a 0 \wedge q =_{32} \mathbf{r16} \\ \wedge l =_{32} \mathbf{r26} \wedge a =_{32} \mathbf{r18} \wedge d =_{32} \mathbf{r3} \wedge d =_{32} \mathbf{r17} \\ \wedge(\mathbf{r0} \times_{32} \mathbf{r17}) +_{32} \mathbf{r16}) =_{32} n \\ \wedge(pc =_{32} l \Rightarrow \mathbf{ref}(\mathbf{r18}) =_{32} \mathbf{r16}) \end{cases}$$

When the program terminates (with $pc =_{32} l$), the loop invariant, *Inv*, establishes, for $q \in \mathbb{N}$, the postcondition of the program: $\vdash pc =_{32} l \wedge q =_{32} (n \bmod d) \wedge \mathit{Inv}(q, n, d, a, l) \Rightarrow \mathit{Post}(n, d, a, l)$.

The commands of *idiv* labelled $l_1, \ldots, l_{11}$ will be referred to as $c_1, \ldots, c_{11}$. Given the precondition, the postcondition and the loop invariant, the verification of *idiv* is in three steps. The first and second to show that either command c_1 establishes the postcondition or c_1 establishes the invariant in a state in which c_6 is selected for execution. The third to show, by induction, that command c_6 establishes the loop invariant and that the loop terminates.

1. Precondition establishes postcondition:

$$\vdash [pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d]\mathit{idiv}[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)]$$

2. Precondition establishes invariant:

$$\vdash [pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l) \wedge \neg n <_{32} d]\mathit{idiv}[pc =_{32} l_6 \wedge \mathit{Inv}(n, n, d, a, l)]$$

3. Invariant establishes postcondition, the proof is by induction on q (rule tl9):

$$\vdash [pc =_{32} l_6 \wedge \mathit{Inv}(q, n, d, a, l)]\mathit{idiv}[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)] \quad \text{for any } q \in \mathbb{N}$$

(a) Base case, $\mathbf{r16} -_{32} \mathbf{r17} <_{32} \mathbf{r3}$:

$$\begin{array}{l}\vdash [pc =_{32} l_6 \wedge \mathit{Inv}(q, n, d, a, l \wedge \mathbf{r16} -_{32} \mathbf{r17}) <_{32} \mathbf{r3}] \\ \quad \mathit{idiv}[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)]\end{array}$$

(b) Inductive case, $\neg(\mathbf{r16} -_{32} \mathbf{r17} <_{32} \mathbf{r3})$:

$$\begin{array}{l}\vdash [pc =_{32} l_6 \wedge \mathit{Inv}(q, n, d, a, l) \wedge \neg \mathbf{r16} -_{32} \mathbf{r17}) <_{32} \mathbf{r3}] \\ \quad \mathit{idiv}[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)]\end{array}$$

Using the approach to verifying programs described in Chapter 4, program *idiv* is verified by constructing an abstraction of *idiv*. This reduces the number of commands which must be considered in the proof of correctness. To verify program *idiv* directly, without constructing an abstraction, complicates the proof since each of the commands in the program must be considered individually. For example, to establish the postcondition from the precondition (Step 1), each of the following command specifications must be established:

$$\begin{array}{l}
\vdash (pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d) \\
\quad \Rightarrow \mathbf{wp}(c_1, pc =_{32} l_2 \wedge \mathit{Pre}(n, d, a, l) \wedge \mathbf{r0} =_{32} 0) \\
\vdash (pc =_{32} l_2 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \wedge \mathbf{r0} =_{32} 0) \\
\quad \Rightarrow \mathbf{wp}(c_2, pc =_{32} l_3 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \wedge \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16}) \\
\vdash (pc =_{32} l_3 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \wedge \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16}) \\
\quad \Rightarrow \mathbf{wp}(c_3, pc =_{32} l_4 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \\
\qquad\qquad \wedge\, \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17}) \\
\vdash (pc =_{32} l_4 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \wedge \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17}) \\
\quad \Rightarrow \mathbf{wp}(c_4, pc =_{32} l_5 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \\
\qquad\qquad \wedge\, \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17} \wedge \mathbf{r1} =_{32} 0) \\
\vdash (pc =_{32} l_5 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \\
\quad \wedge\, \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17} \wedge \mathbf{r1} =_{32} 0) \\
\quad \Rightarrow \mathbf{wp}(c_5, pc =_{32} l_{10} \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \\
\qquad\qquad \wedge\, \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17} \wedge \mathbf{r1} =_{32} 0) \\
\vdash (pc =_{32} l_{10} \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \\
\quad \wedge\, \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17} \wedge \mathbf{r1} =_{32} 0) \\
\quad \Rightarrow \mathbf{wp}(c_{10}, pc =_{32} l_{11} \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \\
\quad \wedge\, \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17} \wedge \mathbf{r1} =_{32} 0 \wedge \mathbf{ref}(\mathbf{r18}) =_{32} \mathbf{r2}) \\
\vdash (pc =_{32} l_{11} \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \\
\quad \wedge\, \mathbf{r0} =_{32} 0 \wedge \mathbf{r2} =_{32} \mathbf{r16} \wedge \mathbf{r3} =_{32} \mathbf{r17} \wedge \mathbf{r1} =_{32} 0 \wedge \mathbf{ref}(\mathbf{r18}) =_{32} \mathbf{r2}) \\
\quad \Rightarrow \mathbf{wp}(c_{11}, pc =_{32} l \wedge \mathit{Post}(n, d, a, l)
\end{array}$$

Each of the remaining steps in the proof would also require commands to be considered separately. To establish the invariant from precondition (Step 2) requires reasoning about commands $c_1, c_2, c_3, c_4, c_5, c_{10}, c_{11}$. To establish the postcondition from the invariant (Step 3), the commands which must be considered in both cases are $c_6, c_7, c_8, c_9, c_{10}, c_{11}$. Although the proof that each command establishes it specification is straightforward, the number of commands which must

$$
\begin{aligned}
C_1 = l_1 :& \textbf{if } \mathbf{r16} <_{32} \mathbf{r17} \\
& \textbf{then } \mathbf{r0}, \mathbf{r1}, \mathbf{r2}, \mathbf{r3}, \mathbf{ref}(\mathbf{r18}) := 0, 1, \mathbf{r16}, \mathbf{r17}, \mathbf{r16}, \mathbf{loc}(\mathbf{r26}) \\
& \textbf{else } \mathbf{r0}, \mathbf{r1}, \mathbf{r2}, \mathbf{r3} := 0, 0, \mathbf{r16}, \mathbf{r17}, l_5
\end{aligned}
$$

$$
\begin{aligned}
C_2 = l_6 :& \textbf{if not } \mathbf{r2} -_{32} \mathbf{r17} <_{32} \mathbf{r3} \\
& \textbf{then } \mathbf{r0}, \mathbf{r1}, \mathbf{r2} := \mathbf{r0} +_{32} 1, 0, \mathbf{r2} -_{32} \mathbf{r17}, l_6 \\
& \textbf{else } \mathbf{r0}, \mathbf{r1}, \mathbf{r2}, \mathbf{ref}(\mathbf{r18}) := \mathbf{r0} +_{32} 1, 1, \mathbf{r2} -_{32} \mathbf{r17}, \mathbf{r2} -_{32} \mathbf{r17}, \mathbf{loc}(\mathbf{r26})
\end{aligned}
$$

Figure 5.3: Commands of Abstraction idiv_2

be considered and the detail required makes the verification of the program complicated. Constructing an abstraction of the program simplifies the verification by describing the operations performed by a group of commands as a single command, reducing the detail which must be considered.

Abstraction of *idiv*

An abstraction of the program *idiv* is obtained by constructing and transforming regions of the program. The abstraction of *idiv* begins with the construction of a region r beginning with c_1 and containing all commands with a constant successor expression. Since only c_{11} is a computed jump (to the label stored in **r26**), region r contains all commands except c_{11}: $r = \mathit{region}(l_1, \mathit{idiv} - \{c_{11}\})$.

An abstraction of region r is obtained by applying transformation T_2. The cut-points of r are commands c_1 (the head of r) and c_6 (from the path $c_6 \longmapsto c_7 \longmapsto c_8 \longmapsto c_9$). These are used to construct regions r_1 and r_2:

$$
\begin{aligned}
r_1 &= \mathit{region}(l_1, \{c_1, c_2, c_3, c_4, c_5, c_{10}\}) \\
r_2 &= \mathit{region}(l_6, \{c_6, c_7, c_8, c_9, c_{10}\})
\end{aligned}
$$

Both r_1 and r_2 are abstracted by the application of T_1 to construct the set of commands $\mathit{gtbody}(r)$:

$$
\begin{aligned}
T_1(r_1) &= (c_1; c_2; c_3; c_4; c_5; c_{10}) \\
T_1(r_2) &= (c_6; c_7; c_8; c_9; c_{10}) \\
\mathit{gtbody}(r) &= \{T_1(r_1), T_1(r_2)\}
\end{aligned}
$$

The result of abstracting region r is $T_2(r) = \mathit{region}(l_1, \mathit{gtbody}(r))$, this contains the two commands $T_1(r_1)$ and $T_1(r_2)$. The first abstraction idiv_1 of program *idiv* is obtained by combining the commands of $T_2(r)$ with *idiv*: $\mathit{idiv}_1 = \mathit{idiv} \uplus \{T_1(r_1), T_1(r_2)\}$. This satisfies the ordering $\mathit{idiv}_1 \sqsubseteq \mathit{idiv}$.

To verify *idiv* it is only necessary to verify *idiv*$_1$, of which only three commands are required: $T_1(r_1)$, $T_1(r_2)$ and c_{11}. A second abstraction *idiv*$_2$ of *idiv* can be obtained by composing $T_1(r_1)$ and $T_1(r_2)$ with c_{11}: *idiv*$_2$ $=$ *idiv*$_1$ $\uplus \{(T_1(r_1); c_{11}), (T_1(r_2); c_{11})\}$. The commands $(T_1(r_1); c_{11})$ and $(T_1(r_2); c_{11})$ will be referred to as C_1 and C_2 respectively, $C_1, C_2 \in$ *idiv*$_2$. The two commands, after simplification, are given in Figure (5.3). Program *idiv*$_2$ satisfies the ordering *idiv*$_2$ $\sqsubseteq$ *idiv*$_1$. By the refinement rule (tl7), to verify *idiv*, it is enough to verify *idiv*$_2$.

Verification of *idiv*$_2$

The verification of program *idiv*$_2$ is carried out using the same steps as for program *idiv*. However, the only commands of *idiv*$_2$ which must be considered are C_1 and C_2. To establish the postcondition from the precondition (Step 1), it is enough to show that command C_1 satisfies the specification:

$$\vdash (pc =_{32} l_1 \wedge Pre(n, d, a, l) \wedge n_{32} d) \Rightarrow \mathbf{wp}(C_1, pc =_{32} l \wedge Post(n, d, a, l))$$

By the rule for programs (tl6), this establishes $\vdash [pc =_{32} l_1 \wedge Pre(n, d, a, l) \wedge n <_{32} d]\, idiv_2\, [pc =_{32} l \wedge Post(n, d, a, l)]$. From *idiv*$_2$ $\sqsubseteq$ *idiv* and the refinement rule (tl7), this establishes $\vdash [pc =_{32} l_1 \wedge Pre(n, d, a, l) \wedge n <_{32} d]\, idiv\, [pc =_{32} l \wedge Post(n, d, a, l)]$. The remaining steps are similar. To show that the precondition establishes the invariant (Step 2), only command C_1 must be considered. To show that the invariant establishes the postcondition (Step 3), only command C_2 is required.

The proof of Step (1) is representative of the way in which a proof is carried out using the proof rules for programs and for the commands. The proof for Step (1) will be given here; proofs for the other steps are similar.

Step (1): *Precondition establishes postcondition.*

$$\vdash [pc =_{32} l_1 \wedge Pre(n, d, a, l) \wedge n <_{32} d]\, idiv_2\, [pc =_{32} l \wedge Post(n, d, a, l)]$$

Since $pc =_{32} l_1$, command c_1 of program *idiv*$_2$ is selected and, by rule (tl6), the assertion to prove is that the assumptions satisfy $\mathbf{wp}(C_1, pc =_{32} l \wedge Post(n, d, a, l))$. From rule (tl3), the fact that c_1 is a conditional command and the assumptions $n <_{32} d$ and $\mathbf{r16} =_{32} n \wedge \mathbf{r17} =_{32} d$ (definition of *Pre*), it follows that the precondition must satisfy:

$$\begin{array}{ll}\vdash & (pc =_{32} l_1 \wedge Pre(n, d, a, l) \wedge n <_{32} d) \\ & \Rightarrow \mathbf{wp}((\mathbf{r0}, \mathbf{r1}, \mathbf{r2}, \mathbf{r3}, \mathbf{ref}(\mathbf{r18}) := 0, 1, \mathbf{r17}, \mathbf{r17}, \mathbf{r16}, \mathbf{r26}), pc =_{32} l \wedge Post(n, d, a, l))\end{array}$$

The assumptions can be weakened (rule tl5) with the postcondition updated by the assignments of C_1. To apply the weakening rule requires a proof that the precondition establishes the updated postcondition:

$$\begin{array}{ll}\vdash & (pc =_{32} l_1 \wedge Pre(n, d, a, l) \wedge n <_{32} d) \\ & \Rightarrow \\ & (pc =_{32} l \wedge Post(n, d, a, l)) \lhd (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{r17}) \\ & \qquad \cdot (\mathbf{r3}, \mathbf{r17}) \cdot (\mathbf{ref}(\mathbf{r18}), \mathbf{r16})\end{array} \tag{5.1}$$

That this is *true* can be shown from the definitions and by substitution:

$$
\begin{array}{lll}
\vdash & pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d & \text{(assumptions)} \\
\vdash & \mathit{Pre}(n, d, a, l) & \\
& \quad \Rightarrow n \geq_a 0 \wedge d >_a 0 \wedge \mathbf{r26} =_{32} l \wedge \mathbf{r18} =_{32} a & \text{(definition } \mathit{Pre}\text{)} \\
& \qquad \wedge\, \mathbf{r16} =_{32} n \wedge \mathbf{r17} =_{32} n & \\
\vdash & \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \Rightarrow n =_{32} (0 \times_{32} d) +_{32} n & (n <_{32} d)) \\
\vdash & \mathit{Pre}(n, d, a, l) \wedge n <_{32} d \Rightarrow n =_{32} (0 \times_{32} d) +_{32} \mathbf{r16} & (\mathbf{r16} =_{32} n) \\
\vdash & \mathit{Pre}(n, d, a, l) \wedge n <_{32} d & \\
& \Rightarrow\ (n =_{32} (\mathbf{r0} \times_{32} d) +_{32} \mathbf{ref}(\mathbf{r18})) & \text{(substitution)} \\
& \qquad\qquad \lhd (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{r17}) & \\
& \qquad\qquad\quad \cdot (\mathbf{r3}, \mathbf{r17}) \cdot (\mathbf{ref}(\mathbf{r18}), \mathbf{r16}) & \\
\vdash & \mathit{Pre}(n, d, a, l) & \\
& \Rightarrow\ (pc =_{32} l) \lhd (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{r17}) & \text{(substitution and } l =_{32} \mathbf{r26}\text{)} \\
& \qquad\qquad\qquad \cdot (\mathbf{r3}, \mathbf{r17}) \cdot (\mathbf{ref}(\mathbf{r18}), \mathbf{r16}) & \\
\vdash & \mathit{Pre}(n, d, a, l) \wedge n <_{32} d & \\
& \Rightarrow\ (pc =_{32} l \wedge \mathit{Post}(n, d, a, l)) \lhd (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) & \text{(definition of } \mathit{Post}\text{)} \\
& \qquad\qquad\qquad \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{r17}) & \\
& \qquad\qquad\qquad \cdot (\mathbf{r3}, \mathbf{r17}) \cdot (\mathbf{ref}(\mathbf{r18}), \mathbf{r16}) &
\end{array}
$$

By rule (tl1), Assertion (5.1) establishes the weakest precondition of the assignment. It follows that the precondition establishes the postcondition when $\mathbf{r3} <_{32} \mathbf{r17}$. This is enough to prove the intermediate specification of Step (1).

Proof of Program *idiv*

The correctness of program idiv_2 is established by combining the intermediate specifications of each of the steps. Assume $n, d, a, l \in \mathbb{N}$. When $n < d$, Step (1) establishes:

$$\vdash [pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l) \wedge n <_{32} d]\mathit{idiv}_2[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)]$$

When $n >_a d$ the proof is by the transitivity rule (tl8), the Steps (2) and (3) establish

$$\vdash [pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l) \wedge \neg n <_{32} d]\mathit{idiv}_2[pc =_{32} l_6 \wedge \mathit{Inv}(n, d, a, l)]$$
$$\vdash [pc =_{32} l_6 \wedge \mathit{Inv}(n, d, a, l)]\mathit{idiv}_2[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)]$$

The correctness of idiv_2 follows, by cases of $n <_{32} d$ and transitivity (rule tl8), establishing:

$$\vdash [pc =_{32} l_1 \wedge \mathit{Pre}(n, d, a, l)]\mathit{idiv}_2[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)]$$

By the refinement rule (tl7), this also establishes the correctness of *idiv*:

$$\vdash [pc =_{32} l \wedge \mathit{Pre}(n, d, a, l)]\mathit{idiv}[pc =_{32} l \wedge \mathit{Post}(n, d, a, l)]$$

□

Verifying $idiv_2$, rather than program *idiv*, reduces the size of the proof by reducing the number of commands which needed to be considered. The main differences between programs *idiv* and $idiv_2$ are the actions carried out by the commands. In program *idiv*, each command models a processor instruction and therefore performs a simple action. In program $idiv_2$, commands C_1 and C_2 perform actions which are equivalent to the behaviour of a group of commands. Verifying program $idiv_2$ is therefore simplified by constructing and reasoning about a small number of commands, each of which performs a more complex action than the individual instructions.

5.2 The Motorola 68000 Architecture

Two processor languages will be considered, as examples of the ability to describe processor instructions in terms of $\mathcal{L}$. The first processor language is that of the Motorola 68000 (which will be referred to as the M68000). The M68000 is a 32 bit microprocessor with a large number of instructions each of which can perform a number of operations on data (Motorola, 1986). Descriptions of the M68000 are given by Ford & Topp (1988) and Wakerly (1989). A formal definition of the semantics of a large subset of the M68000 instructions is given by Yuan Yu (1992). The values which can be represented by the M68000 are the natural numbers up to 2^{32}. Each location in memory is identified by a value and each location stores a single byte. An instruction can be stored in one or more consecutive locations. The memory locations are addressed by the values (up to 2^{32}).

The M68000 has 16 general purpose registers organised as 8 *data registers* and 8 *address registers*. The data registers are named **D0**, ... , **D7** and store the arguments to, and results of, data operations. The address registers are named **A0**, ... , **A7** and store the addresses in memory of data or instructions.

$$\{\mathbf{D0}, \ldots, \mathbf{D7}, \mathbf{A0}, \ldots, \mathbf{A7}, \mathbf{PC}, \mathbf{SR}\} \subset Regs$$

The program counter of the processor, **PC**, identifies the currently executing instruction and a status register, **SR**, is assigned a value to reflect the result of an operation. The program counter of $\mathcal{L}$ can be considered synonymous with that of the M68000, $pc = \mathbf{PC}$. All registers store long-words although a processor instruction can also manipulate a register as a byte or a word.

Address register **A7** is used as a stack pointer, storing the address of the top of the stack. The stack grows from the top to the bottom of memory: if x is the address of the top of the stack and y the address of the second item on the stack then $x < y$. The address register **A6** is typically used by programs as a *frame pointer*, holding the address of a section of local memory to be used by a routine of the program.

5.2.1 Instructions

An instruction of the M68000 implements one or more operations, each of which is the result of applying some function to one or more arguments. The general form of an instruction is

Syntax	$\mathcal{L}$ expression	Description
$\#x$	x	Immediate
x	$\mathbf{ref}(x)$	Absolute addressing
Dn	**Dn**	Data register direct
An	**An**	Address register direct
An@	$\mathbf{ref}(\mathbf{An})$	Address register indirect
An@+	$\mathbf{ref}(\mathbf{An})$	Address register with post-increment ($\mathbf{An} := \mathbf{An} +_{32} sz$)
An@−	$\mathbf{ref}(\mathbf{An} -_{32} sz)$	Address register with pre-decrement ($\mathbf{An} := \mathbf{An} -_{32} sz$)
$\mathbf{An}@(x)$	$\mathbf{ref}(\mathbf{An} +_{32} x)$	Address register indirect with displacement
$\mathbf{An}@(d_1, \mathbf{r} : sz, sc)$	$\mathbf{ref}(\mathbf{An} +_{32} \mathbf{ext}(sz, Long, d_1) +_{32} \mathbf{ext}(sz, Long, \mathbf{r}) \times_{32} sc)$	Address register indirect with index
$\mathbf{An}@(d_1)@(d_2, \mathbf{r} : sz, sc)$	$\mathbf{ref}(\mathbf{readl}(\mathbf{An} +_{32} \mathbf{ext}(sz, Long, d_1)) +_{32} (\mathbf{ext}(sz, Long, \mathbf{r}) \times_{32} sc) +_{32} \mathbf{ext}(sz, Long, d_2))$	Memory indirect post-indexed
$\mathbf{An}@(d_1, \mathbf{r} : sz, sc)@(d_2)$	$\mathbf{ref}(\ \mathbf{readl}(\mathbf{An} +_{32} \mathbf{ext}(sz, Long, d_1) +_{32} \mathbf{ext}(sz, Long, \mathbf{r})) +_{32} \mathbf{ext}(sz, Long, d_2))$	Memory indirect pre-indexed

where $x \in Values$, displacements $d_1, d_2 \in Values$, size $sz \in \{Long, Word, Byte\}$ and scale $sc \in Values$.

Figure 5.4: Addressing Modes

`inst`.*sz src*, *dst*. The instruction identifier is `inst`, *sz* is the size of the operation, *src* is a source argument and *dst* is a destination argument. The instruction arguments are used to calculate the source and destination operands to the operation implemented by the instruction. Instructions operate on data of size byte, word or long-word and an instruction is said to be of size byte, word or long-word. The size *sz* of an instruction is represented as `b`, `w` or `l` for operations on bytes, words and longs respectively. An instruction can *read from* or *write to* the memory variables and the registers and the size of the instruction determines the size of the bit-vector which is read or written.

Addressing Modes

Figure (5.4) describes the addressing modes for instructions of the M68000; a full description of the addressing modes is given in Section B.1.2 of the appendix. Each addressing mode is determined by the syntax of the instruction arguments; the corresponding $\mathcal{L}$ expression (in Figure 5.4) describes the general form of the definition in $\mathcal{L}$. As well as accessing memory locations, the *address register with post-increment* and *address register with pre-decrement* modes also update operands. In the post-increment mode, the register **An** is incremented by the size of the instruction after the operation defined by the instruction has been completed. In the pre-decrement mode, the register **An** is decremented by the size of the instruction before the operation defined by the instruction begins.

An instruction argument using an addressing mode which is interpreted as a name expression of $\mathcal{L}$ can appear as either a source or destination argument. For example, the *absolute* and the *data register direct* modes are defined by name expressions (Figure 5.4). An argument which is interpreted only as a value expression can occur only as a source argument, e.g. the *immediate* addressing mode is a value expression only.

Interpretation of Results

The status register, **SR**, stores information about the result of operations. The five least significant bits of register **SR** are flags indicating *condition codes* and are set according to functions on the arguments to and the result of an operation. The rules used to calculate the condition codes depend on the particular instruction. There are five condition codes: x, the *extend* flag, n, the *negative* flag, z, the *zero*, v, the *overflow* and c, the *carry* flag. The extend flag is general purpose: it is used by rotate operations to extend a bit-vector and is set by arithmetic operations to correspond to the carry flag. The negative flag is set when the result of an operation, interpreted as a two's complement number, is less than 0 and the zero flag is set when the result of the operation is equal to 0. When the result of the operation is too large to be represented, the overflow flag is set. The carry flag is set by the arithmetic operations to reflect the carry out of an addition; other operations clear the carry (set the flag to 0).

A function **mkSR** is defined in Section B.1 of the appendix and constructs a bit-vector of five bits. The status register can be updated with the result of an operation by the assignment **SR** $:=$ **mkSR**(x, n, z, v, c), where x, n, z, v, c are the extend, negative, zero, overflow and carry flags respectively. The status register **SR** is used by testing the individual bits; e.g. to determine the value of the zero flag z, the test is the expression **bit**$(2)($**SR**$)$.

5.2.2 Operations

The instructions of the M68000 implement operations to move data between names, to perform arithmetic or comparison operations on data and to perform operations on the flow of control. Some instructions have specialised forms which carry out the operation with greater speed or which use less memory than the more general form. Instructions may also assign values to the status register **SR** by setting the condition codes to reflect the result of an operation. An example of how instructions calculate the condition codes is given in Section B.1.1 of the appendix.

Data Movement

Data movement instructions store the value of the source operand in the name given by the destination operand. These instructions may also update the status register to the result of testing the source operand, e.g. whether it is zero or negative. The M68000 has a general instruction for moving data between variables as well as more specialised instructions.

The general *move* instruction, written `move.`*sz src*, *dst*, assigns to the destination *dst*, the value of size *sz* obtained from the source argument *src*. If the size *sz* is less than a long-word then only the byte or word of the destination is altered. e.g. A move of size byte to a register will alter only the first byte of the register: `move.b` $\#1$, **D0** sets the first byte of the data register **D0** to 1. This can be modelled by the $\mathcal{L}$ command:

$$\mathbf{D0} := \mathbf{mkLong}(\mathbf{B}3(\mathbf{D0}), \mathbf{B}2(\mathbf{D0}), \mathbf{B}1(\mathbf{D0}), 1)$$

Note that the successor expression for the instruction is simply $pc +_{32} 4$, selecting the next instruction in memory. To simplify the presentation, the successor expression for the $\mathcal{L}$ commands modelling instructions will not be given where the expression simply increments the program counter to the next instruction in memory.

The *load effective address* instruction, written `lea` *src*, **An**, is a specialised data movement instruction. The destination operand is an address register which is assigned the address obtained from the source argument. This can be modelled as the $\mathcal{L}$ command: **An** $:=$ *src*. The *link and allocate* instruction, `link` **An**, $\#x$, is also a specialised instruction, intended to implement the function call of a high-level language. The value x, interpreted as an integer, is added to the stack pointer, **A7**, with the effect of allocating area on the stack for use by the function. The instruction pushes the value of the address register **An** onto the stack, stores the address of the top of the stack in **An** and adds x to the stack pointer **A7**.

$$\mathbf{A7}, \mathbf{An}, \mathbf{ref}(\mathbf{A7} -_{32} 4) := (\mathbf{A7} -_{32} 4) +_{32} x, \mathbf{A7} -_{32} 4, \mathbf{An} \quad (\text{when } \mathbf{An} \neq \mathbf{A7})$$

The *unlink and de-allocate memory* instruction, `unlk` **An**, is the reverse of the `link` instruction. The stack pointer, **A7**, is assigned the value of **An** and the address register **An** is assigned the value at the top of the stack.

$$\mathbf{A7}, \mathbf{An} := \mathbf{An} +_{32} 4, \mathbf{readl}(\mathbf{An}) \quad (\text{when } \mathbf{An} \neq \mathbf{A7})$$

Arithmetic Instructions

The arithmetic instructions provide addition, subtraction, multiplication and division on signed and unsigned bit-vectors of size byte, word and long-word. The arithmetic instructions have a common semantics in which only the operation (of addition, subtraction, etc) varies.

In the *addition* instruction, `add`.sz src, dst, the destination operand is assigned the value of $src +_{sz} dst$. When the destination is an address register, the status register is unchanged. When the destination operand is not an address register, the status register is assigned a value calculated from an interpretation f.

$$dst, \mathbf{SR} := src +_{sz} dst, f(src +_{sz} dst)$$

The *comparison* instruction, `cmp`.sz src_1, src_2, subtracts the first source operand src_1 from the second source operand src_2 and sets the status register to reflect the result. Comparison instructions do not retain the result of the subtraction and the only changes made are to the value of the status register. When the instruction is of size *Long*, the comparison can be calculated by the following function **calcSR**:

$$\mathbf{calcSR} : (\mathcal{E} \times \mathcal{E}) \to \mathcal{E}$$

$$\mathbf{calcSR}(x, y) \stackrel{\text{def}}{=} \begin{cases} \mathbf{mkSR}(\mathbf{bit}(4)(\mathbf{SR}), \mathbf{bit}(31)(x -_{32} y), (x -_{32} y) =_{32} 0, \\ \qquad (\mathbf{bit}(31)(y) =_a \mathbf{bit}(31)(x -_{32} y) \\ \qquad\ \ \mathbf{and\ not\ bit}(31)(x) =_a \mathbf{bit}(31)(y)), \\ \qquad ((\mathbf{bit}(31)(y)\ \mathbf{or\ bit}(31)(x -_{32} y))\ \mathbf{and\ not\ bit}(31)(x)) \\ \qquad\ \ \mathbf{or}\ (\mathbf{bit}(31)(y)\ \mathbf{and\ bit}(31)(x -_{32} y))) \end{cases}$$

The result of a comparison between src_1 and src_2, when $sz = Long$, is to assign the result of $\mathbf{calcSR}(src_2, src_1)$ to the status register, $\mathbf{SR} := \mathbf{calcSR}(src_2, src_1)$.

Program Control

The flow of control through a program is controlled by assigning values to the program counter **PC**. This is an operation performed by *jump* and *branch* instructions. The M68000 also includes instructions which support the implementation of sub-routines. The address to which control is to be returned is stored at the top of the stack before control passes to the sub-routine. When the sub-routine terminates, the value at the top of the stack is interpreted as the label to which control returns.

The *jump* instruction, `jmp` src, unconditionally passes control to the instruction at location src, this is the $\mathcal{L}$ command **goto** src. The *branch* instruction, `B`cc src, passes control to the instruction at address src if a test on the status register succeeds. The condition code cc determines the test to be performed and the individual bits of the status register which are to be tested. If the test fails, control is passed to the next instruction in memory. For example, when the condition is `CS` the test is that the carry flag is set (bit 0 of **SR** is 1), assume the next instruction is at label l:

$$\mathbf{if\ bit}(0)(\mathbf{SR})\ \mathbf{then\ goto}\ src\ \mathbf{else\ goto}\ l$$

A *jump to sub-routine* instruction, `jsr` src, stores the address l of the next instruction in memory on the top of the system stack and control is then passed to label src. The stack pointer, register **A7**, is decremented by the size of an address (4 bytes).

$$\mathbf{ref}(\mathbf{A7} -_{32} 4), \mathbf{A7}, \mathbf{PC} := l, \mathbf{A7} -_{32} 4, src$$

```
sum(unsigned int n, int *a, int *b)
{
    unsigned int i=1;
    b[0]=a[0];
    while(i!=n)
    {
       b[i]=a[i]+b[i-1];
       i=i+1;
    }
}
```

Figure 5.5: Summation: C Program

The *return from sub-routine*, `rts`, ends a sub-routine by passing control to the label at the top of the stack. The stack pointer is incremented by the size of an address.

$$\mathbf{A7}, \mathbf{PC} := \mathbf{A7} +_{32} 4, \mathbf{ref}(\mathbf{A7})$$

Summary

The M68000 processor language is based on a small number of data and address registers together with specialised registers **PC** and **SR** to select instructions and perform tests. The M68000 instructions can make use of a large number of addressing modes to access memory variables. The majority of the variables in a program of the M68000 will therefore be memory variables rather than registers. The M68000 also provides implicit support for a stack and, through the use of frame pointers, for the implementation of high-level functions. The instructions of the M68000 are relatively straightforward and their description in terms of commands of $\mathcal{L}$ does not pose any great difficulty. The greatest complexity in modelling M68000 instructions is therefore in the model of the data operations of the M68000 as expressions of $\mathcal{L}$. The basic requirement of the language $\mathcal{L}$ is that it can be used to model both the data operations and the instructions of the M68000. This requirement is satisfied, allowing the object code programs of the M68000 to be modelled and verified as programs of $\mathcal{L}$.

5.2.3 Example: Summation

The C program of Figure (5.5) stores a running total of the sum of elements of arrays a and b in array a. In the C language, an array is represented as an address in memory. The arguments to function *sum* are the number of integers n, the address of the first element of the array of integers, a, and the address of the first element of the array in which the results are to be stored, b. A local

```
_sum:
        link a6, #0                   ; push A6 onto stack, A6:=A7
        movel d2, sp@-                ; push D2 onto stack
        movel a6@(8), d1              ; D1 is n
        movel a6@(12), a1             ; A1 is a
        movel a6@(16), a0             ; A0 is b
        moveq #1, d0                  ; D0 is i, D0:=1
        movel a1@, a0@                ; b[0]:=a[0]
        cmpl d0, d1                   ; compare i against n
        jeq L3                        ; if n=i then goto L3
L4:
        movel a1@(d0:l:4), d2         ; D2 := a[i]
        addl a0@(-4,d0:l:4), d2       ; D2 := D2+b[i-1]
        movel d2, a0@(d0:l:4)         ; b[i]:= D2
        addql #1, d0                  ; i=i+1
        cmpl d0, d1                   ; compare i against n
        jne L4                        ; if n≠i then goto L4
L3:
        movel a6@(-4), d2             ; pop D2 from stack
        unlk a6                       ; A7:=A6, pop A6 from stack
        rts                           ; return from routine
```

Figure 5.6: Summation: M68000 Program

variable i is used as a counter and is initially set to 1. The first element of array b is assigned the integer stored in the first element of array a, $b(0) := a(0)$. While $i \neq n$ (the C expression i!=n), the ith element of b is assigned the sum of the $i - 1$th element of b and ith element of a. The function completes execution when $i = n$.

Object Code Program

The C program of Figure (5.5) was compiled with the GNU compiler to produce the object code program of Figure (5.6). The assembly language program includes synthetic instructions `jeq` and `jne` which are aliases for M68000 instructions. For the program *sum*, these are equivalent to the branch on condition instruction, `bcc`. The instruction `jne` branches to a label if the result of the last comparison was an inequality. Instruction `jeq` branches to a label if the result of the last comparison was an equality. The syntax of the instructions in the program of Figure (5.6) also combines the operation size with the instruction name. For example, `movel` is the `move` instruction on operands of size *Long*.

$$
\begin{array}{ll}
l_1 : & := (\mathbf{A6}, \mathbf{A7} -_{32} 4) \cdot (\mathbf{A7}, \mathbf{A7} -_{32} 4) \cdot \mathbf{writel}(\mathbf{A7} -_{32} 4, \mathbf{A6}), l_2 \\
l_2 : & := (\mathbf{A7}, \mathbf{A7} -_{32} 4) \cdot \mathbf{writel}(\mathbf{A7} -_{32} 4, \mathbf{D2}), l_3 \\
l_3 : & \mathbf{D1} := \mathbf{readl}(\mathbf{A6} +_{32} 8), l_4 \\
l_4 : & \mathbf{A1} := \mathbf{readl}(\mathbf{A6} +_{32} 12), l_5 \\
l_5 : & \mathbf{A0} := \mathbf{readl}(\mathbf{A6} +_{32} 16), l_6 \\
l_6 : & \mathbf{D0} := \mathbf{mkLong}(0, 0, 0, 1), l_7 \\
l_7 : & := \mathbf{writel}(\mathbf{A0}, \mathbf{readl}(\mathbf{A1})), l_8 \\
l_8 : & \mathbf{SR} := \mathbf{calcSR}(\mathbf{D1}, \mathbf{D0}), l_9 \\
l_9 : & \mathbf{if\ bit}(2)(\mathbf{SR})\ \mathbf{then\ goto}\ l_{16}\ \mathbf{else\ goto}\ l_{10} \\
 & \\
l_{10} : & \mathbf{D2} := \mathbf{readl}(\mathbf{A1} + (\mathbf{D0} \times_{32} 4)), l_{11} \\
l_{11} : & \mathbf{D2}, := \mathbf{readl}(\mathbf{A0} + (\mathbf{D0} \times_{32} 4) -_{32} 4) +_{32} \mathbf{D2}, l_{12} \\
l_{12} : & := \mathbf{writel}(\mathbf{A0} +_{32} (\mathbf{D0} \times_{32} 4), \mathbf{D2}), l_{13} \\
l_{13} : & \mathbf{D0} := \mathbf{D0} +_{32} 1, l_{14} \\
l_{14} : & \mathbf{SR} := \mathbf{calcSR}(\mathbf{D1}, \mathbf{D0}), l_{15} \\
l_{15} : & \mathbf{if\ not\ bit}(2)(\mathbf{SR})\ \mathbf{then\ goto}\ l_{10}\ \mathbf{else\ goto}\ l_{16} \\
 & \\
l_{16} : & \mathbf{D2} := \mathbf{readl}(\mathbf{A6} -_{32} 4), l_{17} \\
l_{17} : & \mathbf{A7}, \mathbf{A6} := \mathbf{A6} +_{32} 4, \mathbf{readl}(\mathbf{A7}), l_{18} \\
l_{18} : & \mathbf{A7} := \mathbf{A7} +_{32} 4, \mathbf{readl}(\mathbf{A7})
\end{array}
$$

Figure 5.7: Summation: $\mathcal{L}$ Program *sum*

$\mathcal{L}$ **Program** *sum*

The M68000 program is translated to the language $\mathcal{L}$ by replacing the instructions of Figure (5.6) with $\mathcal{L}$ commands. The result is the $\mathcal{L}$ program *sum* of Figure (5.7). All memory accesses in the M68000 program are as long-words and modelled by the $\mathcal{L}$ functions **readl** and **writel**. In the M68000 processor language, the majority of instructions update the status register **SR**. When the arithmetic operations are on unsigned integers, the only use of the status register is in the conditional branch instructions. All operations are assumed to be on unsigned integers and assignments to the status register which do not affect the execution of the program have been removed from the $\mathcal{L}$ program *sum*.

The data operations used in the program *sum* lead to a large proof of correctness, which will not be given here. However, the program is straightforward and, other than the length of the proof, the verification causes no difficulties. The steps needed for abstracting and constructing a proof of correctness will be described. The commands of the $\mathcal{L}$ program *sum* labelled $l_1, \ldots, l_{18}$ will be referred to as $c_1, \ldots, c_{18}$.

Specification of *sum*

The specification is of the function implemented by the program and will not consider the system dependent operations. For example, the commands c_1 and c_2, which obtain the arguments and store data, are required to satisfy the constraints imposed by the operating system. Similarly, commands c_{16}, c_{17} and c_{18} restore the state of the registers before passing control out of the program. The specification is of the behaviour of the program between commands c_3 and c_{16} and the program is considered to terminate when control reaches command c_{16}.

The precondition, *Pre*, requires that the long-word stored in location $\mathbf{A6} +_{32} 8$ is greater than 1 ($n > 1$). The address of the first element of array a is stored in location $\mathbf{A6} +_{32} 12$ and the first element of array b is in location $\mathbf{A6} +_{32} 16$. Each element is a long-word, stored in four consecutive bytes, and the arrays must be distinct.

$$\mathit{Pre}(n, a, b) \stackrel{\text{def}}{=} \begin{cases} (\mathbf{A6} +_{32} 8) =_{32} n \wedge n >_{32} 1 \\ \wedge(\mathbf{A6} +_{32} 12) =_{32} a \wedge (\mathbf{A6} +_{32} 16) =_{32} b \\ \wedge\forall(\lambda v :\ n >_{32} v \Rightarrow (a +_{32} (v \times_{32} 4) >_{32} b +_{32} (v \times_{32} 4)) \\ \qquad\qquad\qquad\qquad \vee (b +_{32} (v \times_{32} 4) >_{32} a +_{32} (v \times_{32} 4))) \end{cases}$$

The postcondition, *Post*, requires that every element i of array b is the sum of the first i elements of array a.

$$\mathit{Post}(n, a, b) \stackrel{\text{def}}{=} \begin{cases} \forall(\lambda x :(n >_{32} x) \\ \qquad \Rightarrow \mathbf{ref}(b +_{32} (x \times_{32} 4)) \\ \qquad\quad =_{32} (\mathbf{ref}(b +_{32} (x -_{32} 1) \times_{32} 4) +_{32} \mathbf{ref}(a +_{32} x \times_{32} 4))) \end{cases}$$

The program is correct if beginning in a state satisfying the precondition, execution of the program eventually leads to the establishment of the postcondition at command c_{16}.

$$\vdash [\mathit{Pre}(n, a, b) \wedge \mathit{pc} =_{32} l_3]\mathit{sum}[\mathit{Post}(n, a, b) \wedge \mathit{pc} =_{32} l_{16}] \quad (\text{for } n, a, b \in \mathit{Values})$$

Abstraction of *sum*

An abstraction sum_1 of program *sum* can be obtained by constructing a region beginning at command c_3 and excluding commands c_{16}, c_{17} and c_{18}: $r = \mathit{region}(l_3, \mathit{sum} - \{c_{16}, c_{16}, c_{18}\})$. Since c_3 does not reach either command c_1 or command c_2, these are excluded from r by the definition of *region*.

An abstraction is constructed from r by applying transformation T_2. The cut-points of r are c_3, the head of the region, and c_{10}, which begins a loop in r, and two regions are constructed from these commands.

$$\begin{aligned} r_1 &= \mathit{region}(l_3, \{c_3, c_4, c_5, c_6, c_7, c_8, c_9\}) \\ r_2 &= \mathit{region}(l_{10}, \{c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}\}) \end{aligned}$$

$$
\begin{array}{l}
C_1 = l_3 : \textbf{if readl}(\mathbf{A6} +_{32} 8) -_{32} 1 =_{32} 1 \\
\quad \textbf{then} := ((\mathbf{D1}, \textbf{readl}(\mathbf{A6} +_{32} 8)) \cdot (\mathbf{A1}, \textbf{readl}(\mathbf{A6} +_{32} 12)) \\
\qquad \cdot (\mathbf{A0}, \textbf{readl}(\mathbf{A6} +_{32} 16)) \cdot (\mathbf{D0}, 1) \\
\qquad \cdot (\mathbf{SR}, \textbf{calcSR}(\textbf{readl}(\mathbf{A6} +_{32} 8)), 1), \\
\qquad \cdot \textbf{writel}(\textbf{readl}(\mathbf{A6} +_{32} 16), \textbf{readl}(\mathbf{A6} +_{32} 12)), l_{16}) \\
\quad \textbf{else} := ((\mathbf{D1}, \textbf{readl}(\mathbf{A6} +_{32} 8)) \cdot (\mathbf{A1}, \textbf{readl}(\mathbf{A6} +_{32} 12)) \\
\qquad \cdot (\mathbf{A0}, \textbf{readl}(\mathbf{A6} +_{32} 16)) \cdot (\mathbf{D0}, 0) \\
\qquad \cdot (\mathbf{SR}, \textbf{calcSR}(\textbf{readl}(\mathbf{A6} +_{32} 8)), 1) \\
\qquad \cdot \textbf{writel}(\textbf{readl}(\mathbf{A6} +_{32} 16), \textbf{readl}(\mathbf{A6} +_{32} 12)), l_{10}) \\
C_2 = l_{10} : \textbf{if not D1} -_{32} (\mathbf{D0} +_{32} 1) =_{32} 0 \\
\quad \textbf{then} := ((\mathbf{D2}, \textbf{readl}(\mathbf{A0} +_{32} (\mathbf{D0} \times_{32} 4)) +_{32} \textbf{readl}(\mathbf{A1} +_{32} (\mathbf{D0} \times_{32} 4))) \\
\qquad \cdot (\mathbf{D0}, \mathbf{D0} +_{32} 1) \cdot (\mathbf{SR}, \textbf{calcSR}(\mathbf{D1}, \mathbf{D0} +_{32} 1)) \\
\qquad \cdot\ \textbf{writel}(\textbf{readl}(\mathbf{A0} +_{32} (\mathbf{D0} \times_{32} 4)), \\
\qquad\qquad \textbf{readl}(\mathbf{A0} +_{32} (\mathbf{D0} \times_{32} 4)) +_{32} \textbf{readl}(\mathbf{A1} +_{32} (\mathbf{D0} \times_{32} 4))), \\
\qquad l_{10}) \\
\quad \textbf{else} := ((\mathbf{D2}, \textbf{readl}(\mathbf{A0} +_{32} (\mathbf{D0} \times_{32} 4)) +_{32} \textbf{readl}(\mathbf{A1} +_{32} (\mathbf{D0} \times_{32} 4))) \\
\qquad \cdot (\mathbf{D0}, \mathbf{D0} +_{32} 1) \cdot (\mathbf{SR}, \textbf{calcSR}(\mathbf{D1}, \mathbf{D0} +_{32} 1)) \\
\qquad \cdot\ \textbf{writel}(\textbf{readl}(\mathbf{A0} +_{32} (\mathbf{D0} \times_{32} 4)), \\
\qquad\qquad \textbf{readl}(\mathbf{A0} +_{32} (\mathbf{D0} \times_{32} 4)) +_{32} \textbf{readl}(\mathbf{A1} +_{32} (\mathbf{D0} \times_{32} 4))), \\
\qquad l_{16})
\end{array}
$$

Figure 5.8: Commands of Abstraction sum_1

Both r_1 and r_2 are single loops and are transformed by applying T_1 to obtain the set $gtbody(r)$.

$$
\begin{array}{l}
T_1(r_1) = (c_3; c_4; c_5; c_6; c_7; c_8; c_9) \\
T_1(r_2) = (c_{10}; c_{11}; c_{12}; c_{13}; c_{14}; c_{15}) \\
gtbody(r) = \{T_1(r_1), T_2(r_2)\}
\end{array}
$$

The commands $T_1(r_1)$ and $T_1(r_2)$, will be referred to as C_1 and C_2: $C_1 = T_1(r_1)$ and $C_2 = T_1(r_2)$. Commands C_1 and C_2, after simplification, are given in Figure (5.8). The simplifications are straightforward since there is only one assignment to a memory variable, **writel**, in each region and the name expression can be distinguished syntactically. Note that the expression **bit**(2)(**SR**) of commands c_9 and c_{15} reduces, after substitution, to $\mathbf{D1} -_{32} \mathbf{D0} =_{32} 0$.

Command C_1 is the head of region $T_2(r)$ and, since $C_1 \longmapsto C_2$, C_2 is in the body of region $T_2(r)$, $body(T_2(r)) = gtbody(r)$. The abstraction sum_1 is obtained by combining the body of $T_2(r)$ with *sum*.

$$sum_1 = sum \uplus body(T_2(r))$$

For the precondition $pc =_{32} l_3 \wedge \mathit{Pre}(n, a, b)$ and postcondition $pc =_{32} l_{16} \wedge \mathit{Post}(n, a, b)$ only commands C_1 and C_2 are required. Command C_1 is enabled when $pc =_{32} l_3$ and selects either command c_{16} or command C_2. Command C_2 forms a loop and is enabled when $pc =_{32} l_{10}$. Execution of C_2 selects either C_2, to perform a second iteration, or selects c_{16}, terminating the loop.

Verification of sum_1

Program sum_1 is correct if, for any $n, a, b \in \mathit{Values}$, with execution beginning with the command labelled l_3 in a state satisfying the precondition, control reaches the command labelled l_{16} and establishes the postcondition.

$$\vdash [\mathit{Pre}(n, a, b) \wedge pc =_{32} l_3]\mathit{sum}_1[\mathit{Post}(n, a, b) \wedge pc =_{32} l_{16}]$$

Command C_2 forms a loop and the proof is by induction on the difference between the number of elements n and the value of the counter (i of the C program). The invariant, *Inv*, for the loop at C_2 is applied to four arguments, the arrays a and b, the number of elements n and the difference d between n and the value of the counter, **D0**. The invariant requires that $\mathbf{D1} =_{32} n$ and the address of the first elements of array a and b are stored in register **A1** and **A0** respectively. The arrays must be distinct and the value stored in $b(x)$ for $x <_a \mathbf{D0}$ must be $b(x -_{32} 1) +_{32} a(x)$. When the counter **D0** is equal to n, $d = 0$, the loop terminates and control passes to command c_{16}.

$$\mathit{Inv} : (\mathit{Values} \times \mathit{Values} \times \mathit{Values} \times \mathit{Values}) \to \mathcal{A}$$

$$\mathit{Inv}(d, n, a, b) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} d =_{32} n -_a \mathbf{D0} \wedge n =_{32} \mathbf{D1} \wedge a =_{32} \mathbf{A0} \wedge b =_{32} \mathbf{A1} \\ \wedge \forall(\lambda x : (\mathbf{D0} >_{32} x) \\ \qquad \Rightarrow \mathbf{ref}(b +_{32} (x \times_{32} 4)) \\ \qquad\quad =_{32} (\mathbf{ref}(b +_{32} (x -_{32} 1) \times_{32} 4) +_{32} \mathbf{ref}(a +_{32} x \times_{32} 4))) \\ \wedge \forall(\lambda v :\ n >_{32} v \Rightarrow (a +_{32} (v \times_{32} 4) >_{32} b +_{32} (v \times_{32} 4)) \\ \qquad\qquad\qquad \vee (b +_{32} (v \times_{32} 4) >_{32} a +_{32} (v \times_{32} 4))) \\ \wedge (d =_{32} 0 \Rightarrow pc =_{32} l_{16}) \end{array} \right.$$

The verification is in two parts: the first is to establish the invariant from the precondition. The initial difference, d, between n and **D0** is $n - 1$ since $\mathbf{D0} =_{32} 1$ when the loop at l_{10} begins and the assertion to be established in the first step is, for any n, a, b, $\mathit{Inv}(n - 1, a, b)$. The second step is to show that, for any $d \in \mathit{Values}$, the invariant $\mathit{Inv}(d, n, a, b)$ establishes the postcondition $\mathit{Post}(n, a, b)$ and that control will be at command c_{16}. The proof is in the following steps:

1. Precondition establishes invariant:
   $\vdash [\mathit{Pre}(n, a, b) \wedge pc =_{32} l_3]\mathit{sum}_1[\mathit{Inv}(n-1, n, a, b) \wedge pc =_{32} l_{16}]$ From the definition of *Pre*, $n >_{32} 1$ and $\mathbf{readl}(\mathbf{A6} +_{32} 8) =_{32} n$. By the conditional rule (tl3), the proof is based on the false branch of command C_1 and is straightforward from application of the rules and by substitution.

2. Invariant establishes postcondition:
   $\vdash [\mathit{Inv}(d, n, a, b) \wedge \mathit{pc} =_{32} l_{10}]\mathit{sum}_1[\mathit{Post}(n, a, b) \wedge \mathit{pc} =_{32} l_{16}]$

   The proof is by the induction on d with the inductive hypothesis that for $i < d$ and precondition $\mathit{Inv}(i, n, a, b)$, sum_1 establishes $\mathit{Post}(n, a, b)$.

   The proof is based on the command C_2, which is enabled when $\mathit{pc} =_{32} l_{10}$. There are two cases to consider: the base case, when $\mathbf{D1} =_{32} \mathbf{D0} +_{32} 1$, establishes the postcondition directly, and the inductive case establishes the assumptions of the inductive hypothesis by showing that when $\mathbf{D1} =_{32} \mathbf{D0} +_{32} 1$, sum_1 establishes $\mathit{Inv}(d - 1, n, a, b)$.

   (a) Invariant establishes postcondition (base case):
   $\vdash [\mathit{Inv}(d, n, a, b) \wedge \mathit{pc} =_{32} l_{10} \wedge \mathbf{D1} =_{32} \mathbf{D0} +_{32} 1]\mathit{sum}_1[\mathit{Post}(n, a, b) \wedge \mathit{pc} =_{32} l_{16}]$
   From the assumption, $\mathit{Inv}(d, n, a, b)$, $\mathbf{D1} =_{32} n$ and therefore $d =_{32} 1$. The command C_2 assigns $\mathbf{D0} +_{32} 1$ to $\mathbf{D1}$ and the postcondition for this case can be strengthened with $\mathit{Inv}(0, n, a, b)$.

   $$\vdash \mathit{Inv}(d, n, a, b) \wedge \mathit{pc} =_{32} l_{10} \wedge \mathbf{D1} =_{32} \mathbf{D0} +_{32} 1 \Rightarrow \mathbf{wp}(C_2, Inv(0, n, a, b))$$

   From the definition of *Inv*, $\vdash \mathit{Inv}(0, n, a, b) \Rightarrow \mathit{pc} =_{32} l_{16}$. The remainder of the proof for this case is to show that the invariant, with $d =_{32} 0$, establishes *Post*.

   $$\vdash \mathit{Inv}(0, n, a, b) \Rightarrow \mathit{Post}(n, a, b)$$

   This is straightforward from the definitions of *Inv* and *Post*.

   (b) Invariant establishes inductive hypothesis (inductive case):
   $\vdash [\mathit{Inv}(d, n, a, b) \wedge \mathit{pc} =_{32} l_{10} \wedge \neg \mathbf{D1} =_{32} \mathbf{D0} +_{32} 1]\mathit{sum}_1[\mathit{Inv}(d-1, n, a, b) \wedge \mathit{pc} =_{32} l_{16}]$
   Command C_2 is selected and the proof is by establishing the weakest precondition for invariant $\mathit{Inv}(d - 1, n, a, b)$ from the assumptions.

   $$\begin{aligned}&\vdash \mathit{Inv}(d, n, a, b) \wedge \mathit{pc} =_{32} l_{10} \wedge \neg \mathbf{D1} =_{32} \mathbf{D0} +_{32} 1\\ &\quad\Rightarrow \mathbf{wp}(C_2, \mathit{Inv}(d - 1, n, a, b) \wedge \mathit{pc} =_{32} l_{16})\end{aligned}$$

   Since $\mathbf{D1} =_{32} \mathbf{D0} +_{32} 1$ is *false*, the true branch of C_2 is chosen. This assigns l_{10} to the program counter, establishing $\mathit{pc} =_{32} l_{10}$, and the proof of $\mathit{Inv}(d-1, n, a, b)$ is obtained by substitution of the assignments and from the assumption $\mathit{Inv}(d, n, a, b)$. The inductive hypothesis can then be used to establish the postcondition $\mathit{Post}(n, a, b) \wedge \mathit{pc} =_{32} l_{16}$, completing the proof.

By showing that the invariant establishes the postcondition and that the precondition establishes the invariant, the proof follows from the transitivity rule.

$$\begin{array}{lr}\vdash [\mathit{Pre}(n, a, b) \wedge \mathit{pc} =_{32} l_3]\mathit{sum}_1[\mathit{Inv}(n - 1, n, a, b) \wedge \mathit{pc} =_{32} l_{10}] & \\ \vdash [\mathit{Inv}(d, n, a, b) \wedge \mathit{pc} =_{32} l_{10}]\mathit{sum}_1[\mathit{Post}(n, a, b) \wedge \mathit{pc} =_{32} l_{16}] & (\text{for all } d \in \mathit{Values}) \\ \vdash [\mathit{Pre}(n, a, b) \wedge \mathit{pc} =_{32} l_3]\mathit{sum}_1[\mathit{Post}(n, a, b) \wedge \mathit{pc} =_{32} l_{16}] & (\text{Transitivity, tl8})\end{array}$$

_div:	link a6,#0	l_1 :	$:=(\mathbf{A6}, \mathbf{A7} -_{32} 4) \cdot (\mathbf{A7}, \mathbf{A7} -_{32} 4)$ $\cdot\, \mathbf{writel}(\mathbf{A7} -_{32} 4, \mathbf{A6}), l_2$
	movel a6@(8),d1	l_2 :	$\mathbf{D1} := \mathbf{readl}(\mathbf{A6} +_{32} 8), l_3$
	movel a6@(12),a0	l_3 :	$\mathbf{A0} := \mathbf{readl}(\mathbf{A6} +_{32} 12), l_4$
	movel a6@(16),a1	l_4 :	$\mathbf{A1} := \mathbf{readl}(\mathbf{A6} +_{32} 16), l_5$
	clrl d0	l_5 :	$\mathbf{D0} := \mathbf{Long}(0), l_6$
	cmpl d1,a0	l_6 :	$\mathbf{SR} := \mathbf{calcSR}(\mathbf{A0}, \mathbf{D1}), l_7$
	jcc L8	l_7 :	**if not bit**$(2)(\mathbf{SR})$ **then goto** l_{12} **else goto** l_8
L9:	addql #1,d0	l_8 :	$\mathbf{D0} := \mathbf{D0} +_{32} 1, l_9$
	subl a0,d1	l_9 :	$\mathbf{D1} := \mathbf{D1} -_{32} \mathbf{A0}, l_{10}$
	cmpl d1,a0	l_{10} :	$\mathbf{SR} := \mathbf{calcSR}(\mathbf{A0}, \mathbf{D1}), l_{11}$
	jcs L9	l_{11} :	**if bit**$(2)(\mathbf{SR})$ **then goto** l_8 **else goto** l_{12}
L8:	movel d1,a1@	l_{12} :	$:= \mathbf{writel}(\mathbf{D1}, \mathbf{A1}), l_{13}$
	unlk a6	l_{13} :	$\mathbf{A7}, \mathbf{A6} := \mathbf{A6} +_{32} 4, \mathbf{readl}(\mathbf{A7}), l_{14}$
	rts	l_{14} :	$\mathbf{A7} := \mathbf{A7} +_{32} 4, \mathbf{readl}(\mathbf{A7})$
	M68000 program		$\mathcal{L}$ program *m68div*

Figure 5.9: Division: M68000

This completes the proof for program sum_1. □

The verification is of the function implemented by the program. To show that the program operates correctly, the specification must be strengthened with assertions on the state of the machine. The properties described by these assertions will be those required for the correct operation of the program and will vary between machines and operating systems. For example, the object program of Figure (5.6) terminates when control passes to the address at the top of the stack, **readl**(**A7**). To ensure that the program returns control to the correct instruction, the value of the expression **readl**(**A7**) must be maintained by the program. However, this property does not affect the result of the function implemented by the program.

5.2.4 Example: Division

The processor language determines the variables and data operations available to a program; these affect the implementation of programs for the processor. Because the M68000 processor provides a limited number of registers, programs of the M68000 make extensive use of the memory variables. However, this does not significantly affect the difficulty of verifying a program.

For an example of the effect of the processor language, the C program of Figure (5.1) for division was compiled for the M68000 processor to produce the object code program of Figure (5.9). This was translated into the $\mathcal{L}$ program *m68div* of Figure (5.9), which will be shown to implement the division of natural numbers.

The major differences between program *idiv* (of Figure 5.2) and program *m68div* are in the use of registers and memory variables. Parameters to program *idiv* are passed in registers and the result is stored in a register. Parameters to program *m68div* are passed on the machine stack, which is implemented as memory variables identified by register **A7**; the result of *m68div* is also stored in a register. Other differences include the use of the status register **SR** to record the result of comparisons and the use of the stack to store the label to which control is to return. These differences are a consequence of the instructions provided by the M68000 processor language. The differences between *idiv* and *m68div* do not affect the approach to verifying the programs.

Specification of *m68div*

The specification of program *m68div* is essentially that of *idiv*, the only changes required are to reflect the use of different variables. The C variable n is passed to *m68div* in address $(\mathbf{A7} +_{32} 4)$, variable d is passed in address $(\mathbf{A7} +_{32} 8)$ and variable r is passed in address $(\mathbf{A7} +_{32} 12)$. When the program terminates, it must return control to the label l stored in the location identified by **A7**. At the end of the program, the result must be stored in register **D0**. The precondition of *m68div* is *m68Pre*$(n, d, a, l) \in \mathcal{A}$ and the postcondition is *m68Post*$(n, d, a, l) \in \mathcal{A}$ (where $n, d, a, l \in \mathbb{N}$):

$$\begin{aligned}
\textit{m68Pre}(n,d,a,l) &\stackrel{\text{def}}{=} d >_a 0 \wedge n \geq_a 0 \wedge \mathbf{readl}(\mathbf{A7} +_{32} 4) =_{32} n \wedge \mathbf{readl}(\mathbf{A7} +_{32} 8) =_{32} d \\
&\qquad \wedge \mathbf{readl}(\mathbf{A7} +_{32} 12) =_{32} a \wedge \mathbf{readl}(\mathbf{A7}) =_{32} l \\
\textit{m68Post}(n,d,a,l) &\stackrel{\text{def}}{=} n \geq_a 0 \wedge d >_a 0 \wedge \mathbf{readl}(\mathbf{A7}) =_{32} l \wedge \\
&\qquad \mathbf{A1} =_{32} a \wedge n =_{32} (\mathbf{D0} \times_{32} +_{32} \mathbf{readl}(\mathbf{readl}(\mathbf{A7} +_{32} 12)))
\end{aligned}$$

The verification of *m68div* is by induction on the loop beginning at label l_8 to establish an invariant from which the postcondition follows. The assertion *m68Inv*(q, n, d, a, l) (where $q, n, d, a, , l \in \mathbb{N}$) is the invariant for the loop in *m68div* at label l_8:

$$\textit{m68Inv}(q,n,d,a,l) \stackrel{\text{def}}{=} \begin{cases} n \geq_a 0 \wedge d >_a 0 \wedge q =_{32} \mathbf{D0} \\ \wedge l =_{32} \mathbf{readl}(\mathbf{A6} +_{32} 4) \wedge a =_{32} \mathbf{A1} \wedge d =_{32} \mathbf{A0} \wedge \\ \wedge (\mathbf{D0} \times_{32} \mathbf{D1}) +_{32} \mathbf{D0}) =_{32} n \\ \wedge (\mathit{pc} =_{32} l \Rightarrow \mathbf{readl}(\mathbf{A1}) =_{32} \mathbf{D1}) \end{cases}$$

As with the pre- and postcondition, the invariant for *m68div* is that of *idiv*, with changes made to reflect the variables used.

The specification to be satisfied by *m68div* is, for any $n, d, a, l \in \mathbb{N}$:

$$\vdash [\mathit{pc} =_{32} l_1 \wedge \textit{m68Pre}(n,d,a,l)]\textit{m68div}[\mathit{pc} =_{32} l \wedge \textit{m68Post}(n,d,a,l)]$$

The steps needed to verify *m68div* are similar to those for *idiv* and a proof of correctness for *m68div* will not be given. The abstraction of program *m68div* and the steps needed to verify the abstraction will be briefly described.

$$
\begin{array}{l}
C_1 = l_1 : \textbf{if not mkBit}(\textbf{readl}(\textbf{A6} +_{32} 12) -_{32} \textbf{readl}(\textbf{A6} +_{32} 8)) \\
\quad \textbf{then} := (\textbf{A7}, \textbf{A7} +_{32} 4) \cdot (\textbf{D1}, \textbf{readl}(\textbf{A6} +_{32} 8)) \\
\qquad \cdot (\textbf{A0}, \textbf{readl}(\textbf{A6} +_{32} 12)) \cdot (\textbf{A1}, \textbf{readl}(\textbf{A6} +_{32} 16)) \cdot (\textbf{D0}, \textbf{Long}(0)) \\
\qquad \cdot (\textbf{SR}, \textbf{calcSR}(\textbf{readl}(\textbf{A6} +_{32} 12), \textbf{readl}(\textbf{A6} +_{32} 8))) \\
\qquad \cdot (\textbf{writel}(\textbf{readl}(\textbf{A6} +_{32} 8), \textbf{readl}(\textbf{A6} +_{32} 16)) \\
\qquad \cdot (\textbf{writel}(\textbf{A7} -_{32} 4, \textbf{A6})), \textbf{readl}(\textbf{A7}) \\
\quad \textbf{else} := (\textbf{A6}, \textbf{A7} -_{32} 4) \cdot (\textbf{A7}, \textbf{A7} -_{32} 4) \cdot (\textbf{D1}, \textbf{readl}(\textbf{A6} +_{32} 8)) \\
\qquad \cdot (\textbf{A0}, \textbf{readl}(\textbf{A6} +_{32} 12)) \cdot (\textbf{A1}, \textbf{readl}(\textbf{A6} +_{32} 16)) \cdot (\textbf{D0}, \textbf{Long}(0)) \\
\qquad \cdot (\textbf{SR}, \textbf{calcSR}(\textbf{readl}(\textbf{A6} +_{32} 12), \textbf{readl}(\textbf{A6} +_{32} 8))) \\
\qquad \cdot (\textbf{writel}(\textbf{A7} -_{32} 4, \textbf{A6})), l_8
\end{array}
$$

$$
\begin{array}{l}
C_2 = l_8 : \textbf{if mkBit}(\textbf{A0} -_{32} (\textbf{D1} -_{32} \textbf{A0})) \\
\quad \textbf{then} := (\textbf{D0}, \textbf{D0} +_{32} 1) \cdot (\textbf{D1}, \textbf{D1} -_{32} \textbf{A0}) \cdot (\textbf{SR}, \textbf{calcSR}(\textbf{A0}, (\textbf{D1} -_{32} \textbf{A0})), l_8 \\
\quad \textbf{else} := (\textbf{D0}, \textbf{D0} +_{32} 1) \cdot (\textbf{D1}, \textbf{D1} -_{32} \textbf{A0}) \cdot (\textbf{SR}, \textbf{calcSR}(\textbf{A0}, (\textbf{D1} -_{32} \textbf{A0}))) \\
\qquad \cdot (\textbf{A7}, \textbf{A6} +_{32} 8) \cdot (\textbf{A6}, \textbf{readl}(\textbf{A7})) \cdot \textbf{writel}(\textbf{D1} -_{32} \textbf{A0}, \textbf{A1}), \textbf{readl}(\textbf{A6} +_{32} 4)
\end{array}
$$

Figure 5.10: Commands of Abstraction *m68div*$_2$

Abstraction of *m68div*

The abstraction of *m68div* is constructed in two parts. First the general transformation T_2 is applied to a region which is constructed from all commands of *m68div* which have a constant successor expression. These are all commands except the command at l_{14}. The result is combined with *m68div* to form the first abstraction *m68div*$_1 \sqsubseteq$ *m68div*. The command at l_{14} is then composed with commands of *m68div*$_1$ to form the second abstraction *m68div*$_2$. The command of *m68div* labelled l_i will be referred to as c_i; e.g. c_1 is the command labelled l_1.

To construct the first abstraction *m68div*$_1$: region r is constructed as $r = \textit{region}(l_1, \textit{m68div} - \{c_{14}\})$. There are two cut-points in r: command c_1 (the head of r) and command c_8. These form two regions r_1 and r_2 in r: $r_1 = \textit{region}(l_1, \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{12}, c_{13}\})$ and $r_2 = \textit{region}(l_8, \{c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}\})$. The result of abstracting r, $T_2(r)$, is the region containing commands $T_1(r_1)$ and $T_1(r_2)$. These are combined with *m68div* to obtain the first abstraction: $\textit{m68div}_1 = \textit{m68div} \uplus \{T_1(r_1), T_1(r_2)\}$.

The second abstraction, *m68div*$_2$ is obtained by composing command c_{14} with the two transformed regions. Let $C_1 = (T_1(r_1); c_{14})$ and $C_2 = (T_1(r_2); c_{14})$. The abstraction of *m68div*$_1$ is program $\textit{m68div}_2 = \textit{m68div}_1 \uplus \{C_1, C_2\}$ and $\textit{m68div}_2 \sqsubseteq \textit{m68div}$. Only commands $C_1, C_2 \in \textit{m68div}_2$ need to be considered during the proof of correctness. Commands C_1 and C_2, after simplification, are given in Figure (5.10).

Verification of *m68div*$_2$

Program *m68div*$_2$ is an abstraction of *m68div*: verifying *m68div*$_2$ is enough to verify *m68div*. The specification to be established is therefore, for any $n, d, a, l \in \mathbb{N}$:

$$\vdash [pc =_{32} l_1 \wedge \mathit{m68Pre}(n, d, a, l)]\mathit{m68div}_2[pc =_{32} l \wedge \mathit{m68Post}(n, d, a, l)]$$

The steps required to verify *m68div*$_2$ are those required to verify *idiv*:

1. Precondition establishes postcondition:

   $$\vdash [pc =_{32} l_1 \wedge \mathit{m68Pre}(n, d, a, l) \wedge n <_{32} d]\mathit{m68div}_2[pc =_{32} l \wedge \mathit{m68Post}(n, d, a, l)]$$

   This follows from: $\vdash (pc =_{32} l_1 \wedge \mathit{m68Pre}(n, d, a, l) \wedge n <_{32} d)$
   $\Rightarrow \mathbf{wp}(C_1, pc =_{32} l \wedge \mathit{m68Post}(n, d, a, l))$

2. Precondition establishes invariant:

   $$\vdash [pc =_{32} l_1 \wedge \mathit{m68Pre}(n, d, a, l) \wedge \neg n <_{32} d]\mathit{m68div}_2[pc =_{32} l_8 \wedge \mathit{m68Inv}(n, n, d, a, l)]$$

   This follows from: $\vdash (pc =_{32} l_1 \wedge \mathit{m68Pre}(n, d, a, l) \wedge \neg n <_{32} d)$
   $\Rightarrow \mathbf{wp}(C_1, pc =_{32} l_8 \wedge \mathit{m68Inv}(n, n, d, a, l))$

3. Invariant establishes postcondition, the proof is by induction on $q \in \mathbb{N}$ (rule tl9):

   $$\vdash [pc =_{32} l_8 \wedge \mathit{m68Inv}(q, n, d, a, l)]\mathit{m68div}_2[pc =_{32} l \wedge \mathit{m68Post}(n, d, a, l)]$$

   This is in two cases, in both only command C_2 is required

   (a) Base case, $(\mathbf{D1} -_{32} \mathbf{A0}) <_{32} \mathbf{A0}$. This is by a proof of:

   $$\begin{array}{l}\vdash (pc =_{32} l_8 \wedge \mathit{m68Inv}(q, n, d, a, l) \wedge (\mathbf{D1} -_{32} \mathbf{A0}) <_{32} \mathbf{A0}) \\ \quad \Rightarrow \mathbf{wp}(C_2, pc =_{32} l \wedge \mathit{m68Post}(n, d, a, l))\end{array}$$

   (b) Inductive case, **not** $(\mathbf{D1} -_{32} \mathbf{A0}) <_{32} \mathbf{A0}$. This follows from:

   $$\begin{array}{l}\vdash (pc =_{32} l_8 \wedge \mathit{m68Inv}(q, n, d, a, l) \wedge \mathbf{not}\ \mathbf{D1} -_{32} \mathbf{A0} <_{32} \mathbf{A0}) \\ \quad \Rightarrow \mathbf{wp}(C_2, pc =_{32} l \wedge \mathit{m68Post}(n, d, a, l))\end{array}$$

Each of these steps establishes an intermediate specification for *m68div*$_2$ which can be combined, as for program *idiv*, to establish the specification of *m68div*$_2$. The correctness of *m68div* then follows by the refinement rule (tl7). Note that although *m68div* was obtained by translating a program for the M68000 processor language, the specification and verification of *m68div* is similar to the program *idiv*. This is a consequence of the use of the language $\mathcal{L}$ to model object code, which allows the syntactic differences between programs to be ignored in favour of the actions performed by the programs.

5.3 The PowerPC Architecture

The language $\mathcal{L}$ is intended to be independent of any processor language and must be able to model the instructions of processors other than the M68000. As an example of the ability to model the object code of different processors, instructions and programs of the PowerPC processor will be described in terms of $\mathcal{L}$. The PowerPC is a processor architecture based on a reduced instruction set design (Motorola Inc. and IBM Corp., 1997). The architecture is implemented as a 32 bit processor and as a 64 bit processor. The two implementations differ mainly in the size of the bit-vector which can be manipulated as single unit (bit-vectors of size 32 and 64 respectively). The description given here is of a subset of the language of the 32 bit processor.

The terms and conventions used by the PowerPC language differ from those used to describe the M68000 processors. The PowerPC reference manual (Motorola Inc. and IBM Corp., 1997) defines a *byte* to be a bit-vector of 8 bits, a *half-word* to be a bit-vector of 16 bits and a *word* to be a bit-vector of 32 bits. The naming of the PowerPC instructions reflects these terms. In the PowerPC processor language, the most significant bit of a bit-vector of size n is said to be a position 0, and the least significant bit at position $n-1$. This is reflected in the instructions whose arguments include the position of a bit in a bit-vector. The terms and conventions used in the description here will be as before: a bit-vector of 16 bits will be called a *word*, a bit-vector of 32 bits a *long-word*, the most significant bit is at position $n-1$ and the least significant at position 0.

The values which can be represented by the PowerPC are the natural numbers up to 2^{32}. Each address is a value and each memory location stores a byte. An instruction of the PowerPC is stored in 4 consecutive locations and the label of each instruction must be a multiple of 4. When control passes to label l (**goto** l) and l is not a multiple of 4, the two least significant bits are ignored. For example, values 0, 1, 2 and 3 are interpreted as the location 0.

5.3.1 Registers

The PowerPC has 32 general purpose integer registers, **r0** to **r31**, a *condition register*, **CR**, and a *link register*, **LR**. Other registers including a special purpose register **XER** and a *count register* **CTR**. Although there is no program counter visible to the object code programs of the PowerPC, a program counter is assumed in the definition of the semantics given in the processor manual (Motorola Inc. and IBM Corp., 1997). The set of registers for the PowerPC will include the identifier *pc* to act as the program counter of the language $\mathcal{L}$.

$$\{\mathbf{r0}, \mathbf{r1}, \ldots, \mathbf{r31}, \mathbf{CR}, \mathbf{LR}, \mathbf{XER}, \mathbf{CTR}, pc\} \subseteq \mathit{Regs}$$

All registers store values as bit-vectors of length 32. The general purpose registers **r0** to **r31** store addresses in memory for instructions which move data between the registers and memory. The general purpose registers also store the arguments to and results of the operations implemented by the processor instructions. The link register **LR** is used to implement sub-routines and stores the address to which control is to return.

The condition register, **CR**, is organised as 8 *fields* of 4 bits each. Each field reflects the result of an operation on data and is similar to the status register, **SR**, of the M68000. Bit 0 of a field indicates that the result of an operation was a negative number (using two's complement representation); bit 1 indicates that the result was a positive number (greater than 0); bit 2 indicates that the result was 0 and bit 3 indicates that an overflow occurred. An instruction may specify the field of the condition register to be used, when the field is not specified, field 0 is assumed.

The **XER** register is also used to store information about data. The two most significant bits of the register indicate that an operation generated an overflow, the third most significant bit indicates that an operation generated a carry. The least significant byte of the register is used by some instructions as a counter. The count register, **CTR**, is used together with some forms of the jump instruction to implement iteration and stores the number of iterations be performed. The count register is also used in some instructions to store the label of the instruction to which control is to pass.

5.3.2 Instructions

The general form of a PowerPC instruction is `inst` dst, src_1, src_2 where `inst` is the instruction name, *dst* is the destination argument and src_1, src_2 are the source arguments. Each instruction operates on data of a given size and this determines the instruction name. For example, the instructions named `lbz` and `lwz` implement the same operation but operate on a byte and a long-word (a PowerPC word) respectively.

When the general purpose registers, **r0**, ... , **r31** occur as an argument to an instruction, they are denoted by the numbers 0 to 31. Whether an instruction argument such as 1 is interpreted as a register or as a value is determined by the semantics of the instruction. Here, instruction arguments which are registers will be written **r**, **r0**, ... and values will be denoted $v, v_0, \ldots$. The instructions may interpret a value v as either a signed or an unsigned number.

Data Movement

There are two types of data movement instruction: a *load* moves data from the memory variables to the registers, a *store* moves data from the registers to memory. For both operations, a number of instructions are defined to operate on different sizes of data and to use the different addressing modes of the PowerPC.

In the *register indirect* addressing mode, the operand is in memory at the location determined from a register **r**. In the *register indirect with immediate index* mode, an argument is written $v(\mathbf{r})$ and the address of the operand is obtained by the addition of v to the register **r**. In the *register indirect with index* mode there are at least two source registers **r1** and **r2** and the address is obtained by the addition of **r1** and **r2**. All addressing modes for the data movement instructions interpret the register **r0** as the value 0. For either of the indexed addressing modes, the instruction may also update a register with the calculated address.

The *load word and zero with update indexed*, written `lwzux` $\mathbf{r1}, \mathbf{r2}, \mathbf{r3}$, operates on a long-word (a PowerPC word) and uses register indirect with index addressing. Register **r1** and **r2** are destination arguments and the source argument is register **r3**. The sum of **r2** and **r3** identifies a location in memory in which a value v is stored. Register **r1** is assigned the value v and register **r2** is assigned the address of v.

$$\mathbf{r1}, \mathbf{r2} := \mathbf{ref}(\mathbf{r2} +_{32} \mathbf{r3}), \mathbf{r2} +_{32} \mathbf{r3} \quad (\text{if } \mathbf{r1} \neq 0)$$

The PowerPC reference manual defines the instruction in which $\mathbf{r1} = \mathbf{r2}$ to be invalid. This ensures that the assignment to **r1** and **r2** is always correct.

The *store half-word with update* instruction, written `sthu` $\mathbf{r1}, v(\mathbf{r2})$, stores the 16 bit value (a PowerPC half-word) contained in the lower half of the register **r1** in the memory location whose address is the sum of v and **r2**. Register **r2** is updated with this address.

$$\mathbf{ref}(\mathbf{r2} +_{32} \mathbf{ext}(\mathit{Long}, \mathit{Word}, v)), \mathbf{r2} := \mathbf{mkWord}(\mathbf{r1}), \mathbf{r2} +_{32} \mathbf{ext}(\mathit{Long}, \mathit{Word}, v)$$

Program Control

A *branch* instruction passes control to a target location and may be conditional on the value of the register **CR**. The target of the branch is a constant or is obtained from the link register **LR** or the count register **CTR**.

An *unconditional branch* has a single argument from which the target of the jump is calculated. The *absolute branch* instruction, `ba` v, passes control to the instruction at address v, $\mathbf{goto}\ \mathbf{loc}(v)$. The *branch and link* instruction, `bl` v, stores the address of the next instruction in memory in the link register. The target of the branch is the instruction whose address is calculated from $pc + v$ (where v is interpreted as a signed number).

$$\mathbf{LR}, pc := pc +_{32} 4, \mathbf{loc}(pc +_{32} v)$$

The instruction *branch to link register*, `blr`, implements a return from sub-routine: control is passed to the address stored in the link register, $\mathbf{goto}\ \mathbf{loc}(\mathbf{LR})$.

Conditional branch instructions have the form `bc` a, b, v where $a, b \leq 31$ are *control arguments* and v is the target address. Argument $a \in \mathit{Values}$ determines how the result of the test is to be interpreted and argument $b \in \mathit{Values}$ determines the bit of the condition register **CR** to be tested. Argument a may also provide information to allow the processor to predict the result of the test. This allows the processor to perform some optimisations but does not otherwise affect the execution of the program.

In the *branch conditional* instruction, `bc` $4, 6, v$ the first argument, 4, indicates that the tested bit must not be 0. The second argument indicates that the test is of bit 6 of the condition register **CR** (bit 2 of field 1 of the condition register). Argument v is the address, relative to the current instruction, to which control passes.

$$\mathbf{if\ not\ bit}(6)(\mathbf{CR})\ \mathbf{then\ goto\ loc}(pc +_{32} v)\ \mathbf{else\ goto}\ l$$

```
int strlength(char *a)
{   int c;
    c=0;
    while(*(a+c)!=0)
    { c=c+1; }
    return c;
}
```

Figure 5.11: Strlength: C Program

Arithmetic and Comparison Instructions

The arithmetic operations include addition, subtraction, multiplication and division on bytes, words and long-words. The arithmetic instructions are typically of the form `inst` $\mathbf{r0}, \mathbf{r1}, \mathbf{r2}$ and implement the operation $\mathbf{r0} := f(\mathbf{r1}, \mathbf{r2})$, where f is a function on the values. The *add* instruction, `add` $\mathbf{r0}, \mathbf{r1}, \mathbf{r2}$, assigns to $\mathbf{r0}$ the sum of $\mathbf{r1}$ and $\mathbf{r2}$. The *add immediate* instruction, `addi` $\mathbf{r0}, \mathbf{r1}, v$, assigns to $\mathbf{r0}$ the sum of $\mathbf{r1}$ and $v \in$ *Values*.

$$\begin{array}{ll} \texttt{add}\ \mathbf{r0}, \mathbf{r1}, \mathbf{r2} & \mathbf{r0} := \mathbf{r1} +_{32} \mathbf{r2} \\ \texttt{addi}\ \mathbf{r0}, \mathbf{r1}, v & \mathbf{r0} := \mathbf{r1} +_{32} v \end{array}$$

Comparison instructions set the flags of the registers **CR** and **XER** to reflect the difference between the operands. The *compare* instruction is written `cmp` $a, \mathbf{r0}, \mathbf{r1}$ and the comparison is by the subtraction of **r1** from **r0**. Field a, for $a < 8$, of the condition register **CR** is set to reflect the result of the comparison. One of bits 0, 1 and 2 of field a is set and the remainder are cleared.

Summary

The PowerPC processor language is based on a large number of general purpose registers. There are only three addressing modes and only data movement instructions can transfer data between registers and memory variables. The PowerPC supports sub-routines by providing the link register **LR** but does not otherwise support facilities such as stack or frame pointers. The instructions of the PowerPC are generally simpler than those of the M68000. For example, both the M68000 and the PowerPC have instructions to add two variables. However, the PowerPC addition instruction can only make use of the registers while the M68000 instruction can also access memory variables using a range of addressing modes. The simplicity of the PowerPC instructions and the memory operations of the processor mean that the model of instructions in the language $\mathcal{L}$ is straightforward. This model also illustrates the ability of the language $\mathcal{L}$ to describe the instructions of different processors. The complexity of the model in $\mathcal{L}$ depends on the complexity of the instructions and data operations of the processor. The language $\mathcal{L}$ does not, therefore, introduce additional complexity in the model of processor instructions.

5.3.3 Example: String Length

For an example of the verification of a PowerPC program in terms of its model in $\mathcal{L}$, consider the C program of Figure (5.11) which finds the length of a string. In the C language, a string is an array of characters terminated by the null character; a character v is a natural number such that $v < 256$ and the null character is the value 0. The argument a to the program is the address in memory of the first element of the string. The integer variable c, initially 0, counts the number of elements of the string preceding the null character. While the element stored in location $a + c$ is not 0, the value of c is incremented by 1. When the null character is found, the value of c is returned as the result of the program.

Object Code Program

The program of Figure (5.11) was compiled with the GNU C optimising compiler to obtain the program of Figure (5.12). Two instructions used in the program, `mr` and `li`, are synonyms for PowerPC instructions. The *load immediate* instruction, `li` $\mathbf{r}, v$, assigns the word v to register $\mathbf{r}$, $\mathbf{r} := \mathbf{Word}(v)$. The *move register* instruction, `mr` $\mathbf{r1}, \mathbf{r2}$, assigns the value of register **r2** to register **r1**, $\mathbf{r1} := \mathbf{r2}$. The instructions of the object program are assumed to be aligned correctly and there are four bytes between each instruction. For example, the second instruction is assumed to be stored at the address `.strlength` $+ 4$. This requirement is typically implemented by the assembler which translates the assembly language program of Figure (5.12) to the object code of the processor.

The program of Figure (5.12) begins with the command labelled `.strlength`. Register **r3** contains the address of argument a of the C program and this address is assigned to register **r9**. Register **r3** is then assigned the value 0 and implements variable c of the C program. The first element of the array, stored at the address of **r9**, is assigned to register **r0**. Register **r0** is compared against 0 and the results of the comparison are stored in field 1 of the condition register. If $\mathbf{r0} =_{32} 0$ (bit 6, or bit 2 of field 1, of the condition register is set) then the program terminates immediately, transferring control to the label stored in the link register **LR**.

The command labelled $L..4$ begins the loop implementing the `while` command of the C program. Register **r3** is incremented by 1 and the byte at the address $\mathbf{r9} + \mathbf{r3}$ is stored in register **r0**. This is compared against the null character, integer 0, and the result stored in field 1 of the condition register. If the byte is not 0 then control passes to the command labelled $L..4$. If the byte is equal to 0, the program terminates and passes control to the address stored in the link register and the result of the program is the value stored in register **r3**.

$\mathcal{L}$ Program *len*

The object code program of Figure (5.12) was translated into $\mathcal{L}$ to obtain the $\mathcal{L}$ program *len* of Figure (5.13). The PowerPC program of Figure (5.12) uses only field 1 of the condition register **CR**. In the $\mathcal{L}$ program *len* field 1 of register **CR** is represented as a register $\mathbf{CRF} \in$ *Regs*. Bit 6

```
.strlength:
        mr 9, 3              ; r9:=r3
        li 3, 0              ; r3:=0
        lbz 0, 0(9)          ; r0:=ref(r9+0);
        cmpwi 1, 0, 0        ; compare r0 with 0, results in CR field 1 (r0=0)
        bclr 12, 6           ; return if bit 6 of CR (bit 2 of field 1) is set
L..4:
        addi 3, 3, 1         ; r3:=r3+1
        lbzx 0, 9, 3         ; r0:=ref(r9+r3)
        cmpwi 1, 0, 0        ; compare r0 with 0, results in CR field 1
        bc 4, 6, L..4        ; branch to L..4 if bit 6 of CR is set (r0=0)
        blr                  ; branch to address in link register (return from routine)
```

Figure 5.12: Strlength: Optimised PowerPC Program

of the condition register **CR** corresponds to bit 2 of the first condition register field, **CRF**. The test $\mathbf{bit}(2)(\mathbf{CRF})$ is equivalent to the boolean expression $\mathbf{r0} =_{32} 0$.

A function, **calcCRF**, to construct the bit-vector of register **CRF** is defined in terms of the function **mkSR**. Function **calcCRF** is similar to the function **calcSR** defined for the M68000 processor language. The function is applied to an argument x and the values of the condition codes are determined by the rules for the comparison instruction. There are four bits in the bit-vector of **CRF** and the fifth argument to **mkSR** is undefined.

$$\mathbf{calcCRF} : \mathcal{E} \to \mathcal{E}$$
$$\mathbf{calcCRF}(x) \stackrel{\text{def}}{=} \mathbf{mkSR}(\mathit{undef}(\mathit{Values}), \mathbf{bit}(31)(\mathbf{XER}), x =_{32} 0, x >_a 0, 0 >_a x)$$

where $x =_{32} 0$ and $x >_a 0$ are the comparisons for signed numbers. The equality $=_{32}$ is used for this example since the program is defined for natural numbers. Note that for any $x \in \mathcal{E}$, $\mathbf{bit}(2)(\mathbf{calcCRF}(x)) \equiv (x =_{32} 0)$ and $\mathbf{bit}(1)(\mathbf{calcCRF}(x)) \equiv x >_a 0$.

The labels of program *len* are $l_1, l_2, \ldots, l_{10} \in \mathit{Labels}$. The commands of the program *len* at labels $l_1, l_2, \ldots, l_{10}$ will be referred as $c_1, c_2, \ldots, c_{10}$ respectively. The commands are assumed to be stored in sequence and, since each PowerPC instruction is stored in four bytes, for $1 \leq i \leq 9$, $l_{i+1} = l_i + 4$. In addition, it is assumed that label l stored in the link register is not one of $l_2, \ldots, l_{10}$. The program does not store values in memory and only registers occur in assignment commands. Since each command updates a single register, the assignment lists for each command are correct in any state.

$$
\begin{array}{ll}
l_1: & \mathbf{r9} := \mathbf{r3}, \mathbf{loc}(l_2) \\
l_2: & \mathbf{r3} := \mathbf{Long}(0), \mathbf{loc}(l_3) \\
l_3: & \mathbf{r0} := \mathbf{mkLong}(0,0,0,\mathbf{ref}(\mathbf{r9})), \mathbf{loc}(l_4) \\
l_4: & \mathbf{CRF} := \mathbf{calcCRF}(\mathbf{r0}), \mathbf{loc}(l_5) \\
l_5: & \mathbf{if\ bit}(2)(\mathbf{CRF})\ \mathbf{then\ goto\ loc}(\mathbf{LR})\ \mathbf{else\ goto\ loc}(l_6) \\
\\
l_6: & \mathbf{r3} := \mathbf{r3} +_{32} 1, \mathbf{loc}(l_7) \\
l_7: & \mathbf{r0} := \mathbf{mkLong}(0,0,0,\mathbf{ref}(r9 +_{32} r3)), \mathbf{loc}(l_8) \\
l_8: & \mathbf{CRF} := \mathbf{calcCRF}(\mathbf{r0}), \mathbf{loc}(l_9) \\
l_9: & \mathbf{if\ not\ bit}(2)(\mathbf{CRF})\ \mathbf{then\ goto\ loc}(l_6)\ \mathbf{else\ goto\ loc}(l_{10}) \\
\\
l_{10}: & \mathbf{goto\ loc}(\mathbf{LR})
\end{array}
$$

Figure 5.13: Strlength: $\mathcal{L}$ Program *len*

Specification of *len*

The precondition of the program, *Pre*(a, n, l), requires that the link register has the value l, register **r3** the address a of the first element and that the index of the first null character is n.

$$
\mathit{Pre}(a,n,l) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} a =_{32} \mathbf{r3} \wedge l =_{32} \mathbf{LR} \wedge \mathbf{ref}(\mathbf{r3} +_{32} n) =_{32} 0 \\ \wedge \forall(\lambda y : n >_{32} y \Rightarrow \neg\mathbf{ref}(\mathbf{r3} +_{32} y) =_{32} 0)) \end{array} \right.
$$

The program terminates when control passes to the label stored in the link register **LR** and this must be the label l with which execution began. The index of the first element of the string which is a null character is stored in register **r3**. All elements of the string at index $y <_a \mathbf{r3}$ must be non-zero.

$$
\mathit{Post}(a,l) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathbf{ref}(a +_{32} \mathbf{r3}) =_{32} 0 \\ \wedge l =_{32} \mathbf{LR} \wedge \forall(\lambda y : \mathbf{r3} >_{32} y \Rightarrow \neg\mathbf{ref}(a +_{32} y) =_{32} 0) \end{array} \right.
$$

Program *len* begins execution when control passes to the command at l_1 and ends when control passes to the label l of the link register. If *len* begins in a state satisfying the precondition then eventually the postcondition is established.

$$
\vdash [\mathit{Pre}(a,l) \wedge \mathit{pc} =_{32} l_1]\mathit{len}[\mathit{Post}(a,l) \wedge \mathit{pc} =_{32} \mathbf{LR}] \quad \text{for } a \in \mathit{Values}, l \in \mathit{Labels}
$$

Abstraction of *len*

The abstraction of program *len* is constructed in two steps. The first constructs and abstracts a region r of *len* in which all commands have constant successor expressions. This is used to

$$
\begin{array}{ll}
C_1 = l_1 : & \textbf{if } \mathbf{ref}(\mathbf{r3}) =_{32} 0 \\
& \quad \textbf{then } (:= ((\mathbf{r9}, \mathbf{r3}) \cdot (\mathbf{r3}, 0) \cdot (\mathbf{CRF}, \mathbf{calcCRF}(\mathbf{ref}(\mathbf{r9})))), \mathbf{loc}(\mathbf{LR})) \\
& \quad \textbf{else } (:= ((\mathbf{r9}, \mathbf{r3}) \cdot (\mathbf{r3}, 0) \cdot (\mathbf{CRF}, \mathbf{calcCRF}(\mathbf{ref}(\mathbf{r9})))), l_6) \\
C_2 = l_6 : & \textbf{if not } \mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1) =_{32} 0 \\
& \quad \textbf{then } := (((\mathbf{r3}, \mathbf{r3} +_{32} 1) \cdot (\mathbf{r0}, \mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1)) \\
& \qquad\qquad \cdot (\mathbf{CRF}, \mathbf{calcCRF}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1))), l_6) \\
& \quad \textbf{else } := (((\mathbf{r3}, \mathbf{r3} +_{32} 1) \cdot (\mathbf{r0}, \mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1)) \\
& \qquad\qquad \cdot (\mathbf{CRF}, \mathbf{calcCRF}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1))), \mathbf{loc}(\mathbf{LR}))
\end{array}
$$

Figure 5.14: Commands of Abstraction len_2

form the program len_1 such that $len_1 \sqsubseteq len$. The commands of len_1 are then composed with the commands excluded from r. This forms the program len_2 which abstracts len_1, $len_2 \sqsubseteq len_1$. The abstraction len_2 will be the program which is verified.

Program len_1 is obtained by abstracting from region $r = region(l_1, len - \{c_5 c_{10}\})$ of program *len*. The general transformation T_2 is applied to r. The cut-points of r are commands c_1 and c_2. These form two regions, r_1 and r_2, of r, to which the path transformation is applied to construct the set $gtbody(r)$.

$$
\begin{array}{l}
r_1 = region(l_1, \{c_1, c_2, c_3, c_4\}) \\
r_2 = region(l_6, \{c_6, c_7, c_8, c_9\}) \\
gtbody(r) = \{T_1(r_1), T_1(r_2)\}
\end{array}
$$

The result of $T_2(r)$ is the region $region(l_1, \{T_1(r_1), T_1(r_2)\})$. The body of $T_2(r)$ is combined with *len* to form the abstraction len_1: $len_1 = len \uplus \{T_1(r_1), T_1(r_2)\}$.

The abstraction len_2 of len_1 is obtained by composing command $T_1(r_1)$ with c_5 and command $T_1(r_2)$ with c_{10}. Let $C_1 = (T_1(r_1); c_5)$ and $C_2 = (T_1(r_2))$. Commands C_1 and C_2, after simplification, are given in Figure (5.14); a consequence of the transformations is that the condition register **CRF** is no longer used in the tests of the conditional commands. The program len_2 is obtained by combining the commands C_1 and $T_1(r_2); c_{10}$ with len_1.

$$len_2 \stackrel{\text{def}}{=} len_1 \uplus \{C_1, C_2\}$$

Note that the simplification of the commands removes bit-vector constructors when possible, e.g. the expression $\mathbf{mkLong}(0, 0, 0, \mathbf{ref}(\mathbf{r9}))$ is replaced with $\mathbf{ref}(\mathbf{r9})$ and $\mathbf{mkLong}(0, 0, 0, 0)$ is replaced with 0. The program len_2 satisfies the refinement ordering $len_2 \sqsubseteq len_1$ and therefore satisfies the ordering $len_2 \sqsubseteq len$.

Only two commands of len_2 are needed to verify the program: the first, C_1, describes the path in *len* from c_1 to the loop at c_6. The second, $T_1(r_2); c_{10}$, describes the loop in *len* beginning at c_6. The loop terminates when control is passed to the label stored in the link register **LR**.

Verification of len_2

The verification proof for program len_2 follows a similar pattern to that for the program *idiv* of Section 5.2.4 and only the main steps of the proof are described here. The proof is based on the induction rule (tl9). An assertion *Inv* is shown to be established by the precondition *Pre*, to be invariant for the loop at $T_1(r_2); c_{10}$, and to establish the postcondition.

The induction is on the difference between the value of **r3** at the beginning of the loop and the index of the first null character. The invariant *Inv* is applied to values $d, a, n \in \mathit{Values}$ and $l \in \mathit{Labels}$. As in the precondition, a is the address of the first element, n the index of the first null character and l the label stored in the link register. Value d is the difference between **r3** and n; when $d =_{32} 0$, control passes to the label l. No element of the string stored at the addresses from a up to but excluding $a +_{32}$ **r3** is the null character.

$$\mathit{Inv} : (\mathit{Values}, \mathit{Values}, \mathit{Values}, \mathit{Labels}) \to \mathcal{A}$$
$$\mathit{Inv}(d, a, n, l) \stackrel{\text{def}}{=} \begin{cases} d =_{32} \mathbf{r3} -_{32} n \land a =_{32} \mathbf{r9} \land \mathbf{ref}(a +_{32} n) =_{32} 0 \\ \land\, l =_{32} \mathbf{LR} \\ \land\, \forall(\lambda y : \mathbf{r3} >_{32} y \Rightarrow \neg\mathbf{ref}(a +_{32} y) =_{32} 0) \end{cases}$$

The register **CRF** does not occur in the assertions *Pre*, *Inv* and *Post* nor is it used as a value in the commands of Figure (5.14). For clarity, the assignments to the register **CRF** made by the commands will be removed from the substitution expressions which occur in the proof.

The proof is in two steps: the first considering the precondition of the program and the second considering the invariant. There are two cases for each step. If the program begins with the null character in the first element of a, $\mathbf{ref}(\mathbf{r3}) =_{32} 0$, then the program establishes the postcondition. Otherwise, the program is shown to establish the invariant. The proof for both cases is similar and only the second will be considered. For the second step, the proof is by induction and the base case begins in a state in which $\mathbf{r3} +_{32} 1$ is the address of the null character. For this case, the invariant is shown to establish the postcondition directly. In the inductive case, $\mathbf{ref}(\mathbf{r3} +_{32} 1)$ is not the null character and the proof is by establishing the inductive hypothesis. The verification of len_2 is in the following steps:

1. Precondition establishes invariant:
   $\vdash [\mathit{Pre}(a, n, l) \land \mathit{pc} =_{32} l_1 \land \neg(\mathbf{ref}(\mathbf{r3}) =_{32} 0)]\mathit{len}_2[\mathit{Inv}(n, a, n, l) \land \mathit{pc} =_{32} l_5]$
2. Invariant establishes postcondition:
   $\vdash [\mathit{Inv}(d, a, n, l) \land \mathit{pc} =_{32} l_6]\mathit{len}_2[\mathit{Post}(a, n, l) \land \mathit{pc} =_{32} \mathbf{loc}(\mathbf{LR})]$

   The proof is by induction, in the following steps:

   (a) Invariant establishes postcondition: (Base case):
   $\vdash [\mathit{Inv}(d, a, n, l) \land \mathit{pc} =_{32} l_6 \land \mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1) =_{32} 0]\mathit{len}_2[\mathit{Post}(a, n, l) \land \mathit{pc} =_{32} \mathbf{loc}(\mathbf{LR})]$

   (b) Invariant establishes postcondition (Inductive case):
   $\vdash [\mathit{Inv}(d, a, n, l) \land \mathit{pc} =_{32} l_6 \land \neg\mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1) =_{32} 0]\mathit{len}_2[\mathit{Inv}(d-1, n, a, l) \land \mathit{pc} =_{32} l_2]$

The proof for last step is representative of the use of the proof rule for induction (tl9) and will be given here. The proofs for the remaining steps are similar.

Invariant establishes postcondition, inductive case (Step 2b): $\vdash [\mathit{Inv}(d,a,n,l) \land pc =_{32} l_6 \land \neg\mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1) =_{32} 0]\mathit{len}_2[\mathit{Inv}(d-1,n,a,l) \land pc =_{32} l_2]$

The command of len_2 enabled when $pc =_{32} l_6$ is $T_1(r_2); c_{10}$ and the proof is based on the weakest precondition of the command.

$$\begin{aligned}&\vdash \mathit{Inv}(d,a,n,l) \land pc =_{32} l_6 \land \neg\mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1) =_{32} 0\\ &\quad\Rightarrow \mathbf{wp}((T_1(r_2); c_{10}), \mathit{Inv}(d-1,n,a,l) \land pc =_{32} l_2)\end{aligned}$$

From the assumption that $\mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1) =_{32} 0$ is *false* and the conditional rule (tl3), the true branch of the command is chosen. This is an assignment command and, removing the assignment to **CRF**, the assertion needed for the weakening rule (tl5) and assignment rule (tl1) is the result of updating the invariant with the assignments.

$$\begin{aligned}&\vdash (\mathit{Inv}(d,a,n,l) \land pc =_{32} l_6 \land \neg(\mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1) =_{32} 0))\\ &\quad\Rightarrow\\ &\quad \mathit{Inv}(d-1,a,n,l) \land pc =_{32} l_6) \lhd (pc, l_6) \cdot (\mathbf{r3}, \mathbf{r3} +_{32} 1) \cdot (\mathbf{r0}, \mathbf{ref}(\mathbf{r9} +_{32} \mathbf{r3} +_{32} 1))\end{aligned}$$

The proof of this assertion is straightforward from the definitions. Since $d-1 < d$, the inductive hypothesis establishes the postcondition (Induction rule, tl9).

$$\vdash [\mathit{Inv}(d-1,a,n,l) \land pc =_{32} l_6]\mathit{len}_2[\mathit{Post}(a,n,l) \land pc =_{32} \mathbf{LR}]$$

This completes the proof for this case.

By showing the precondition and invariant establish the postcondition in all cases, the program len_2 is verified. Since the program len_2 is an abstraction of the programs len_1 and *len*, both of these are also verified. □

The verification of program *len* followed the approach described in Section 4.4: an abstraction of the program is constructed and shown to satisfy the specification. This approach is independent of the processor language and was also used to verify the programs of the M68000. It is therefore an example of the use of the language $\mathcal{L}$ to generalise methods for verifying object code across different processor languages. The approach of Section 4.4 can be used to verify the different object code programs of different processor languages, provided that the object code is described in terms of the language $\mathcal{L}$

5.3.4 Example: Division

The PowerPC processor is based on RISC design, which encourages the use of registers rather than memory locations. A program for the PowerPC will therefore make greater use of the registers than the equivalent program for a processor such as the M68000, which is based on a CISC design. However, these differences do not affect the approach used to verify a program. For an

.div:	cmplw 1,3,4	1_1 :	$\mathbf{CRF} := \mathbf{calcCRF}(\mathbf{r4} -_{32} \mathbf{r3}), l_2$
	li 0,0	l_2 :	$\mathbf{r0} := \mathbf{Long}(0), l_3$
	bc 4,5,L..8	l_3 :	$\textbf{if not bit}(1)(\mathbf{CRF})$ **then goto** l_8 **else goto** l_4
L..9:	subf 3,4,3	l_4 :	$\mathbf{r3} := \mathbf{r3} -_{32} \mathbf{r4}, l_5$
	cmplw 1,3,4	l_5 :	$\mathbf{CRF} := \mathbf{calcCRF}(\mathbf{r4} -_{32} \mathbf{r3}), l_6$
	addic 0,0,1	l_6 :	$\mathbf{r0} := \mathbf{r0} +_{32} 1, l_7$
	bc 12,5,L..9	l_7 :	$\textbf{if bit}(1)(\mathbf{CRF})$ **then goto** l_4 **else goto** l_8
L..8:	stw 3,0(5)	l_8 :	$:= \mathbf{writel}(\mathbf{r3}, \mathbf{r5}), l_9$
	mr 3,0	l_9 :	$\mathbf{r3} := \mathbf{r0}, l_{10}$
	blr	l_{10} :	$\textbf{goto loc}(\mathbf{LR})$

PowerPC program $\mathcal{L}$ program *ppcdiv*

Figure 5.15: Division: PowerPC

example of this, consider the C program of Figure (5.1) for the division of natural numbers. This was compiled to produce the object code program for the PowerPC processor of Figure (5.15) which was then translated to the $\mathcal{L}$ program *ppcdiv* of Figure (5.15), by replacing processor instructions with their equivalent $\mathcal{L}$ commands.

The parameters to program *ppcdiv* are stored in the processor registers as is the result of executing the program. This is contrast to program *m68div*, for the M68000 processor, in which the parameters are stored on the machine stack (implemented by memory variables). As with program *m68div*, the specification and properties to be established by *ppcdiv* are those of program *idiv*. The only differences are in the variables used by the programs. In program *ppcdiv*, the C variable n is stored in register **r3**, variable d is stored in register **r4** and variable r is register **r5**. When the program ends, the result must be stored in register **r0** and control must pass to the label l stored in the link register **LR**. The precondition of *ppcdiv* is the assertion $\mathit{ppcPre}(n, d, a, l)$ (where $n, d, a, l \in \mathbb{N}$). The postcondition is the assertion $\mathit{ppcPost}(n, d, a, l)$.

$$\begin{aligned}\mathit{ppcPre}(n, d, a, l) &\stackrel{\text{def}}{=} d >_a 0 \land n \geq_a 0 \land \mathbf{r3} =_{32} n \land \mathbf{r4} =_{32} d \land \mathbf{r5} =_{32} a \land \mathbf{LR} =_{32} l \\ \mathit{ppcPost}(n, d, a, l) &\stackrel{\text{def}}{=} n \geq_a 0 \land d >_a 0 \land \mathbf{LR} =_{32} l \land \mathbf{r5} =_{32} a \\ &\qquad \land n =_{32} (\mathbf{r0} \times_{32} d) +_{32} \mathbf{readl}(\mathbf{r5})\end{aligned}$$

The specification of program *ppcdiv* is, for any $n, d, a, l \in \mathbb{N}$:

$$\vdash [pc =_{32} l_1 \land \mathit{ppcPre}(n, d, a, l)]\mathit{ppcdiv}[pc =_{32} l \land \mathit{ppcPost}(n, d, a, l)]$$

The verification of *ppcdiv* follows the steps used for *idiv* and is based on the loop in *ppcdiv*

$$
\begin{array}{l}
C_1 = l_1 : \textbf{if not } \mathbf{r4} >_{32} \mathbf{r3} \\
\qquad\quad \textbf{then } \mathbf{CRF}, \mathbf{r0} := \mathbf{calcCRF}(\mathbf{r4} -_{32} \mathbf{r3}), \mathbf{Long}(0), l_8 \\
\qquad\quad \textbf{else } :=(\mathbf{CRF}, \mathbf{calcCRF}(\mathbf{r4} -_{32} \mathbf{r3})) \cdot (\mathbf{r3}, \mathbf{r0}) \cdot \mathbf{writel}(\mathbf{r3}, \mathbf{r5}), \mathbf{loc}(\mathbf{LR})
\end{array}
$$

$$
\begin{array}{l}
C_2 = l_8 : \textbf{if } \mathbf{r4} >_{32} (\mathbf{r3} -_{32} \mathbf{r4}) \\
\qquad\quad \textbf{then } \mathbf{r3}, \mathbf{CRF}, Br0 := \mathbf{r3} -_{32} \mathbf{r4}, \mathbf{calcCRF}(\mathbf{r4}, \mathbf{r3} -_{32} \mathbf{r4}), \mathbf{r0} +_{32} 1, l_4 \\
\qquad\quad \textbf{else } :=(\mathbf{r3}, \mathbf{r3} -_{32} \mathbf{r4}) \cdot (\mathbf{CRF}, \mathbf{calcCRF}(\mathbf{r4} -_{32} \mathbf{r3})), (\mathbf{r3}, \mathbf{r0}) \\
\qquad\qquad\quad \mathbf{writel}(\mathbf{r3} -_{32} \mathbf{r4}, \mathbf{r5}), \mathbf{loc}(\mathbf{LR})
\end{array}
$$

Figure 5.16: Commands of Abstraction *ppcdiv*$_2$

beginning at label l_4. The properties of the loop are established by induction on the value of register **r3**, using proof rule (tl9). The invariant for the loop is the assertion *ppcInv*(q, n, d, a, l):

$$
\mathit{ppcInv}(q, n, d, a, l) \stackrel{\text{def}}{=} \begin{cases} n \geq_a 0 \land d >_a 0 \land q =_{32} \mathbf{r3} \land l =_{32} \mathbf{LR} \\ \land a =_{32} \mathbf{r5} \land d =_{32} \mathbf{r4} \land (\mathbf{r3} \times_{32} d) +_{32} \mathbf{r0}) =_{32} n \\ \land (pc =_{32} l \Rightarrow \mathbf{readl}(\mathbf{r5}) =_{32} \mathbf{r3}) \end{cases}
$$

As with program *m68div*, the verification of *ppcdiv* is similar to that of *idiv* and will not be given here. However, the abstraction of program *ppcdiv* and the steps required to verify the abstraction will be described.

Abstraction of *ppcdiv*

Because program *ppcdiv* has a computed jump command (at label l_{10}), the abstraction of *ppcdiv* is in two parts. The first constructs an abstraction *ppcdiv*$_1$ by applying the general transformation T_2 to the region r containing all commands except the computed jump. The commands of *ppcdiv* labelled $l_1, \ldots, l_{10}$ will be referred to as $c_1, \ldots, c_{10}$. Region r begins with command c_1, $r = \mathit{region}(l_1, \mathit{ppcdiv} - \{c_{10}\})$. Applying the general transformation $T_2(r)$ constructs two regions r_1 and r_2, beginning with c_1 (the head of r) and c_4 (the head of a loop in r): $r_1 = \mathit{region}(l_1, \{c_1, c_2, c_3, c_8, c_9\})$ and $r_2 = \mathit{region}(l_4, \{c_4, c_5, c_6, c_7, c_8, c_9\})$. The path transformation T_1 is applied to both these regions and the result of the general transformation is a region containing command $T_1(r_1)$ and $T_1(r_2)$. These are combined with *ppcdiv* to form the first abstraction: $\mathit{ppcdiv}_1 = \mathit{ppcdiv} \uplus \{T_1(r_1), T_1(r_2)\}$.

The two commands $T_1(r_1)$ and $T_1(r_2)$ are composed with c_{10} to form the second abstraction *ppcdiv*$_2$. Let $C_1 = (T_1(r_1); c_{10})$ and $C_2 = (T_1(r_2); c_{10}))$. Program *ppcdiv*$_2$ is obtained by combining C_1 and C_2 with *ppcdiv*$_1$: $\mathit{ppcdiv}_2 = \mathit{ppcdiv}_1 \uplus \{C_1, C_2\}$. The commands C_1 and C_2 after simplification are given in Figure (5.16). Program *ppcdiv*$_2$ is an abstraction of *ppcdiv*, $\mathit{ppcdiv}_2 \sqsubseteq \mathit{ppcdiv}$ and verifying *ppcdiv*$_2$ is enough to establish the correctness of *ppcdiv*.

Verification of $ppcdiv_2$

The specification which must be satisfied by $ppcdiv_2$ is that of *ppcdiv*:

$$\vdash [pc =_{32} l_1 \wedge ppcPre(n, d, a, l)]ppcdiv_2[pc =_{32} l \wedge ppcPost(n, d, a, l)]$$

The steps required to show the correctness of $ppcdiv_2$ are those required for *idiv*:

1. Precondition establishes postcondition:

$$\vdash [pc =_{32} l_1 \wedge ppcPre(n, d, a, l) \wedge n <_{32} d]ppcdiv_2[pc =_{32} l \wedge ppcPost(n, d, a, l)]$$

   This is established by command C_1: $\vdash (pc =_{32} l_1 \wedge ppcPre(n, d, a, l) \wedge n <_{32} d)$
   $\Rightarrow \mathbf{wp}(C_1, pc =_{32} l \wedge ppcPost(n, d, a, l))$

2. Precondition establishes invariant:

$$\vdash [pc =_{32} l_1 \wedge ppcPre(n, d, a, l) \wedge \neg n <_{32} d]ppcdiv_2[pc =_{32} l_4 \wedge ppcInv(n, n, d, a, l)]$$

   This requires: $\vdash (pc =_{32} l_1 \wedge ppcPre(n, d, a, l) \wedge \neg n <_{32} d)$
   $\Rightarrow \mathbf{wp}(C_1, pc =_{32} l_4 \wedge ppcInv(n, n, d, a, l))$

3. Invariant establishes postcondition, the proof is by induction on $q \in \mathbb{N}$ (rule tl9):

$$\vdash [pc =_{32} l_4 \wedge ppcInv(q, n, d, a, l)]ppcdiv_2[pc =_{32} l \wedge ppcPost(n, d, a, l)]$$

   The two cases can be established from the command C_2:

   (a) Base case, $\mathbf{r3} -_{32} \mathbf{r4} <_{32} \mathbf{r4}$:

$$\vdash (pc =_{32} l_4 \wedge ppcInv(q, n, d, a, l) \wedge (\mathbf{r3} -_{32} \mathbf{r4}) <_{32} \mathbf{r4}) \\ \Rightarrow \mathbf{wp}(C_2, pc =_{32} l \wedge ppcPost(n, d, a, l))$$

   (b) Inductive case, $\neg\mathbf{r3} -_{32} \mathbf{r4} <_{32} \mathbf{r4}$:

$$\vdash (pc =_{32} l_4 \wedge ppcInv(q, n, d, a, l) \wedge \neg(\mathbf{r3} -_{32} \mathbf{r4}) <_{32} \mathbf{r4}) \\ \Rightarrow \mathbf{wp}(C_2, pc =_{32} l \wedge ppcPost(n, d, a, l))$$

These steps are similar to those needed to verify program *m68div* for the M68000 processor. The greatest difference between programs *ppcdiv* and *m68div*, apart from the variables used, is the number of commands in the program. The M68000 program *m68div* has more commands than *ppcdiv* since it must transfer the parameters of the program from memory to registers. However, the verification of both programs is based on constructing abstractions of the programs and only the paths between the cut-points of the program are considered when abstract a program. Since both program *m68div* and program *ppcdiv* have similar flow-graphs (based around a single loop), both have the same number of cut-points. Consequently, the same number of commands are considered during the verification of both program $m68div_2$ and program $ppcdiv_2$.

5.4 Other Features of Processor Languages

Processor languages often include features which are not easily reasoned about or manipulated. These include instructions or data operations which are intended for a particular application or an execution model in which the flow of control is always determined by label expressions. An instruction which performs a complex operation can result in the need to reason about a large number of basic expressions. A processor can organise the registers or memory variables in such a way that a name is always determined by calculating an expression. This will lead to reasoning about a large number of expressions when verifying a program. A complex model for selecting instruction can complicate the abstraction of a program and require additional techniques to determine the flow of control through a program.

The models, in the language $\mathcal{L}$, of a class of specialised instructions, a method for organising the registers and an execution model based on label expressions will be described. The instructions implement operations to copy many items of data between names. The method of organisation of registers is similar to that of the SPARC processor (Weaver & Germond, 1994) and requires registers to be identified by name expressions. The execution model is for delayed execution of instructions and the example is also based on the selection rules of the SPARC processor.

5.4.1 Multiple Data Move Instructions

A multiple data move instruction copies a number of items of data between locations in memory or between memory and the registers. The operation performed is similar to a simultaneous assignment of $i > 0$ values to i names. The number of data items to be moved may be determined by the value of a register or memory variable. Typical applications of multiple move instructions include storing and restoring the values of registers and copying sections of memory.

An instruction which moves data between the registers and the memory variables will typically assign the values of a subset of the registers to consecutive locations in memory. The registers to be copied are calculated from a source argument *src* and a destination argument *dst* identifies the address of the first location in memory. The contents of the registers are copied to the memory locations in a defined order, allowing the implementation of the reverse operation, copying the contents of the locations to the registers, to be simplified.

Instructions which copy data between locations in memory obtain the location of the first item to be copied from a source argument src_1, the number of items to be copied from a source argument src_2 and the location in which the first item is to be stored from a destination argument *dst*. The data is copied between location by a series of assignments of the form (for byte sized operations) $\mathbf{ref}(src_2 +_a f_2(i)) := \mathbf{ref}(src_1 +_a f_1(i))$ where i is the index of the ith element, $f_1(i)$ is the location relative to src_1 from which the value is obtained and $f_2(i)$ the location relative to src_2 to which the value is assigned.

The semantics of a multiple move instruction can be defined as a single assignment command in which the assignments are determined from the arguments to the instruction. Alternatively the instruction can be defined by iteration, with each assignment calculated from a decreasing index. The first method, although complex, results in a more accurate model of the behaviour of the instruction. The second method results in a loop in an $\mathcal{L}$ program which is not present in the equivalent object code.

Both approaches cause problems when verifying the program since proof rules depend on the substitution in an assertion of the assignments made by a command. Because the multiple data move instruction performs a large number of assignments, the task of performing the substitutions manually becomes difficult. This problem will occur however the semantics of the instruction are defined, because the basic operation is to make a large number of assignments. However, once the semantics of the instruction are defined it is possible to derive proof rules for the particular instruction from the standard rules for assignment command. These may be used in the construction of automated tools which are specialised for the particular processor language and which simplify the manipulation of the instructions.

Example 5.1 For an example of the definition of a multiple data move instruction using the first method, assume a processor language in which there are 32 registers **r0**, ... , **r31**. Also assume a multiple move instruction, with operation size *Byte*, which copies i consecutive registers, beginning with **r0**, to consecutive locations in memory. The source argument, *src*, of the instruction determines the number of registers to be moved, $\mathit{src} = i$. The destination argument, *dst*, is the address of the first location in memory.

The semantics of the instruction will be defined as a single assignment command in which a name is assigned a value conditional on the result of a test. The conditional expression is defined, in Appendix A of the appendix, as the value function with name **cond** and arity 3. For expressions $b, e_1, e_2 \in \mathcal{E}$ and state s, $\mathbf{cond}(b, e_1, e_2)$ satisfies:

$$\begin{array}{ll} \mathbf{cond}(b, e_1, e_2) \equiv_s e_1 & \text{if } b \equiv_{(\mathcal{I}_b, s)} \mathbf{true} \\ \mathbf{cond}(b, e_1, e_2) \equiv_s e_2 & \text{if } b \equiv_{(\mathcal{I}_b, s)} \mathbf{false} \end{array}$$

The assignment list of the command is constructed from assignments to individual locations. The first register is stored in location *dst*, the second at $\mathit{dst} +_{32} 1$, etc. Each assignment is conditional on the value of a decreasing argument: the ith register is stored in memory if the result of $(\mathit{src} \bmod 32) + 32 - i$ is greater than 0. With successor expression $l \in \mathcal{E}_l$, the assignment command for the multiple move is:

$$\begin{array}{l} := ((\mathbf{ref}(\mathit{dst} +_{32} 0), \mathbf{cond}((\mathit{src}\ \mathrm{mod}_a\ 32) +_a 32 -_a 0) >_{32} 0, \mathbf{r0}, \mathbf{ref}(\mathit{dst} +_{32} 0)) \\ \quad \cdot (\mathbf{ref}(\mathit{dst} +_{32} 1), \mathbf{cond}((\mathit{src}\ \mathrm{mod}_a\ 32) +_a 32 -_a 1) >_{32} 0, \mathbf{r1}, \mathbf{ref}(\mathit{dst} +_{32} 1)) \\ \quad \cdot (\mathbf{ref}(\mathit{dst} +_{32} 2), \mathbf{cond}((\mathit{src}\ \mathrm{mod}_a\ 32) +_a 32 -_a 2) >_{32} 0, \mathbf{r3}, \mathbf{ref}(\mathit{dst} +_{32} 2)) \\ \qquad \vdots \\ \quad \cdot (\mathbf{ref}(\mathit{dst} +_{32} 31), \mathbf{cond}((\mathit{src}\ \mathrm{mod}_a\ 32) +_a 32 -_a 31) >_{32} 0, \mathbf{r31}, \mathbf{ref}(\mathit{dst} +_{32} 31)), \\ \quad l) \end{array}$$

The effect of assignment $(\mathbf{ref}(\mathit{dst}+_{32} i), \mathbf{cond}((\mathit{src} \bmod_a 32) +_a 32 -_a i) >_{32} 0, \mathbf{r}_i, \mathbf{ref}(\mathit{dst}+_{32} i))$ is to update the ith destination location with the ith register, $\mathbf{r}_i$, if the register is to be copied. If the register is not to be copied, the condition, $\mathbf{cond}((\mathit{src} \bmod_a 32) +_a 32 -_a i) >_{32} 0$, is *false* and the location is assigned its original value. The definition of the instruction will therefore construct an assignment list with an assignment for each of the 32 locations beginning at address *dst*. The first *src* locations are assigned the value of the first *src* registers and the remaining locations are assigned their original value. □

5.4.2 Organisation of Registers

The organisation of a processor's registers can include the use of expressions to calculate the register to be used by an instruction. The registers used by an instruction are referred to in terms of a *register function*, a function ranging over the names constructed from elements of the set *Regs*. Register functions are a subset of the name expressions, $\mathcal{E}_n$, and the verification techniques for a processor languages with register functions are those used for a language without.

Example 5.2 The *register windows* of the SPARC processor (Weaver & Germond, 1994) are an example of registers organised using expressions to determine the register to be used.

For a register model similar to that of the SPARC processor, assume $\mathit{Values} = \mathit{Vars} = \mathbb{N}$ and that the registers, here called the *processor registers*, are identified by the set of negative integers.

$$\mathit{Regs} \stackrel{\text{def}}{=} \{x : \mathbb{Z} \mid x < 0\}$$

The registers are distinct from the values, and therefore the variables, since $\mathit{Values} = \mathbb{N}$. This ensures that a name $x \in \mathit{Names}$ constructed from a variable of *Vars* is distinct from a name constructed from the registers.

There are eight *global registers*: $\mathbf{g0}, \mathbf{g1}, \ldots, \mathbf{g7}$. Registers $\mathbf{g0}, \ldots, \mathbf{g7}$ are defined as the names $\mathit{name}(-2), \mathit{name}(-3), \ldots, \mathit{name}(-9)$ respectively. A *register window* contains 24 registers and there are *nrw*, $3 \leq \mathit{nrw} \leq 32$, register windows. The *current window pointer*, *cwp*, is the name $\mathit{name}(-1)$ and has the range of values $0, \ldots, \mathit{nrw}$. The current window pointer identifies which of the register windows is referred to by an instruction. An instruction may use either a global register or one of the 24 *window registers* of the register window referred to by *cwp*; the global registers and the window registers are distinct.

Window registers are mapped to processor registers by an index relative to *cwp*. For $0 \leq i \leq 23$, the ith window register relative to *cwp* is the processor register $-(9+((\mathit{cwp} \bmod \mathit{nrw}) \times 24)+i)$; calculated from the 8 global registers, the *cwp* register and 24 window registers. This is defined in terms of $\mathcal{L}$ as a name function *reg* with arity 2; the first argument is the window register $i \in \mathit{Values}$ and the second the value $v \in \mathit{Values}$ of *cwp*.

$$\begin{array}{l} \mathit{reg} \in \mathcal{F}_n \\ \mathcal{I}_f(\mathit{reg})(i, v) \stackrel{\text{def}}{=} \begin{cases} \mathit{name}(-(9 + ((v \bmod \mathit{nrw}) \times 24) + i)) & \text{if } i < 24 \\ \mathit{name}(\mathit{undef}(\mathit{Regs})) & \text{otherwise} \end{cases} \end{array}$$

The processor register referred to by the ith window register is obtained by the name expression $\mathit{reg}(i, \mathit{cwp})$.

A register window is changed by addition and subtraction on the register *cwp* (modulo the number of windows). e.g. The *save* and *restore* instructions of the SPARC processor perform addition on window registers 1 and 2, alter the register window then store the result of the addition in register 3 of the new window. The save instruction can be modelled by incrementing the register window pointer:

$$\mathit{cwp}, \mathit{reg}(3, (\mathit{cwp} +_a 1) \bmod_a \mathit{nrw})) := (\mathit{cwp} +_a 1) \bmod_a \mathit{nrw}, \mathit{reg}(1, \mathit{cwp}) +_a \mathit{reg}(2, \mathit{cwp})$$

The restore instruction is modelled by decrementing the register window pointer:

$$\mathit{cwp}, \mathit{reg}(3, (\mathit{cwp} -_a 1) \bmod_a \mathit{nrw}) := (\mathit{cwp} -_a 1) \bmod_a \mathit{nrw}, \mathit{reg}(1, \mathit{cwp}) +_a \mathit{reg}(2, \mathit{cwp})$$

An instruction refers to a global register or to a window register. For example, the addition of global register **g0** to the 8th window register, storing the result in the 0th window register, can be modelled as an assignment command with the registers identified by *reg*:

$$\mathit{reg}(0, \mathit{cwp}) := \mathbf{g0} +_a \mathit{reg}(8, \mathit{cwp})$$

If this is followed by a save or a restore instruction, the register referred to by $\mathit{reg}(0, \mathit{cwp})$ will differ from that assigned to by the addition instruction.

The window register model defined by the SPARC processor language is based on overlapping windows. The ith window shares 8 registers with each of the $i + 1$th window and the $i - 1$th window. This can be modelled by defining the function *reg* to map the i window register to the processor register identified by $-_a(9 + ((\mathit{cwp} \bmod \mathit{nrw}) \times 16) + i)$.

$$\mathit{reg}(i, v) = \mathit{name}(-(9 + ((\mathit{cwp} \bmod \mathit{nrw}) \times 16) + i))$$

A full description of register windows and their use is given in the SPARC processor manual (Weaver & Germond, 1994). □

5.4.3 Delayed Instructions

The execution model of a processor language can allow the execution of an instruction to be delayed. An instruction is *delayed* if the changes made to the state by the instruction do not take effect until after the execution of one or more other instructions; typically the delay is for one instruction (see Hennessy & Patterson, 1990). There are two basic types of delayed instruction: a *delayed load* and a *delayed branch*.

A delayed load assigns a value v to a name x and the assignment is delayed until the instruction immediately following the load has finished execution. The implementation of a delayed load is such that an object program can be modelled in the same way as an object program with no delayed load instructions. For example, a processor language can require that the instruction immediately following a delayed load does not use any register or variable assigned a value by the load. A delayed load of v to name x, $x := v$, followed by an instruction $x_1 := f(x)$ would cause an error and would not appear in an object program. Other processors may implement the delayed load by inserting a null instruction after the load. In either case, a delayed load $x := v$ followed by instruction $x_1 := f(x)$ would behave as if the load was not delayed.

Delayed Branching

In a delayed branch, a jump to a target at location l is delayed until after the instruction immediately following the branch has been executed. The execution of the sequence $(l_1 : \textbf{goto}\ l), (l_2 : c)$ is in the order $(l_1 : c), (l_2 : \textbf{goto}\ l)$: instruction c is executed before the delayed branch command.

An example of a language with delayed branching is that of the SPARC processor (Weaver & Germond, 1994). The execution model of the SPARC processor is based on delaying the execution of most instructions and, as a consequence, the majority of jump instructions in the SPARC processor language are delayed. The execution model is based on two program counters: the *program counter*, *pc*, stores the label of the currently executing command and the *next program counter*, *npc*, stores the label of the successor command. If $l_1 : c_1$ is the currently executing command and $l_2 : c_2$ the next command to be executed then $pc =_a l_1$ and $npc =_a l_2$. Each instruction updates both program counters: for instructions which are not jump commands, *pc* is assigned *npc* and *npc* is assigned the label of the next instruction in memory. Each instruction of the SPARC processor is stored in 4 bytes and *npc* is assigned the expression $npc + 4$.

For example, let three instructions make the assignments $x_1 := v_1$, $x_2 := v_2$ and $x_3 := v_3$ and assume labels l_1, l_2, l_3, l_4, l_5 and initial values $pc =_a l_1$ and $npc =_a l_2$. The instructions together with the assignments to the program counters form the sequence:

$$\begin{array}{l} l_1 : x_1, npc, pc := v_1, npc +_a 4, npc \\ l_2 : x_2, npc, pc := v_2, npc +_a 4, npc \\ l_3 : x_3, npc, pc := v_3, npc +_a 4, npc \end{array}$$

The instructions are executed in the order l_1, l_2 then l_3.

When an instruction is a jump to a target l, the register *npc* is generally assigned the label l of the target. Let **jump** be the command defined:

$$\textbf{jump}(l) \stackrel{\text{def}}{=} npc, pc := l, npc$$

Because l is assigned to the next program counter, *npc*, the jump is delayed by one instruction. For example, assume initial values $pc =_a l_1$, $npc =_a l_2$. The sequence:

$$\begin{array}{l} l_1 : \textbf{jump}(l) \\ l_2 : x_2, npc, pc := v_2, npc +_a 4, npc \end{array}$$

is executed in the order l_1, l_2, l. After the command at l_1 is executed, *pc* has the value l_2 and *npc* the value l. The command at label l_2 is executed and assigns *npc* to *pc* passing control to the command at label l.

The SPARC processor language includes jump instructions which *annul* the delay: control is immediately transfered to the target. Let **jumpi** be the immediate jump command defined:

$$\textbf{jumpi}(l) \stackrel{\text{def}}{=} \mathit{npc}, \mathit{pc} := l +_a 4, l$$

After execution of $\textbf{jumpi}(l)$, the program counter is assigned the target of the jump and the next program counter is assigned the label of the instruction following the target. The sequence of commands:

$$\begin{array}{l} l_1 : \textbf{jumpi}(l) \\ l_2 : x_2, \mathit{npc}, \mathit{pc} := v_2, \mathit{npc} +_a 4, \mathit{npc} \end{array}$$

is executed in the order l_1, l. Control passes to the command labelled l and the command at label l_2 is not executed.

Simplifying Programs

The use of two program counters means that the majority of instruction select a successor by the use of the label expression *npc*. Such an instruction, c_1, can always reach another instruction c_2, $c_1 \longmapsto c_2$, since there is always a state in which $\mathit{npc} =_a \mathit{label}(c_2)$. As a consequence, applying the transformation T_2 to a SPARC object code program will have no effect since every instruction is potentially a cut-point of a region. This will occur however the *reaches* relation is defined: to decide whether c_1 can reach c_2, the values which may be assigned to *npc* must be known and this is a function of the state in which c_1 begins.

An abstraction of a program can be obtained, without applying transformations T_1 or T_2, by the sequential composition to individual commands of the program. For example, if a program p contains the commands:

$$\begin{array}{l} l_1 : \textbf{jump}(l) \\ l_2 : x_2, \mathit{npc}, \mathit{pc} := v_2, \mathit{npc} +_a 4, \mathit{npc} \end{array}$$

then a program p' such that $p' \sqsubseteq p$ can be obtained by replacing the command at l_1 with the composition of the two commands:

$$l_1 : (\textbf{jump}(l); l_2 : x_2, \mathit{npc}, \mathit{pc} := v_2, \mathit{npc} +_a 4, \mathit{npc})$$

Additional or alternative transformations may be defined which are specialised to the execution model. These can be applied to regions of a program either to construct an abstraction of the program or to construct a region to which transformation T_2 can be applied. Such transformations can exploit the fact that the *npc* is a program counter and its value must be known when the program begins.

When the initial value of *npc* is known, a transformation based on *constant folding* (Aho et al., 1986) can be used to determine the value of *npc* for a subset of the program. The initial value l of *npc* is substituted into the first command c of the program, at label *pc*. If the expressions assigned to *pc* and *npc* by command c do not contain any other name then they may be reduced to constants. These provide new values for *pc* and *npc* and the process is repeated until an expression which depends on a name other than *pc* or *npc* occurs. The result of this transformation is a subset of the program in a form to which T_2 can be applied.

For example, if the initial value are $pc =_a l_1$ and $npc =_a l_2$ then the commands:

$$\begin{array}{l} l_1 : x_1, npc, pc := v_1, npc +_a 4, npc \\ l_2 : x_2, npc, pc := v_2, npc +_a 4, npc \\ l_3 : x_3, npc, pc := v_3, npc +_a 4, npc \end{array}$$

can be replaced with the commands:

$$\begin{array}{l} l_1 : x_1, npc, pc := v_1, l_3, l_2 \\ l_2 : x_2, npc, pc := v_2, l_3 +_a 4, l_3 \\ l_3 : x_3, npc, pc := v_3, l_3 +_a 8, l_3 +_a 4 \end{array}$$

Since l_2 and l_3 are constants, the relation *reaches* will describe the control flow through the program. A region of the program can then be constructed to which the transformation T_2 can be applied to obtain an abstraction of the region. Other techniques for transforming a program include those used in code optimisation. These can be applied to programs of the SPARC processor language and, therefore, may be used to determine the possible flow of control through a program in a form suitable for transformations T_1 and T_2. Such techniques would be a necessary part of an automated tool which mechanised the abstraction of object programs with an execution model similar to that of the SPARC processor.

5.5 Conclusion

The processor architecture defines the instructions and data operations which are available for use in an object code program. Any instruction can be modelled as a single commands of $\mathcal{L}$, since the semantics of every instruction can be described as a state transformer. The difficulty of verifying an object code program is therefore determined by the number of instructions in the program and by the data operations carried out by each instruction. Because the data operations of a processor are simple (by comparison with those of high-level languages), an object code program is made up of a large number of instructions. Abstracting from the $\mathcal{L}$ program modelling the object code reduces the number of commands which must be considered. Abstraction combines the operations performed by individual instructions, resulting in a single command of $\mathcal{L}$ performing the data operations of a sequence of instructions. The difficulty of a verification proof and the complexity of the $\mathcal{L}$ commands therefore depend on the data operations used by the instructions.

Two processor architectures were considered, representing the two commonly used designs. The Motorola 68000 is a CISC processor: it supports a large number of instructions which carry out complex data operations. The PowerPC is a RISC processor, it has relatively few instructions and provides simple data operations. Modelling the instructions and data operations of both processors in terms of $\mathcal{L}$ is straightforward. Since object code is translated into $\mathcal{L}$ by replacing each instruction with its model in $\mathcal{L}$, this demonstrates the ability of the language $\mathcal{L}$ to model the object code of different processors. The main difference between the processor languages is the number of operations carried out by an instruction. Instructions of the PowerPC carry out fewer operations than instructions of the M68000 and are therefore simpler to model and verify than those of the M68000. Not all features of a processor are straightforward to model in terms of $\mathcal{L}$, this is principally because the features are complex, rather than any deficiency in the language $\mathcal{L}$. For example, it will always be difficult to model and reason about a single instruction which moves large amounts of data.

While a processor language affects the approach used to implement a program it has little affect on the approach used to verify the program. A processor can provide instructions to support operations commonly used in object code, for example the `link` and `unlk` instructions of the M68000. Whether these instructions are used depends on the source of the object code. The programs considered in this chapter were all produced by a compiler. The design of the programs followed the programming conventions of the compiler and of the target operating system. These determined how the variables and operations of the C programs were implemented in the processor language and the structure of the programs. Although each processor had a different set of programming conventions, this did not affect the method used to verify the programs. All programs, whether a high-level or an object code program, are verified using the method of Floyd (1967) or Burstall (1974): by reasoning about the paths between program cut-points. For example, the implementations of natural division on the M68000 and the PowerPC processors (programs *m68div* and *ppcdiv*) both have a similar structure, based around a single loop. The verification of these programs is based on the structure of their flow-graphs and therefore follows a similar pattern for both programs. The difference in the data operations provided by the two processor languages does not affect the approach used to verify the two programs.

This chapter considered the verification of object code programs of different processor languages. The approach used to verify the programs in this chapter is based on the method of Burstall (1974) (and is described in Section 4.4). The same approach was used to verify each of the object code programs: the object code is modelled as a program of $\mathcal{L}$ and abstractions of the $\mathcal{L}$ program are constructed and verified. For the object code programs considered in this chapter, this approach greatly simplified the proof of correctness. For example, the first step in the verification of program *idiv* (of Figure 5.2) required seven different commands to be considered. The same step for the abstraction $idiv_2$ of *idiv* only required a single command to be considered. The approach used to verify the programs reflects the main benefit of the language $\mathcal{L}$: methods for verifying programs of $\mathcal{L}$ can be applied to any object code program described in terms of $\mathcal{L}$. In particular, the methods of abstracting programs of $\mathcal{L}$ can be applied to the programs of any processor language. This allows the object code programs of different processors to verified in a single program logic using the same approach to carry out the proof of correctness.

Chapter 6

Examples: Proof Methods for Object Code

Program verification is based on the method of inductive assertions (Floyd, 1967) or the method of intermittent assertions (Burstall, 1974). The application of these proof methods must take into account the features of the programs to be verified. Program verification generally considers the programs of high-level, structured languages only. The techniques used in program verification are therefore developed for programs with more restrictive data and execution models than object code programs. This means that features of object code may not be considered by the approach used to verify a program, making the task of verifying object code programs more difficult than necessary. It may also be desirable to develop techniques for verification which exploit features of object code programs which do not occur in high-level programs.

Techniques for verifying programs can be developed to exploit features of the processor language in which an object code program is written to simplify verification of the program. Many features of processor languages affect only the implementation of a program rather than the difficulty of verifying a program and this approach is unlikely to lead to generally applicable methods. A second approach is to develop verification methods which can be applied to all object code programs, independently of the processor language. In this approach, general proof methods are specialised to exploit some feature common to all object code programs. These methods are generally based on proof rules which can be applied during a proof of correctness. The language $\mathcal{L}$ supports this approach to developing verification methods by providing the operators needed to define proof rules which can be applied to $\mathcal{L}$ programs.

The main concern of this thesis is to be able to verify the object code programs of arbitrary processors and to simplify verification by abstracting from programs. This chapter is concerned with the secondary problem of the approach used to carry out the proof of correctness of object code. This chapter will consider features specific to object code which can affect verification and will also describe the development of methods for verifying object code. These methods will be developed for the language $\mathcal{L}$, which is a description language for the object code of different processors. This allows their application to any object code program. It also allows the main approach to simplifying verification (by abstracting from programs) to be used in conjunction with any additional techniques which may be developed.

The chapter begins with a description of features of object code programs in **Section 6.1**. This is followed by a description of two proof methods for programs of $\mathcal{L}$. The first, described in **Section 6.2**, is based on data refinement (Hoare, 1972; Morris, 1989), a standard technique in program verification. The second, described in **Section 6.3**, allows the verification of a program to be based on regions of the program. These proof methods will be described in terms of the language $\mathcal{L}$ and can be applied to the object code of any processor.

6.1 Features of Object Code Programs

The verification of an object code program must satisfy the same requirements as the verification of any program: the correctness of a program is proved by reasoning about the paths and loops in the program. The methods of Floyd (1967) and Burstall (1974) describe how such proofs are constructed. The principal requirements are that intermediate specifications are satisfied by loop-free sequences of commands and that the properties of loops are established by induction (Cousot, 1981). These requirements are independent of the data operations, the commands or the execution model of the programming language. The methods described by Floyd (1967) and Burstall (1974) are therefore independent of any processor language. However, the approach used to establish the correctness of a program can be affected by features of the program or of the programming language.

The principal features of a program which affect its verification are the data operations used by the program and the structure of the program's flow-graph. Object code uses the data operations and instructions provided by the processor architecture. The simplicity of these data operations means that a large part of an object code program is made up of sequences of instructions, which implement abstract data operations. These sequences can be reduced to a single command of $\mathcal{L}$, by abstracting from the program, and do not need any additional verification techniques. The structure of an object code program is made up of the paths and loops in the program's flow-graph. An object code program can include interleaving (irreducible) loops and sub-routines, both of which can complicate verification. The approach used to verify an object code program must therefore take into account the structure of object code. However, both the data operations provided by a processor and the structure defined by the flow-graph of an object code program can be used as the basis of a method for verifying object code.

6.1.1 Processor Architecture

A processor architecture determines the data operations and variables available to an object code program. The processor architecture will also support a particular approach to implementing programs, typically by optimising the performance of the processor for particular features. For example, if a processor provides a large number of registers then its programs will be designed to make greater use of the registers than the machine memory since register access is more efficient than accessing memory locations (Hennessy & Patterson, 1990). However, processor architectures do not affect the method used to verify a program. Verification is based on reasoning about the program variables at the program cut-points and these are not determined by the processor architecture or language. The processor only determines the data operations which must be considered when reasoning about instructions in a program.

Proof methods can be developed for particular processor architectures by the use of tools and techniques which are specialised to the data operations and instructions of a processor (e.g. see Yuan Yu, 1992). Such methods can simplify reasoning about the individual instructions of an object code program, by simplifying the $\mathcal{L}$ expressions representing the processor's data operations. However, this does not affect the difficulty of verifying a program. To simplify the verification of a program, the number of instructions considered during the verification must be reduced. This can be achieved by abstracting from the program, a method which is independent of any processor language.

Generalising Proofs Across Processors

The use of the language $\mathcal{L}$ to describe object code programs for verification means that verifying different programs for different processors is no different from verifying different programs for a single processor. However, it is possible to simplify the verification of a single program which is implemented across different processors. Assume a program p is implemented on two different processors as object code programs p_1 and p_2. The difference between programs p_1 and p_2 is the implementation of the data operations of program p using the operations provided by the two processors. If p is correct then separately verifying programs p_1 and p_2 leads to the repetition of work needed to show that the data operations establish the specification of p.

To show that the object code programs p_1 and p_2 are correct, it is only necessary to show that the data operations of p are correctly implemented by p_1 and p_2. Data refinement (Hoare, 1972; Morris, 1989) is a method which allows the correctness of p to be used to verify the object code programs p_1 and p_2. The approach is to show that object code program p_1 and p_2 data refine p by showing that p_1 and p_2 correctly implement the data operations of p. The correctness of p_1 and p_2 can then be established from the correctness of p. This approach is independent of processor architectures and can reduce the work needed when implementing a single program across different processors.

6.1.2 Loops in Programs

An object code program can contain single, nested and interleaving loops. Single and nested loops are found in high-level programs, each loop is either distinct from any other or contained entirely within another loop (Loeckx and Sieber, 1987). For example, the programs of Chapter 5 contain only single loops. The approach used to verify an object program containing single or nested loops is similar to that used for high-level programs (Dijkstra, 1976; Gries, 1981): inner loops are shown to establish an intermediate specification which is generalised over one or more program variables. This specification is then used to show that the outer loop establishes the specification which is used in the proof of correctness for the program.

For example, consider the following $\mathcal{L}$ program p_1 which contains nested loops:

$$\begin{array}{ll}
l_1: & \textbf{if } \mathbf{r0} =_a 0 \textbf{ then goto } l_6 \textbf{ else goto } l_2 \\
l_2: & \mathbf{r0}, \mathbf{r1} := \mathbf{r0} -_a 1, \mathbf{r0}, l_3 \\
l_3: & \textbf{if } \mathbf{r1} =_a 0 \textbf{ then goto } l_5 \textbf{ else goto } l_4 \\
l_4: & \mathbf{r1} := \mathbf{r1} -_a 1, l_3 \\
l_5: & \textbf{goto } l_1 \\
l_6: & \textbf{goto } l_6
\end{array}$$

The outer-most loop begins at label l_1 (of the $\mathcal{L}$ program), the inner loop begins at label l_3. The outer-most loop assigns **r0** to **r1** and decrements **r0**. The inner loop then repeatedly decrements **r1**, until $\mathbf{r1} = 0$, then passes control to the outer loop. To prove $\vdash [pc =_a l_1]p_1[pc =_a l_6]$, both loops must be shown to terminate. This requires a proof by induction on the values of **r0** and **r1**, which is carried out in two steps. The first shows that for any $m \in \mathbb{N}$, the inner loop establishes $\vdash [pc =_a l_4 \wedge \mathbf{r1} =_a m]p_1[pc =_a l_1 \wedge \mathbf{r1} =_a 0]$. This is used in the second step to show that the outer loop $\vdash [pc =_a l_1 \wedge \mathbf{r0} =_a n]p_1[pc =_a l_6 \wedge \mathbf{r0} =_a 0]$ for any $n \in \mathbb{N}$. The two steps can be carried out separately: the specification of the inner loop is independent of the outer loop.

In interleaving loops, two loops have commands in common but neither is entirely contained in the other. Interleaving loops are more difficult to verify than nested loops because the two loops (and their properties) depend on each other. For example, consider the following $\mathcal{L}$ program p_2:

$$\begin{array}{ll}
l_1: & \textbf{if } \mathbf{r0} =_a 0 \textbf{ then goto } l_6 \textbf{ else goto } l_2 \\
l_2: & \mathbf{r1} := \mathbf{r0}, l_3 \\
l_3: & \textbf{if } \mathbf{r1} =_a 0 \textbf{ then } \mathbf{r2} := 1, l_4 \textbf{ else } \mathbf{r2} := 0, l_4 \\
l_4: & \textbf{if } \mathbf{r2} =_a 1 \textbf{ then } \mathbf{r0} := \mathbf{r0} -_a 1, l_1 \textbf{ else goto } l_5 \\
l_5: & \mathbf{r1} := \mathbf{r1} -_a 1, l_3 \\
l_6: & \textbf{goto } l_6
\end{array}$$

Program p_2 has two loops: the first at the commands labelled l_1, l_2, l_3, l_4, the second at the commands labelled l_3, l_4, l_5. The loops are interleaved since the second loop passes control into the body of the first loop (from label l_5 to label l_3).

To verify interleaving loops, the properties of both loops must be considered at the same time. For example, assume that program p_2 must terminate when control reaches l_6: $\vdash [pc =_a l_1]p_2[pc =_a l_6]$. To verify p_2, both loops must be shown to terminate. The proof can be carried out by induction on the values of **r0** and **r1**, to establish the specification $\vdash [pc =_a l_1 \wedge \mathbf{r0} =_a n \wedge \mathbf{r1} =_a m]p_2[pc =_a l_1 \wedge \mathbf{r0} =_a 0 \wedge \mathbf{r1} =_a 1]$ (for any $n, m \in \mathbb{N}$). This will require two applications of the induction rule (tl9) for programs, the first for the value of **r0**, the second for the value of **r1**. This is a general approach, which can be used to establish the properties of interleaving loops in any program. The loops in individual programs can be simpler to verify. For example, in program p_2, the second loop (beginning at label l_3) can be verified independently of the first by showing that it preserves the value of **r0**.

6.1.3 Sub-Routines

Sub-routines are similar to the functions or procedures of high-level languages, implementing an operation which is used by different parts of an object code program but is not provided by the processor language (Wakerly, 1989). A sub-routine can be executed in many different circumstances during the execution of an object code program. Because the instructions making up a sub-routine are part of the program, verifying the program can require repeated proofs that the sub-routine establishes different properties. An alternative is to show that the sub-routine satisfies a specification describing the properties of the sub-routine. The verification of the program can then make use of this specification to derive the desired property directly, without the need to consider the instructions of the the sub-routine. In this approach the instruction passing control to the sub-routine will be selected as a cut-point of the program, in addition to the cut-points needed for the proof methods of Floyd (1967) and Burstall (1974).

Processors languages often include instructions which support the use of sub-routines. These instructions pass control to and from a sub-routine and may also support a particular approach to passing arguments to a sub-routine. For example, the M68000 processor includes the instructions `jsr` and `ret` to pass control to and from a sub-routine. Values are passed to and from a sub-routine in one or more program variables (either registers or memory locations). These variables are often determined by programming conventions for the particular processor. For example, the object code programs of Chapter 5 were produced by the same compiler. For the M68000 processor, values to and from a sub-routine are stored on the machine stack, identified by register **A7**. For the PowerPC processor, values are passed to and from a sub-routine in the registers (beginning with **r3**).

A sub-routine can make use of and change any program variable, including those required for other parts of the program. To verify a sub-routine independently of the program, the specification of the sub-routine must describe the variables whose value may change. To do this, it is necessary to compare the program variables before and after the sub-routine is executed. Using the specification operator for programs is too cumbersome since it requires that variables whose value are unchanged by the sub-routine are explicitly listed. An alternative is to define a specification operator which describes the variables whose value can be changed by the sub-routine, with all other variables assumed to be unchanged. Assume a set of name expressions $N : \mathit{Set}(\mathcal{E}_n)$, assertions $P, Q \in \mathcal{A}$ and program $p \in \mathcal{P}$. Let $\{N\}[P]p[Q] \in \mathcal{A}$ be the assertion specifying a sub-routine, defined:

$$\{N\}[P]p[Q] \stackrel{\text{def}}{=} \begin{cases} \lambda s : P(s) \Rightarrow \\ \quad \exists t : s \stackrel{p}{\leadsto} t \land Q(t) \\ \quad\quad \land \forall (n : \mathcal{E}_n) : \neg(\exists n' : n \in N \land n \equiv_{(\mathcal{I}_n, s)} n') \\ \quad\quad\quad \Rightarrow \mathcal{I}_e(n)(s) = \mathcal{I}_e(n)(t) \end{cases}$$

If $\vdash \{N\}[P]p[Q]$ is *true* then program p beginning in a state satisfying P establishes Q and changes only the variables named by a name expression in N. The name expressions in the set N are those which are assigned values by commands in the program p.

For example, program *idiv* of Figure (5.2) satisfies $\vdash [\mathit{Pre}(n,d,a,l)]\mathit{idiv}[\mathit{Post}(n,d,a,l)]$, for any $n,d,a,l \in \mathbb{N}$. The variables which are used by *idiv* are: **r0**, **r1**, **r2**, **r3**, **ref**(**r18**), *pc*. The specification of *idiv* as a sub-routine will therefore be:

$$\vdash \{\mathbf{r0}, \mathbf{r1}, \mathbf{r2}, \mathbf{r3}, \mathbf{ref}(\mathbf{r18}), \mathit{pc}\}[\mathit{Pre}(n,d,a,l)]\mathit{idiv}[\mathit{Post}(n,d,a,l)]$$

The specification of sub-routines can be used during a proof of correctness as a specification of programs which also describes whether the value of a variable is changed. Let $\mathit{changes}(N)(n) = \lambda s : (\exists (n' \in N) : n \equiv_{(\mathcal{I}_n, s)} n')$. The following proof rules allows the specification of sub-routines to used during the verification of a program:

$$\frac{\vdash \{N\}[P]p[Q]}{\vdash [P]p[Q]} \qquad \frac{\vdash \{N\}[P]p[Q] \quad \vdash P \Rightarrow \neg \mathit{changes}(N)(n)}{\vdash [P \wedge n =_a v]p[Q \wedge n =_a v]}$$

where $n \in \mathcal{E}_n$ and $v \in \mathit{Values}$.

The specification of a sub-routine differs from the methods used to specify procedures of structured languages. In general, a procedure operates on variables *local* to the procedure, which may not be used by the program (Gries, 1981). However, languages, such as the C language (Kernighan & Ritchie, 1978), allow a procedure to have *side-effects*: a procedure can change variables used by other parts of the program. The difficulty of verifying such procedures is similar to the difficulty of verifying the sub-routines of object code programs.

6.1.4 Summary

The verification of a program is based on the flow-graph of the program (Floyd, 1967; Manna, 1974; Burstall, 1974). The data operations used by the program determine the difficulty of reasoning about individual program commands. The structure of the program, defined by its flow-graph, determines the difficulty of verifying the program. The proof methods of Floyd (1967) and (Burstall, 1974), which are generally applied to high-level program, apply equally to object code programs. However, the difficulty of verifying a program can depend on the approach taken to carrying out the proof. For example, verifying that a sub-routine satisfies a general specification can avoid the need to repeatedly prove that the sub-routine establishes particular properties.

It is always possible to develop proof methods specialised to the data operations and instructions of a particular processor. Such methods will not be generally applicable and can impose limits on the approach used to verify programs. For example, proof rules can be defined (in terms of the language $\mathcal{L}$) for the instructions of the M68000 processor. These will simplify proofs involving the specifications of M68000 instructions. However, the rules cannot be applied to the commands of $\mathcal{L}$ and therefore cannot be applied to the abstraction of sequences of M68000 instructions. The choice is therefore between simplifying reasoning about instructions or simplifying reasoning about object code programs.

Methods for verifying programs of $\mathcal{L}$ can be developed which support particular approaches to carrying out a proof and which do not constrain other methods of proof. The data operations and the flow-graph of a program can be used to develop generally applicable verification methods. As examples, two approaches to verifying programs of $\mathcal{L}$ will be described. The first uses data refinement to generalise a single proof of correctness across different implementations of a program. This is based on the use of data operations in an object code program and in a specification. The second method allows a proof of correctness to be structured around the flow-graph of the program. Both methods can be applied to any program of $\mathcal{L}$ and both allow the use of other techniques to simplify the proof of correctness.

6.2 Verification by Data Refinement

Object code programs are often produced by compiling a high-level program to a processor language. A number of factors influence the choice of processor used and a single high-level program may be compiled to several different processors. Verifying the object code of each processor will repeat the proof of correctness of the high-level program. Data refinement (Hoare, 1972; Morris, 1989) allows a proof of correctness for a program to be specialised to the object code implementing that program. The approach is to define and verify an abstract program which satisfies the specification of the high-level program. Each of the object code programs is then shown to correctly implement the data operations of the abstract program (Hoare, 1972). This establishes that each of the object code programs satisfies the specification of the high-level program, specialised to the data operations of the object code.

Data refinement is based on manipulating specifications and requires the ability to reason about the assertion language used to specify programs. The assertion language $\mathcal{A}$ is sufficient for the verification of programs of $\mathcal{L}$ (and therefore any object code program) and to define the operators needed for data refinement. However, it is not suitable for proofs by data refinement, which requires some method (such as an induction scheme on the structure of assertions) for reasoning about the properties of assertions. Such an assertion language is outside the scope of this thesis, which is concerned with program verification and not with the assertion language used to specify programs. The description of data refinement given here will be concerned with the data refinement of programs of $\mathcal{L}$. It will not be concerned with those aspects of data refinement which require reasoning about the properties of assertions. A formal model of data refinement, with a suitable assertion language, can be derived from those described by Hoare (1972) or Morris (1989) (together with the substitution operators of $\mathcal{L}$ which are needed to reason about the name expressions of $\mathcal{L}$).

6.2.1 Data Refinement

Assume p is a program of $\mathcal{L}$ which satisfies a specification made up of precondition P and postcondition Q, $\vdash [P]p[Q]$. If p is an abstract program than it can make use of variables and operations which are not provided by a processor language. Program $p_l \in \mathcal{P}$ (modelling an object code program) is a data refinement of p if it correctly implements the variables and operations of

p in terms of the processor language. The implementation is correct if program p_l satisfies any specification S_l which describes the specification S of program p in terms of the data operations of p_l. Variables of the abstract program p will normally be implemented by expressions in p_l. The specification S_l of p_l can therefore be derived from the specification S of p by replacing variables in S with their implementation by p_l. The variables of p and their implementation in p_l can be described as an assignment list al which is substituted into specification S to obtain specification S_l. If p satisfies specification $\vdash [P]p[Q]$ then the specification of p_l is $\vdash [P \lhd al]p_l[Q \lhd al]$. It is also useful to make data refinement conditional on a relationship between variables, described by the *abstraction invariant* of the implementation. This describes the states in which the data refinement must hold between the programs.

Definition 6.1 *Data refinement of programs*

A program $p_1 \in \mathcal{P}$ is a data refinement of $p_2 \in \mathcal{P}$ under assignment list al and abstraction invariant $I \in \mathcal{A}$ iff $p_1 \leq_{I,al} p_2$.

$$\_ \leq_{\_,\_} \_ : (\mathcal{P} \times (\mathcal{A} \times \mathit{Alist}) \times \mathcal{P}) \to \mathit{boolean}$$
$$p_1 \leq_{I,al} p_2 \overset{\text{def}}{=} \begin{cases} \forall (P, Q : \mathcal{A}) :(\vdash [I \wedge P]p_1[I \wedge Q]) \\ \qquad\qquad\qquad \Rightarrow (\vdash [I \wedge (P \lhd al)]p_2[I \wedge (Q \lhd al)]) \end{cases}$$

□

Assume abstract program p_1 satisfies precondition P and postcondition Q and abstraction invariant I, $\vdash [I \wedge P]p_1[I \wedge Q]$. If p_2 data refines p_1 under assignment list al, $p_1 \leq_{I,al} p_2$, then p_2 satisfies the specification $\vdash [I \wedge (P \lhd al)]p_2[I \wedge (Q \lhd al)]$.

Theorem 6.1 *Proof rule for data refinement*

For any $p_1, p_2 \in \mathcal{P}$, $al \in \mathit{Alist}$ and $P, Q, I \in \mathcal{A}$,

$$\frac{p_1 \leq_{I,al} p_2 \quad \vdash [I \wedge P]p_1[I \wedge Q]}{\vdash [I \wedge (P \lhd al)]p_2[I \wedge (Q \lhd al)]}$$

Proof. Immediate, by definition of data refinement. □

Data refinement is a form of program refinement and has some of the same properties. In particular, data refinement is preserved by refinement.

Theorem 6.2 *Properties of data refinement*

For programs $p_1, p_2, p_3 \in \mathcal{P}$, $I \in \mathcal{A}$ and $al \in \mathit{Alist}$,

1. *If program p_2 is data refined by program p_3 then any abstraction of p_2 is also data refined by p_3.*

$$\frac{p_1 \sqsubseteq p_2 \quad p_2 \leq_{I,al} p_3}{p_1 \leq_{I,al} p_3}$$

2. *If program p_1 is data refined by p_2 then it is also data refined by any refinement of p_2.*

$$\frac{p_1 \leq_{I,al} p_2 \quad p_2 \sqsubseteq p_3}{p_1 \leq_{I,al} p_3}$$

Proof. Straightforward by definition of data refinement (Definition 6.1) and the refinement rule of programs (rule tl7). □

To show that a program p_2 is a data refinement of a program p_1, it is enough to show that every command in p_1 is data refined by p_2. The abstraction invariant I determines the states in which the assignment list al implements the data refinement and must be preserved by every command of p_1.

Theorem 6.3 *Establishing data refinement*

For $c \in \mathcal{C}$, $p_1, p_2 \in \mathcal{P}$, $c \in \mathcal{C}$, $al \in$ Alist and $A, B, A_1, B_1, P, Q, I \in \mathcal{A}$,

$$\frac{\begin{array}{l} \forall c \in p_1 : \forall A, B : (\vdash A \Rightarrow \mathbf{wp}(c, B)) \Rightarrow (\vdash I \wedge A \Rightarrow \mathbf{wp}(c, I \wedge B)) \\ \wedge \vdash (I \wedge (P \lhd al)) \Rightarrow P \\ \wedge \forall c \in p_1 : \forall A, B : \vdash ((I \wedge A) \Rightarrow \mathbf{wp}(c, I \wedge B) \Rightarrow [I \wedge (A \lhd al)]p_2[I \wedge (B \lhd al)]) \end{array}}{p_1 \leq_{I,al} p_2}$$

Theorem (6.3) allows the data refinement p_2 of a program p_1 to implement a command of p_1 by any number of commands in p_2. For example, an integer division operation in an abstract program may be implemented as a sub-routine in the object code implementing the program. The properties of Theorem (6.2) can be used to simplify a proof of data refinement by showing that an abstraction of the implementation data refines the abstract program.

6.2.2 Application of Data Refinement

The form of data refinement used in Definition (6.1) is intended to simplify the verification of different implementations of a single program p. Program p will typically be a high-level program and its implementations produced by compiling p for different processors. Because semantics for high-level languages are difficult to obtain, p will be taken to be a program of $\mathcal{L}$ (which may make use of operations not provided by a processor). Assume programs $p_1, \ldots, p_n \in \mathcal{P}$ model the object code programs implementing p on different processors. To show that each of the implementations is correct would require a proof that each p_i satisfies the specification of p described in terms of the operations provided by the processor language of p_i.

Data refinement provides a simpler approach. First program p is shown to satisfy its specification, $\vdash [P \wedge I]p[Q \wedge I]$, where $I \in \mathcal{A}$ is the abstraction invariant. Each implementation p_i is then verified by showing that p_i data refines p, $p \leq_{I,al} p_i$ (for $al \in$ *Alist*). This step will use Theorem (6.3), to show that each command of p is data refined by one or more commands of p_i. The correctness of p_i then follows from Theorem (6.1), which establishes $\vdash [I \wedge P \triangleleft al]p_i[I \wedge Q \triangleleft al]$. This approach is similar to that of Hoare (1972), where data refinement is used to simplify the program to be verified.

Program abstraction reduces the number of commands to be considered and can therefore simplify both the proof of correctness and the proof of data refinement. The approach is to construct an abstraction p' of the abstract program p, $p' \sqsubseteq p$, containing only commands needed to establish the specification, $\vdash [P \wedge I]p'[Q \wedge I]$. An abstraction p'_i will also be constructed for each object code program p_i, $p'_i \sqsubseteq p_i$. The abstractions will then be shown to be a data refinement of p', $p' \leq_{al} p'_i$. By Theorem (6.2), this is enough to establish $p' \leq_{al} p_i$ and, by Theorem (6.1), $\vdash [I \wedge P \triangleleft al]p_i[I \wedge Q \triangleleft al]$.

6.2.3 Example: Division

Consider the C program for natural division of Figure (5.1) which was compiled to object code programs for both the M68000 and the PowerPC processors. As an example of the use of data refinement, the verification of the PowerPC object code will be repeated by showing its model in $\mathcal{L}$ (program *ppcdiv* of Figure 5.15) is a data refinement of an abstract program for natural division. The abstract program will be derived from program *idiv* of Figure (5.2). This is an implementation of the C program in $\mathcal{L}$ and was shown to be correct by verifying its abstraction $idiv_2$. Only two commands of $idiv_2$ were needed: commands C_1 and C_2 of Figure (5.3). Let $idiv_3$ be the program $\{C_1, C_2\}$; this will be the abstract program for the data refinement. Program $idiv_3$ can be shown to satisfy the specification of $idiv_2$, the proof is the same as for $idiv_2$ (see Chapter 5).

The program $idiv_3$ will be used to verify *ppcdiv* by showing that *ppcdiv* is a data refinement of $idiv_3$. The approach is to show that the abstraction $ppcdiv_2$ of *ppcdiv* (given in Figure 5.16) is a data refinement of $idiv_3$. By Theorem (6.2) and since $ppcdiv_2 \sqsubseteq ppcdiv$, this will show that *ppcdiv* is a data refinement of $idiv_3$. The proof of data refinement is by Theorem (6.3). Programs $idiv_3$ and $ppcdiv_2$ are used to reduce the number of commands which must be considered during the proof.

The specification satisfied by $idiv_3$ is that of $idiv_1$ while the specification to be established by $ppcdiv_2$ is that of *ppcdiv*. Both specifications are given in Chapter 5 and are re-stated here in a form more suitable for the proof of data refinement. The specification satisfied by program $idiv_3$ has precondition $Pre_a(n, d, a, l)$ and postcondition $Post_a(n, d, a, l)$ for any $n, d, a, l \in Values$:

$$Pre_a(n, d, a, l) \stackrel{\text{def}}{=} pc =_{32} l_1 \wedge d > 0 \wedge \mathbf{r16} =_{32} n \wedge \mathbf{r17} =_{32} d \wedge \mathbf{r18} =_{32} a \wedge \mathbf{r26} =_{32} l$$
$$Post_a(n, d, a, l) \stackrel{\text{def}}{=} pc =_{32} l \wedge d > 0 \wedge \mathbf{r18} =_{32} a \wedge n =_{32} (\mathbf{r0} \times_{32} d) +_{32} \mathbf{ref}(a)$$

The specification to be satisfied by $ppcdiv_2$ has precondition $Pre_l(n, d, a, l)$ and postcondition $Post_l(n, d, a, l)$:

$$Pre_l(n, d, a, l) \stackrel{\text{def}}{=} pc =_{32} l_1 \wedge d > 0 \wedge \mathbf{r3} =_{32} n \wedge \mathbf{r4} =_{32} d \wedge \mathbf{r5} =_{32} a \wedge \mathbf{LR} =_{32} l$$
$$Post_l(n, d, a, l) \stackrel{\text{def}}{=} pc =_{32} l \wedge d > 0 \wedge \mathbf{r5} =_{32} a \wedge n =_{32} (\mathbf{r0} \times_{32} d) +_{32} \mathbf{readl}(a)$$

Data refinement is intended to show that the correctness of $idiv_3$ establishes the correctness of $ppcdiv_2$:

$$\frac{\vdash [Pre_a(n,d,a,l)]idiv_3[Post_a(n,d,a,l)]}{\vdash [Pre_l(n,d,a,l)]ppcdiv_2[Post_l(n,d,a,l)]}$$

The specifications differ from those in Chapter 5 only in explicitly specifying the value of the program counter pc.

Implementation of Variables

To show that $ppcdiv_2$ is a data refinement of $idiv_3$ requires an assignment list al and an invariant $I \in \mathcal{A}$. The assignment list describes the implementation of variables in the specification of $idiv_3$ as expressions in $ppcdiv_2$. The variables of $idiv_3$ are $pc, \mathbf{r16}, \mathbf{r17}, \mathbf{r18}, \mathbf{ref}(a)$ and $\mathbf{r26}$. With the exception of pc, these are implemented in $ppcdiv_2$ as the PowerPC registers $\mathbf{r3}, \mathbf{r4}, \mathbf{r5}, \mathbf{readl}(a), \mathbf{LR}$ respectively. This give the assignment list: $al = (\mathbf{r16}, \mathbf{r3}) \cdot (\mathbf{r17}, \mathbf{r4}) \cdot (\mathbf{r18}, \mathbf{r5}) \cdot (\mathbf{ref}(a), \mathbf{readl}(a)) \cdot (\mathbf{r26}, \mathbf{LR})$.

The program counter in both $idiv_3$ and in $ppcdiv_2$ is pc. However, the values required of the program counter are different. In $idiv_3$, the value of the program counter is one of the labels l_1, l_6 or l. In $ppcdiv_2$, the program counter is one of $l_1, \dots, l_{10}, l$: program $ppcdiv_2$ contains all commands of $ppcdiv_2$ except those at labels l_1 and l_8. Since only the commands of $ppcdiv_2$ labelled l_1 and l_8 are needed, the program counter can be implemented by mapping the label l_8 in $ppcdiv_2$ to the label l_6 in $idiv_3$. This uses the conditional expression $\mathbf{cond} \in \mathcal{E}$ described in Chapter 5 and defined in Appendix A. In the assignment list al, the replacement for pc is the expression $\mathbf{cond}(pc =_{32} l_8, l_6, pc)$ (which is equivalent to l_6 when $pc =_{32} l_8$). The full assignment list al is therefore:

$$\begin{aligned} al = &(pc, \mathbf{cond}(pc =_{32} l_8, l_6, pc)) \cdot (\mathbf{r16}, \mathbf{r3}) \cdot (\mathbf{r17}, \mathbf{r4}) \\ &\cdot (\mathbf{r18}, \mathbf{r5}) \cdot (\mathbf{ref}(a), \mathbf{readl}(a)) \cdot (\mathbf{r26}, \mathbf{LR}) \end{aligned}$$

Assignment list al is used to show that $\vdash [I \wedge Pre_a(n,d,a,l)]idiv_3[I \wedge Post_a(n,d,a,l)]$ establishes $\vdash [I \wedge (Pre_a(n,d,a,l) \lhd al)]ppcdiv_2[I \wedge (Post_a(n,d,a,l) \lhd al)]$. To see that the assignment list correctly implements the variables of $idiv_3$ and therefore establishes the correctness of $ppcdiv_2$, note that $(Pre_a(n,d,a,l) \lhd al) = Pre_l(n,d,a,l)$ and $(Post_a(n,d,a,l) \lhd al) = Post_l(n,d,a,l)$. For example, the precondition of $ppcdiv_2$, $Pre_l(n,d,a,l)$, is obtained by:

$$\begin{aligned} & Pre_a(n,d,a,l) \lhd al \\ = \;& (pc =_{32} l_1 \wedge d > 0 \wedge \mathbf{r16} =_{32} n \wedge \mathbf{r17} =_{32} d \wedge \mathbf{r18} =_{32} a \wedge \mathbf{r26} =_{32} l) \lhd al \\ = \;& pc =_{32} l_1 \wedge d > 0 \wedge \mathbf{r3} =_{32} n \wedge \mathbf{r4} =_{32} d \wedge \mathbf{r5} =_{32} a \wedge \mathbf{LR} =_{32} l \\ = \;& Pre_l(n,d,a,l) \end{aligned}$$

Abstraction Invariant

The abstraction invariant for the data refinement between $idiv_3$ and $ppcdiv_2$ must satisfy the two conditions of Theorem (6.3). Let $absInv(n, d, a, l) \in \mathcal{A}$ (where $n, d, a, l \in Values$) be the abstraction invariant defined:

$$absInv(n, d, a, l) \stackrel{\text{def}}{=} \begin{cases} (pc =_{32} l_1 \lor pc =_{32} l_6 \lor pc =_{32} l) \\ \land pc =_{32} l_1 \Rightarrow Pre_a(n, d, a, l) \\ \land pc =_{32} l_6 \Rightarrow Inv(n, d, a, l) \\ \land pc =_{32} l \Rightarrow Post_a(n, d, a, l) \end{cases}$$

where $Inv(n, d, a, l)$ is the loop invariant for $idiv_2$ defined in Chapter 5.

The abstraction invariant $absInv(n, d, a, l)$ satisfies the conditions of Theorem (6.3). The first, $\vdash (absInv(n, d, a, l) \land (Pre_a(n, d, a, l) \lhd al)) \Rightarrow Pre_a(n, d, a, l)$, is immediate by definition of Pre_a which requires $pc =_{32} l_1$. The second, that for any command $c \in idiv_3$ and $A, B \in \mathcal{A}$: $\vdash (A \Rightarrow \mathbf{wp}(c, B)) \Rightarrow absInv(n, d, a, l) \land A \Rightarrow \mathbf{wp}(c, absInv(n, d, a, l) \land B)$ is straightforward. Since c is either C_1 or C_2, the proof is similar to that used to verify program $idiv_2$.

Proof of Data Refinement

To show that $ppcdiv_2$ data refines $idiv_3$, each specification satisfied by a command of $idiv_3$ must be satisfied by $ppcdiv_2$ (Theorem 6.3). The proof is by Theorem (6.3) and, since there are two commands in $idiv_3$, is in two parts. Both require a proof that for any $A, B \in \mathcal{A}$ and $c \in \{C_1, C_2\}$:

$$\begin{aligned} &\vdash (absInv(n, d, a, l) \land A) \Rightarrow \mathbf{wp}(c, I \land B) \\ &\quad \Rightarrow [absInv(n, d, a, l) \land A \lhd al] ppcdiv_2 [absInv(n, d, a, l) \land B \lhd al] \end{aligned}$$

The proofs for both C_1 and C_2 require reasoning by induction about the (arbitrary) assertion A. The assertion language $\mathcal{A}$ does not support such reasoning and the proof here will only describe the steps needed to establish data refinement. Only two commands of $ppcdiv_2$ are needed: the command labelled l_1 and l_8 given in Figure (5.16). The command labelled l_1 will be referred to as C_{11} and the command labelled l_8 will be referred to as C_{18}.

For $c = C_1$: since C_1 is an assignment command labelled with l_1 (Figure 5.3), the assertion $absInv(n, d, a, l) \land A \Rightarrow \mathbf{wp}(C_1, absInv(n, d, a, l) \land B)$ can be replaced by $pc =_a l_1 \land (absInv(n, d, a, l) \land A) \Rightarrow (absInv(n, d, a, l) \land B) \lhd bl_1$ where bl_1 is the assignment list of C_1 (this uses the assignment rule tl1). This must establish the specification $[absInv(n, d, a, l) \land A \lhd al] ppcdiv_2 [absInv(n, d, a, l) \land B \lhd al]$. Since $pc =_a l_1$, it follows that command C_{11} of $ppcdiv_2$ is enabled. Since this is an assignment command labelled with l_1, the proof requires $pc =_a l_1 \land absInv(n, d, a, l) \land (A \lhd al) \Rightarrow (absInv(n, d, a, l) \land B \lhd al) \lhd bl_2$ where bl_2 is the assignment list of C_{11}. The assertion to prove is therefore:

$$\begin{aligned} &\vdash pc =_a l_1 \land (absInv(n, d, a, l) \land A) \Rightarrow (absInv(n, d, a, l) \land B) \lhd bl_1 \\ &\quad \Rightarrow (pc =_a l_1 \land absInv(n, d, a, l) \land (A \lhd al) \Rightarrow (absInv(n, d, a, l) \land B \lhd al) \lhd bl_2) \end{aligned}$$

The truth of this assertion is straightforward, by definition of $\mathit{absInv}(n, d, a, l)$ and by substitution of al, bl_1 and bl_2. The truth of assertion $A \lhd al$ can be established from the truth of A and $\mathit{absInv}(n, d, a, l)$. The invariant requires the values of the variables of idiv_2 and their implementation in ppcdiv_2 to be equal at label l_6. Any other variable used by A is not replaced in $A \lhd al$. Note that this is a property which must be established by induction on A and therefore cannot be established in the assertion language $\mathcal{A}$.

To show that postcondition $(B \lhd al) \lhd bl_2$ is established by $B \lhd bl_1$, note that every variable of idiv_3 assigned a value (by bl_1) is replaced (by al) with its implementation in ppcdiv_2. The assignment list bl_2 then updates the variables of ppcdiv_2 with the values assigned by idiv_3. As before, any variable used by B which is not assigned a value by idiv_3 is not assigned to by ppcdiv_2. The assertion $(B \lhd al) \lhd bl_2$ is therefore satisfied by $B \lhd bl_1$. This completes the first step, showing that command C_1 of idiv_3 is data refined by ppcdiv_2.

The second step, showing that command C_2 is data refined by ppcdiv_2 is similar to the first. The proof is by showing that any specification satisfied by C_2 is satisfied by C_{18}, after replacing the variables of idiv_3 with their implementation in ppcdiv_2. This establishes that C_{18} of program ppcdiv_2 data refines command C_2 of idiv_3. Since there are only two commands in idiv_3, the two steps establish that ppcdiv_2 is a data refinement of idiv_3 under assignment list al and invariant $\mathit{absInv}(n, d, a, l)$: $\mathit{idiv}_3 \leq_{\mathit{absInv}(n,d,al),al} \mathit{ppcdiv}_3$. The correctness of ppcdiv_2 then follows from the correctness of idiv_3.

Note that program *idiv* models an object code program for the Alpha AXP processor implementing the C program for natural division and the PowerPC program *ppcdiv* was also obtained by compiling this C program. The proof that idiv_3 is data refined by ppcdiv_2 is therefore an example of how a proof of correctness for a single program can be transferred between the implementations of the program on different processors.

6.2.4 Related Work

The approach to data refinement used here is simpler than others, such as those of Hoare (1972), Morris (1989) and Abadi & Lamport (1991). These provide a more powerful form of data refinement which allows reasoning about programs which operate on distinct variables and values. The approach used here is sufficient for the relatively simple requirements of verifying different implementations of a program. Note that no part of the approach used here nor of the standard methods (Hoare, 1972; Morris, 1989; Abadi & Lamport, 1991) is specific to a particular language. At the cost of a more complex development, the more powerful methods can be equally well applied to the abstract language $\mathcal{L}$

6.3 Structured Proofs

Verification in a structured language uses the syntax of a program to derive specifications of the program commands from the specification of the program. This is the reverse of verification in a flow-graph language (such as $\mathcal{L}$), where the specification of a program is built up from specifications of the commands (rule tl6 of the program logic). As an example of a proof method which can be applied to any program of $\mathcal{L}$, a technique will be described which allows the verification of a program to be based on the flow-graph of the program. The method uses regions of a program to define a structure which can be used to break down a program specification to specifications of individual program commands.

The method is based on a predicate transformer for regions, **wp**, which interprets a regions as a discrete unit of a program. If r is a region, then $\mathbf{wp}(r, Q)$ is the weakest precondition required for r to terminate in a state satisfying postcondition Q. The function **wp** is based on the flow-graph of a region and on the weakest precondition of the commands making up the region. This is similar to the definition of a *wp* predicate transformer for the compound commands of a structured language (Dijkstra, 1976). The main difference is the treatment of loops in a program. A structured language permits only nested loops and each loop can be verified as a discrete component of the program. A region can contain both nested and interleaving loops. Loops in a region must therefore be considered in terms of the paths between the cut-points of the region.

6.3.1 Weakest Precondition of Regions

The **wp** function for regions is defined in two steps. In the first, a function $\mathbf{wp}_0$ is defined which describes the weakest precondition of a loop-free sequence of commands, beginning with the head of the region. This is used to establish the properties of a path between cut-points of a region. Function $\mathbf{wp}_0$ is used in the second step to derive a function, $\mathbf{wp}_1$, describing the weakest precondition needed to establish a postcondition by executing paths in a region any number of times. The function $\mathbf{wp}_1$ does not require the region to terminate and the **wp** function for regions is obtained by strengthening function $\mathbf{wp}_1$ to require that the region terminates.

Weakest Precondition of Paths

Function $\mathbf{wp}_0$ is defined by recursion on region r. If Q is the postcondition to be established by region r then $\mathbf{wp}_0(r, Q)$ is the weakest precondition needed for the head of r to establish assertion q. Assertion q is either the postcondition Q or the weakest precondition needed for a sub-region $r' \in \mathit{rest}(r)$ to establish Q, $q' \Rightarrow \mathbf{wp}_0(r', Q)$. The recursive definition of $\mathbf{wp}_0$ constructs a series of intermediate assertions for each sub-region r'. These describe the specifications to be established by each command in a path beginning with the head of the region. The path must end in a state in which no command, other than the head of the region, is enabled. If r is a unit region then $\mathbf{wp}_0(r, Q)$ is $\mathbf{wp}(\mathit{head}(r), Q)$ (the only path through r is made up of the head of r).

Definition 6.2 *Weakest precondition of paths*

For any set A of assertions, $\bigvee A$ is *true* in a state s iff there is an assertion in A which is *true* in s.

$$\begin{array}{l}\bigvee \_ : \mathit{Set}(\mathcal{A}) \to \mathcal{A} \\ \bigvee A \stackrel{\text{def}}{=} \lambda(s : \mathit{State}) : \exists a : a \in A \wedge a(s)\end{array}$$

For any region r and assertion Q, $\mathbf{wp}_0(r, Q)$ is the weakest precondition required to establish Q by executing any path through r.

$$\begin{array}{l}\mathbf{wp}_0 : (\mathcal{R} \times \mathcal{A}) \to \mathcal{A} \\ \mathbf{wp}_0(r, Q) \stackrel{\text{def}}{=} \begin{cases} \mathbf{wp}(\mathit{head}(r), Q) & \text{if } \mathit{unit?}(r) \\ \mathbf{wp}(\mathit{head}(r), Q') & \text{otherwise}\end{cases}\end{array}$$

where $Q' = \begin{cases} \bigvee(\{q : \mathcal{A} \mid \exists(r_1 : \mathcal{R}) : r_1 \in \mathit{rest}(r) \wedge (\forall s : q(s) \Rightarrow \mathbf{wp}_0(r_1, Q)(s))\} \\ \quad \cup \{(\mathit{final?}(\mathit{body}(r) - \{\mathit{head}(r)\}) \wedge Q)\})\end{cases}$ □

Because $\mathbf{wp}_0$ is defined by recursion, the proof that a region r satisfies its specification $\vdash P \Rightarrow \mathbf{wp}_0(r, Q)$ can be carried out by induction on r (Theorem 4.9).

The weakest precondition of a region r, $\mathbf{wp}(r, Q)$, is derived from the weakest precondition needed for one or more paths through the region to establish a postcondition, $\mathbf{wp}_1(r, Q)$. A path through region r is constructed as a *maximal sub-region*: the largest sub-region of r which can be constructed from some command of r. The *weakest liberal precondition* needed for r to establish postcondition Q is an assertion, $\mathbf{wp}_1(r, Q)$, such that execution, in sequence, of any number of maximal sub-regions of r in a state satisfying $\mathbf{wp}_1(r, Q)$ eventually produces a state satisfying Q. It is not necessary for execution of r to terminate. To establish termination, the function $\mathbf{wp}(r, Q)$, requires that r produces a state satisfying Q which is final for r.

Definition 6.3 *Weakest precondition of a region*

Region r_1 is a maximal sub-region of r_2, written $r_1 < r_2$ iff r_1 is the largest region which can be constructed from the body of r_2, beginning with any command of r_2.

$$\begin{array}{l}\_ < \_ : (\mathcal{R} \times \mathcal{R}) \to \mathit{boolean} \\ r_1 < r_2 \stackrel{\text{def}}{=} \exists c : c \in \mathit{body}(r_2) \wedge r_1 = \mathit{region}(\mathit{label}(c), \mathit{body}(r_2))\end{array}$$

For region $r \in \mathcal{R}$ and assertion $Q \in \mathcal{A}$, $\mathbf{wp}_1(r, Q) \in \mathcal{A}$ is the weakest liberal precondition required for r to establish assertion Q. Function $\mathbf{wp}_1$ has type $(\mathcal{R} \times \mathcal{A}) \to \mathcal{A}$ and inductive definition:

$$\frac{\mathbf{wp}_0(r, Q)(s)}{\mathbf{wp}_1(r, Q)(s)} \qquad \frac{\mathbf{wp}_1(r, q)(s) \quad r_1 < r \quad \vdash q \Rightarrow \mathbf{wp}_0(r_1, Q)}{\mathbf{wp}_1(r, Q)(s)}$$

where $r_1 \in \mathcal{R}$, $q \in \mathcal{A}$ and $s \in \mathit{State}$.

For region r and assertion Q, $\mathbf{wp}(r, Q)$ is the weakest precondition required for r to terminate and establish assertion Q.

$$\mathbf{wp} : (\mathcal{R} \times \mathcal{A}) \to \mathcal{A}$$
$$\mathbf{wp}(r, Q) \stackrel{\text{def}}{=} \mathbf{wp}_1(r, Q \wedge \mathit{final?}(r))$$

□

The weakest liberal precondition, $\mathbf{wp}_1$, is the transitive closure of the weakest precondition for paths, $\mathbf{wp}_0$. Properties of the weakest precondition of regions, $\mathbf{wp}$, will generally be established by reasoning about the weakest liberal precondition function $\mathbf{wp}_1$ since function $\mathbf{wp}_0$ simply strengthens the postcondition established by $\mathbf{wp}_1$.

6.3.2 Properties of the Weakest Precondition

The weakest precondition for paths has many of the properties of a maximal trace. If there is only one command in a region r, then the weakest precondition of a path is the weakest precondition of the head of r. If there is more than one command then the weakest precondition for a path through r can be established from the weakest preconditions for the head of r (which begins the path) and for any $r' \in \mathit{rest}(r)$ (which describes the remainder of the path).

The weakest liberal precondition, $\mathbf{wp}_1$, specifies the transitive closure of maximal traces through r, beginning with the head of r. Formally, function $\mathbf{wp}_1$ satisfies the equation:

$$\mathbf{wp}_1(r, Q)(s) \Leftrightarrow (\mathit{enabled}(r)(s) \wedge (\exists t : \mathit{mtrace}^+(\mathit{body}(r), s, t)) \wedge Q(t)))$$

where $r \in \mathcal{R}$, $Q \in \mathcal{A}$ and $s, t \in \mathit{State}$ (see Lemma E.4 of the appendix). When the postcondition to be established by the region requires a final state, the weakest liberal precondition specifies the total correctness of a region. The weakest precondition of a region, $\mathbf{wp}(r, Q)$, strengthens postcondition Q to require termination of r.

Theorem 6.4 *Properties of* **wp**

Assume $P, Q \in \mathcal{A}$, $s, t \in \mathit{State}$, $r \in \mathcal{R}$. The assertion $\mathbf{wp}(r, Q)$ is the weakest precondition needed to establish Q by a terminating execution of region r.

$$\mathbf{wp}(r, Q)(s) \Leftrightarrow (\exists t : \mathcal{I}_r(r)(s, t) \wedge Q(t))$$

Reasoning about the weakest precondition, $\mathbf{wp}$, of a region is simpler when considering the weakest liberal precondition, $\mathbf{wp}_1$. Proof rules for $\mathbf{wp}_1$ use the transitive closure of maximal traces to describe the behaviour of sequences of commands in the region. Proof rules for $\mathbf{wp}_0$ can be used to reason about the individual commands making up the sequences specified by $\mathbf{wp}_1$. These two functions are enough when reasoning about the specification of loop-free regions. However, to prove that a region containing a loop establishes a postcondition, it is necessary to use some form of induction (Floyd, 1967). Induction schemes for the regions can be derived from induction on a well-founded set (Loeckx & Sieber, 1987) with elements of the set representing decreasing values of variables used in the region.

$$\frac{\textit{unit?}(r) \quad \vdash P \Rightarrow \mathbf{wp}(\mathit{head}(r), Q)}{\vdash P \Rightarrow \mathbf{wp}_0(r, Q)} \qquad \text{(tl20)}$$

$$\frac{r_1 \in \textit{rest}(r) \quad \vdash P \Rightarrow \mathbf{wp}(\mathit{head}(r), q) \quad \vdash q \Rightarrow \mathbf{wp}_0(r_1, Q)}{\vdash P \Rightarrow \mathbf{wp}_0(r, Q)} \qquad \text{(tl21)}$$

$$\frac{r_1 < r \quad \vdash P \Rightarrow \mathbf{wp}_1(r, q) \quad \vdash q \Rightarrow \mathbf{wp}_1(r_1, Q)}{\vdash P \Rightarrow \mathbf{wp}_1(r, Q)} \qquad \text{(tl22)}$$

$$\frac{j < i \quad r_1 < r \qquad \dfrac{\vdash \textit{enabled}(r_1) \wedge F(j) \Rightarrow \mathbf{wp}_1(r_1, Q)}{\vdash F(i) \Rightarrow \mathbf{wp}_1(r, Q)}}{\vdash F(n) \Rightarrow \mathbf{wp}_1(r, Q)} \qquad \text{(tl23)}$$

$$\frac{r \equiv r_1 \quad \vdash P \Rightarrow \mathbf{wp}(r_1, Q)}{\vdash P \Rightarrow \mathbf{wp}(r, Q)} \qquad \text{(tl24)}$$

$$\frac{\textit{body}(r) \subseteq p \quad \vdash P \Rightarrow \mathbf{wp}_1(r, Q)}{\vdash [P]p[Q]} \qquad \text{(tl25)}$$

where $r, r_1 \in \mathcal{R}$, $i, j, n \in \mathbb{N}$ and $P, Q, q \in \mathcal{A}$

Figure 6.1: Proof Rules for the Regions

Theorem 6.5 *General induction*

Let T be some type and $\prec$ a well founded relation of type $(T \times T) \rightarrow$ boolean. There is a well founded induction scheme on assertions indexed by the type T. For $G : T \rightarrow \mathcal{A}$, $Q \in \mathcal{A}$, $r, r_1 \in \mathcal{R}$, $i, j, n \in T$ and $s \in$ State:

$$\frac{j \prec i \quad r_1 < r \qquad \dfrac{\vdash \textit{enabled}(r_1) \wedge G(j) \Rightarrow \mathbf{wp}_1(r_1, Q)}{\vdash G(i) \Rightarrow \mathbf{wp}_1(r, Q)}}{\vdash G(n) \Rightarrow \mathbf{wp}_1(r, Q)}$$

The induction scheme of Theorem (6.5) is equivalent to the intermittent assertions methods (Burstall, 1974; Cousot & Cousot, 1987). A simple induction scheme based on the natural numbers, similar to that used for the programs (rule tl9), is straightforward to derive from the induction scheme of Theorem (6.5).

strcopy:	li r3, 0	$l_1:$	$\mathbf{r3} := \mathbf{Long}(0), l_2$
loop:	lbzx r4, r3, r1	$l_2:$	$\mathbf{r4} := \mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3})), l_3$
	stb r4, r3, r2	$l_3:$	$\mathbf{ref}(\mathbf{r2} +_{32} \mathbf{r3}) := \mathbf{r4}, l_4$
	addic 3, 3, 1	$l_4:$	$\mathbf{r3} := \mathbf{r3} +_{32} 1, l5$
	cmpwi 1, r3, 0	$l_5:$	$\mathbf{CRF} := \mathbf{calcCRF}(\mathbf{r4}), l_6$
	bc 4, 5, loop	$l_6:$	**if** $\mathbf{bit}(2)(\mathbf{CRF})$ **then goto** l_2 **else goto** l_7
	blr	$l_7:$	**goto** $\mathbf{loc}(\mathbf{LR})$

PowerPC program — $\mathcal{L}$ program *strcopy*

Figure 6.2: String Copy

Proof Rules

Both function $\mathbf{wp}_1$ and function **wp** can be used to reason about programs. Both establish that a region eventually produces a state satisfying a postcondition. If the region is a subset of a program, then the program must also produce that state. The weakest precondition, **wp**, also allows a region to be replaced with an equivalent region. During the course of a proof, this can be used to manipulate the region being verified.

Proof rules for the regions are given in Figure (6.1). Rule (tl20) and rule (tl21) derive the specification of a path in a region r from the specifications of paths in sub-regions of r. Rule (tl22) breaks down the specification of a region into a specification to be established by a maximal sub-region. This rule is similar to the composition rule in structured languages (Dijkstra, 1976). Rule (tl23) is an induction scheme for regions based on the natural numbers. Rule (tl24) allows region r to be replaced with an equivalent region and rule (tl25) relates the specification of a region in a program to the program.

6.3.3 Example: String Copy

For an example of the use of regions to verify a program, consider the PowerPC program of Figure (6.2). This copies the contents of an array a into an array b. Both arrays are identified by their address in memory: the address of the first element of a is stored in register **r1**, the address of the first element of b is stored in **r2**. The end of array a is identified by an element containing the value 0. The program uses the conventions of the C language, where a string is identified by its first element and terminated by a null character (Kernighan & Ritchie, 1978).

The PowerPC program is modelled by the $\mathcal{L}$ program *strcopy* of Figure (6.2). Program *strcopy*$_1$ of Figure (6.3) is an abstraction of *strcopy*, *strcopy*$_1 \sqsubseteq$ *strcopy*. Program *strcopy*$_1$ is constructed using the methods described in Chapter 4: a region is formed beginning with the

$$
\begin{array}{l}
C_1 = l_1 : \mathbf{r3} := \mathbf{Long}(0), l_2 \\
C_2 = l_2 : \textbf{if } \mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3}) =_{32} 0 \\
\qquad \textbf{then } \mathbf{r4}, \mathbf{ref}(\mathbf{r2} +_{32} \mathbf{r3}), \mathbf{r3}, \mathbf{CRF} := \\
\qquad\qquad \mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3}), \mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3}), \mathbf{r3} +_{32} 1, \\
\qquad\qquad \mathbf{calcCRF}(\mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3})), \mathbf{loc}(\mathbf{LR}) \\
\qquad \textbf{else } \mathbf{r4}, \mathbf{ref}(\mathbf{r2} +_{32} \mathbf{r3}), \mathbf{r3}, \mathbf{CRF} := \\
\qquad\qquad \mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3}), \mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3}), \mathbf{r3} +_{32} 1, \\
\qquad\qquad \mathbf{calcCRF}(\mathbf{ref}(\mathbf{r1} +_{32} \mathbf{r3})), l_2
\end{array}
$$

Figure 6.3: Abstraction *strcopy*$_1$ of *strcopy*

command labelled l_1 and excluding the command at l_7, this is abstracted by applying the general transformation T_2. The resulting commands are composed with the command of *strcopy* at label l_7. The commands of *strcopy*$_1$ will be referred to as C_1 and C_2

Specification of *strcopy*$_1$

The specification to be satisfied by *strcopy*$_1$ requires that, given the address of two arrays, a, b, the program copies the contents of a into b. The precondition $\mathit{Pre}(a, b, l, n) \in \mathcal{A}$ requires that the link register **LR** contains the label l to which control is to return. No command in the program can be labelled with l. The precondition also requires that nth element of the array a is the first null character. The postcondition $\mathit{Post}(a, b, l, n) \in \mathcal{A}$ requires that array b is equal to array a.

$$
\mathit{Pre}(a, b, l, n) \stackrel{\text{def}}{=} \begin{cases} \mathbf{LR} =_{32} l \wedge \neg(l =_{32} l_1 \vee l =_{32} l_2) \wedge \mathbf{ref}(a +_{32} n) =_{32} 0 \\ \wedge \forall(\lambda i : i <_a n \Rightarrow \neg \mathbf{ref}(a +_{32} i) =_{32} 0) \end{cases}
$$

$$
\mathit{Post}(a, b, l, n) \stackrel{\text{def}}{=} \begin{cases} \mathbf{LR} =_{32} l \wedge \mathbf{ref}(a +_{32} n) =_{32} 0 \\ \wedge \forall(\lambda i : i <_a n \Rightarrow \neg \mathbf{ref}(a +_{32} i) =_{32} 0) \\ \wedge \forall(\lambda i : n \geq_a i \Rightarrow \mathbf{ref}(a +_{32} i) =_{32} \mathbf{ref}(b +_{32} i)) \end{cases}
$$

The specification to be satisfied by *strcopy*$_1$ is, for any $a, b, n, l \in \mathbb{N}$:

$$
\vdash [\mathit{pc} =_{32} l_1 \wedge \mathit{Pre}(a, b, l, n)]\mathit{strcopy}_1[\mathit{pc} =_{32} l \wedge \mathit{Post}(a, b, l, n)]
$$

Verification of *strcopy*$_1$

Program *strcopy*$_1$ will be verified by constructing a region r of the program and showing that this region satisfies the specification. Region r begins at label l_1 and contains the two commands C_1 and C_2: $r = \mathit{region}(l_1, \{C_1, C_2\})$. The specification to be satisfied by r is defined as a single assertion $spec(i, a, b, l, n)$, which indexes the properties required of the region by the value of the program counter. The properties described by $spec(i, a, b, l, n)$ will also be used during the proof of correctness. For example, the invariant of the loop at command C_2 is described by $spec(i, a, b, l, n)$, where $i \in \mathbb{N}$ is value on which the induction rule (tl23) is based. For i, a, b, l, n, assertion $spec(i, a, b, l, n)$ is defined:

$$spec(i, a, b, l, n) \stackrel{\text{def}}{=} \begin{cases} Pre(a, b, l, n) \\ \wedge pc =_{32} l_2 \Rightarrow i =_{32} (n -_{32} \mathbf{r3}) \\ \qquad\qquad \forall(\lambda i : i <_a \mathbf{r3} \Rightarrow \mathbf{ref}(a +_{32} i) =_{32} \mathbf{ref}(b +_{32} i)) \\ \wedge pc =_{32} l \Rightarrow i =_{32} (n -_{32} \mathbf{r3}) \\ \qquad\qquad \forall(\lambda i : i <_a \mathbf{r3} \Rightarrow \mathbf{ref}(a +_{32} i) =_{32} \mathbf{ref}(b +_{32} i)) \end{cases}$$

Using the value of the program counter to index the properties of the specification is similar to approach used by Burstall (1974) to apply the method of intermittent assertions.

The specification to be satisfied by region r is, for any $i, a, b, l, n \in \mathbb{N}$:

$$\vdash (pc =_{32} l_1 \wedge spec(i, a, b, l, n)) \Rightarrow \mathbf{wp}(r, pc =_{32} l \wedge spec(0, a, b, l, n))$$

Region r is verified by deriving the intermediate specifications to be established by commands of the region from the specification of the region. This approach contrasts with method described in Chapter 4 in which the program commands are shown to establish intermediate specifications from which the program specification is derived.

The verification of region r proceeds as follows:

1. By definition, $\vdash pc =_{32} spec(0, a, b, l, n) \Rightarrow final?(r)$. The predicate transformer **wp** can therefore be replaced with $\mathbf{wp}_1$ and the specification to be satisfied is:

$$\vdash (pc =_{32} l_1 \wedge spec(i, a, b, l, n)) \Rightarrow \mathbf{wp}_1(r, pc =_{32} l \wedge spec(0, a, b, l, n))$$

2. Rule (tl23) is applied for a proof by induction on i. There are two cases, when $i = 0$ and when $i > 0$. Let $r_1 = unit(C_2)$; in both cases of i. If follows that both $r_1 \in rest(r)$ and $r_1 < r$ are *true*. Also note that $head(r) = C_1$ and $head(r_1) = C_2$.

   (a) Base case, $i = 0$: By definition of $\mathbf{wp}_1$ it is only necessary to show that $\vdash pc =_{32} l_1 \wedge spec(0, a, b, l, n) \Rightarrow \mathbf{wp}_0(r, pc =_{32} l \wedge spec(0, a, b, l, n))$. This is by repeated application of rule (tl20) and rule (tl21).

$$\begin{aligned} &\vdash pc =_{32} l_1 \wedge spec(0, a, b, l, n) \\ &\quad \Rightarrow \mathbf{wp}(head(r), pc =_{32} l_2 \wedge spec(0, a, b, l, n)) \\ &\vdash pc =_{32} l_2 \wedge spec(0, a, b, l, n) \\ &\quad \Rightarrow \mathbf{wp}(head(r_1), pc =_{32} l \wedge spec(0, a, b, l, n)) \end{aligned}$$

   Both these assertions can be established from commands C_1 and C_2. The specification $\vdash pc =_{32} l_1 \wedge spec(0, a, b, l, n) \Rightarrow \mathbf{wp}_0(r, pc =_{32} l \wedge spec(0, a, b, l, n)$ follows from the proof rules.

(b) Inductive case, $i > 0$: The proof is similar to that for the base case with the exception that the specification must establish the inductive hypothesis. This requires assertions $\mathit{spec}(i-1, a, b, l, n)$ and $\mathit{enabled}(r')$ (for some $r' < r$) to be established by $\mathbf{wp}_0(r, pc =_{32} l_2 \wedge \mathit{spec}(i-1, a, b, l, n))$. Since the postcondition to be established in this case includes $pc =_{32} l_2$, a region r' can be constructed as $r' = \mathit{region}(l_2, \mathit{strcopy}_1)$. The label of region r' is l_2 and r' is enabled when $pc =_{32} l_2$. By definition of *region*, the only command in r' is C_2; region r' is therefore region r_1, $r' = \mathit{unit}(C_2)$. Control does not pass from command C_2 to command C_1: $\neg(C_2 \longmapsto C_1)$.

The proof, as for the base case, is by the application of rule (tl20) and rule (tl21). The assertion to be proved is $\vdash pc =_{32} l_1 \wedge \mathit{spec}(i, a, b, l, n) \Rightarrow \mathbf{wp}_0(r, pc =_{32} l_2 \wedge \mathit{spec}(i-1, a, b, l, n))$. This requires the following assertions:

$$\begin{array}{l}\vdash pc =_{32} l_1 \wedge \mathit{spec}(i, a, b, l, n) \\ \quad \Rightarrow \mathbf{wp}(\mathit{head}(r), pc =_{32} l_2 \wedge \mathit{spec}(i, a, b, l, n)) \\ \vdash pc =_{32} l_2 \wedge \mathit{spec}(i, a, b, l, n) \\ \quad \Rightarrow \mathbf{wp}(\mathit{head}(r_1), pc =_{32} l_2 \wedge \mathit{spec}(i-1, a, b, l, n))\end{array}$$

Showing that r establishes $pc =_{32} l_2 \wedge \mathit{spec}(i-1, a, b, l, n))$ allows the inductive hypothesis to be satisfied. The hypothesis establishes the property $\vdash \mathit{enabled}(r_1) \wedge \mathit{spec}(i-1, a, b, l, n) \Rightarrow \mathbf{wp}(r_1, pc =_{32} l \wedge \mathit{spec}(0, a, b, l, n))$. From rule (tl22) and from the assertion $\vdash pc =_{32} l_1 \wedge \mathit{spec}(i, a, b, l, n) \Rightarrow \mathbf{wp}_0(r, pc =_{32} l_2 \wedge \mathit{spec}(i-1, a, b, l, n))$, this establishes:

$$\vdash pc =_{32} l_1 \wedge \mathit{spec}(i, a, b, l, n) \Rightarrow \mathbf{wp}_1(r, pc =_{32} l \wedge \mathit{spec}(0, a, b, l, n))$$

This completes the proof for this case.

Verifying that the region r establishes its specification is enough to show that the program $\mathit{strcopy}_1$ satisfies:

$$\vdash [pc =_{32} l_1 \wedge \mathit{spec}(n, a, b, l, n)]\mathit{strcopy}_1[pc =_{32} l \wedge \mathit{spec}(0, a, b, l, n)]$$

From the weakening rule (tl10) and the strengthening rule (tl11) for programs, and from the truth of assertions:

$$\begin{array}{l}\vdash pc =_{32} l_1 \wedge \mathit{Pre}(a, b, l, n) \Rightarrow pc =_{32} l_1 \wedge \mathit{spec}(n, a, b, l, n) \\ \vdash pc =_{32} l \wedge \mathit{spec}(0, a, b, l, n) \Rightarrow pc =_{32} l \wedge \mathit{Post}(a, b, l, n)\end{array}$$

the specification of the program, $\vdash [\mathit{Pre}(a, b, l, n)]\mathit{strcopy}_1[\mathit{Post}(a, b, l, n)$, is established from the specification of the region.

6.3.4 Verifying Regions of a Program

Because a region of a program can be considered as a program, verifying that a region establishes a postcondition is equivalent to verifying a program. The *wp* function for the regions is therefore an instance of the methods of Floyd (1967) and Burstall (1974) for proving program correctness. The weakest precondition functions for regions of a program allow program verification to be based on the flow-graph of a program. The transformation T_2 requires the ability to mechanically construct regions of a program. The weakest precondition transformers for regions allow the use of the tools which construct regions during program verification.

The weakest precondition transformers can also be used to establish the correctness of region transformations which consider only the external behaviour of a region. These are typically the transformations used in compilation and code optimisation. If the compiling transformation T_c is applied to region r, it must construct a region $T_c(r)$ which terminates whenever r terminates and which produces the same states as r: $\mathcal{I}_r(r)(s,t) \Rightarrow \mathcal{I}_r(T_c(r))(s,t)$. This is the definition of refinement in structured languages (see Chapter 4) and can be specified by the **wp** predicate transformer (as $\vdash \mathbf{wp}(r,Q) \Rightarrow \mathbf{wp}(T_c(r),Q)$, for any $Q \in \mathcal{A}$). Code optimising transformations are generally defined on the flow-graph of a region and the **wp** function allows the use of the flow-graph to verify the optimising transformation.

Because only the external behaviour of a region is considered, the **wp** transformer is not enough when considering program refinement and abstraction. Program refinement considers all states produced by a program (such as the body of a region) while the interpretation of regions considers only the states in which a region begins and ends. The interpretation of regions is therefore not enough when considering the abstraction of programs: for $r_1, r_2 \in \mathcal{R}$, it is not possible to establish $\mathit{body}(r_1) \sqsubseteq \mathit{body}(r_2)$ from $\forall s,t : \mathcal{I}_r(r_1)(s,t) \Rightarrow \mathcal{I}_r(r_2)(s,t)$. Furthermore, the **wp** transformer cannot be used to show that abstracting transformations are correct. If region r contains an infinite loop and never terminates, $\neg\mathcal{I}_r(r)(s,t)$ for all $s,t \in \mathit{State}$, there is no region r' such that $\vdash \mathbf{wp}(r',Q) \Rightarrow \mathbf{wp}(r,Q)$ (because this requires $(\exists t : \mathcal{I}_r(r')(s,t)) \Rightarrow \exists t : \mathcal{I}_r(r)(s,t)$).

The refinement relation of regions is a more powerful approach to comparing the behaviour of regions than the *wp* transformers for regions. The refinement relation (as well as transformations T_1 and T_2) considers both the internal and the external behaviour of regions. The internal behaviour of a region allows refinement between regions to establish refinement between the programs making up the regions. This reflects the main use of regions as a vehicle for transforming programs, rather than as a method for structuring programs during verification. Because both the internal and external behaviour of a region are considered by the refinement relation, it is possible to show that transformations abstract from regions; that the transformed region also abstracts from programs and that the abstraction preserves the failures of the regions.

6.4 Conclusion

Processor languages differ from each other in the syntax of the instructions and in the data operations used by the instructions. However, these differences are superficial. The object code of all processor languages are programs which make changes to variables. The instructions and data operations of a processor do not affect the basic function of a program, which is to establish the properties required by its specification. It is therefore unnecessary to develop proof methods especially for the verification of object code. Any such proof method will be equivalent to the method of inductive assertions (Floyd, 1967) or the method of intermittent assertions (Manna, 1974; Burstall, 1974). This follows from the fact that object code is simply a program of a processor language. The method used to verify object code is therefore the same as the method used to verify any program: the properties of interest are described in terms of program variables and proof is based on the paths and loops in the program flow-graph. The only features introduced by object code programs are the models for data, which allow pointers, and for execution, which permit interleaving loops. The properties of these features can be established using operators provided by the language $\mathcal{L}$ and the methods of Floyd (1967) or Burstall (1974).

A number of proof methods have been developed to simplify reasoning about programs (see Cousot, 1981). These are, generally, intended for program verification in a system of logic, rather than by reasoning directly about the semantics of a programming language. The application of these methods to programs of the abstract language $\mathcal{L}$ is straightforward. Two proof methods were described in this chapter. The first is based on data refinement (Morris, 1989) and provides a means for justifying a single proof of correctness for a program implemented on a number of processors. The approach taken relies on the use of a program logic to describe the effect on specification of implementing a program in a processor language. This allows the implementation of a correct program to be verified by showing that it data refines the program.

The second proof method allows the flow-graph of a program to be used in a proof of correctness. This presumes that tools for constructing and manipulating a region are available. This assumption can be satisfied by a system which abstracts from programs using the program transformations T_1 and T_2. Separate tools are straightforward to construct, using the flow-analysis techniques of code optimisation (Hecht, 1977; Aho, Sethi & Ullman, 1986). The method for reasoning about flow-graphs using the weakest precondition functions, $\mathbf{wp}_1$ and $\mathbf{wp}$, is essentially the method of intermittent assertions (Manna, 1974; Burstall, 1974). The main feature of this proof method is that it can be applied to any program of $\mathcal{L}$. As a consequence, it is an example of a proof method which can be applied to the object code of a range of processors.

The proof methods described in this chapter are examples of a general feature of the abstract language $\mathcal{L}$. Because the correctness of $\mathcal{L}$ programs is independent of any particular proof method, the proof methods which have developed for program verification can be applied to the programs of $\mathcal{L}$. The two proof methods described in this chapter can both be applied to any program implemented in any processor language. The first provides a way of simplifying the verification of a single program implemented across different processors. The second exploits a feature (the program flow-graph) common to all object code programs of all processor languages. This reflects one of the main uses of the language $\mathcal{L}$: to generalise methods for verifying programs across different processor languages. Developing the methods for the language $\mathcal{L}$ means they can be applied to verify arbitrary object code programs.

Chapter 7

Verification of the Theory

The methods described in this thesis are intended, first, to allow object code programs to be verified and, second, to simplify verification by manipulating the text of programs. To ensure the correctness of the methods, the theory presented in Chapters 3, 4 and 6 has been verified using the PVS theorem prover (Owre et al., 1993). This allows the theorems and rules for manipulating programs of $\mathcal{L}$ to be used without the need to repeat the proof of the theory. It also ensures that the language $\mathcal{L}$ can be defined in terms which can be manipulated by a theorem prover. This allows the methods described in this thesis to be implemented as an automated proof tool and also allows a wide range of techniques to be used for the implementation. For example, the rules for the substitution operator can be implemented either by a simplification procedure (applying the rules as theorems of a logic) or by a decision procedure (which operates directly on the $\mathcal{L}$ expressions).

The verified theory includes all theorems and lemmas, the substitution rules and the proof rules of the program logic. It does not include the example programs, which are not part of the theory needed for program verification. There is a difference between the verification of a mathematical theory and its implementation as an automated proof tool for program verification. The work with PVS verifies the theory on which automated tools will be based. The implementation of a proof tool must consider the issues common in program development, such as the representation of the language $\mathcal{L}$ and the algorithms used to analyse and manipulate $\mathcal{L}$ programs. The implementation must also take into account issues specific to automated proof tools, such as the procedures needed to reason about the logical formulas making up a program specification. Some of the issues involved in implementing the methods described in this thesis as a proof tool will be considered. However, the implementation of proof tools is outside the scope of this thesis. A description of methods for implementing proof tools is given by Duffy (1991) and Boulton (1994), among others.

This chapter describes the use of the PVS system to verify the theory and the relationship between the proofs carried out in PVS and the mathematical proofs of the theorems and lemmas (given in Appendix C and Appendix D). The PVS system is described in **Section 7.1**. This is followed, in **Section 7.2**, by a description of the use of PVS to develop the theory. This will also consider the relationship between mathematical and mechanical proofs. The verification of the theory in PVS will be described in **Section 7.3**. The implementation of the theory in a proof tool is discussed in **Section 7.4**.

7.1 The PVS System

The PVS system is based on an interactive theorem prover for typed higher order logic (Owre, Rushby & Shankar, 1993) and is intended for reasoning about logical specifications. A number of facilities simplify the use of PVS, including a user interface and automated management of specifications and proofs. Specifications are defined as theories, made up of definitions, axioms and theorems, in a specification language based on higher order logic. The PVS specification language supports user defined data types and function definition by primitive recursion. A proof is carried out manually, by issuing commands which invoke the proof procedures of the theorem prover. The theorem prover also provides a simple scripting language, to allow the definition of commands using the primitive commands of the proof tool. The proof procedures of PVS include rewriting, decision procedures for simple logical formulas and for integer arithmetic. A proof in PVS (as in other theorem provers) is the series of commands which must be issued to the theorem prover to show that a formula is true.

Proof tools have been developed which use the PVS system to simplify and carry out proofs (Skakkebaek and Shankar, 1994; Jacobs et al., 1998). These are based on implementing alternative user interfaces to the PVS system, rather than by making direct use of the PVS components. Support for extensions to the PVS system is limited and more restrictive than is provided by tools such as the Isabelle or HOL theorem provers (Paulson, 1995a; Gordon & Melham, 1993). The Isabelle and HOL theorem provers are intended to provide a framework in which proof tools can be implemented. These systems support the development of proof tools based on the theorem provers by providing and documenting access to the procedures and data representation of the theorem prover. In contrast, access to the procedures of the PVS theorem prover is limited and extending the PVS system is a non-trivial task. This reflects the PVS system's intended use as a general proof tool for reasoning about specifications rather than a framework in which other proof tools can be developed.

7.1.1 The PVS Specification Language

A PVS specification is organised as modules, called *theories*, in which types, functions and variables are specified and logical formulas given as axioms or theorems. Functions and types can be arguments to modules and can also be referred to by other modules. Logical formulas can be presented as axioms, which are assumed to be *true*, or theorems, which must be proved. A logical formula is any expression ranging over the booleans. The usual operators are available and include logical equivalence, syntactic equality, the universal and existential quantifiers and the Hilbert epsilon. Formulas expressing equivalence can be used by the prover as rewrite rules.

A type is either declared without definition or is defined by enumeration, as an abstract data type or as a subtype by predicates on the values of an existing type. Types provided by PVS include the natural numbers, booleans and infinite sequences of a given type. The definition of an abstract data type (ADT), which may be parameterised, generates a theory containing the definitions and declarations needed for its use. In particular, the sub-term relation, $\ll$, is generated by the PVS theorem prover for each recursively defined abstract data type. The definition of an ADT has the form:

$$
\begin{array}{l}
\mathit{name}[p_1, \ldots, p_n] : \texttt{datatype} \\
\texttt{begin} \\
\mathit{cons}_1(\mathit{acc}_1 : T_{(1,1)}, \ldots, \mathit{acc}_k : T_{(1,k)}) : \mathit{rec}_1 \\
\qquad\qquad\qquad\vdots \\
\mathit{cons}_m(\mathit{acc}_1 : T_{(m,1)}, \ldots, \mathit{acc}_{(m,j)}) : \mathit{rec}_m \\
\texttt{end}\ \mathit{name}
\end{array}
$$

The ADT identifier is *name*, each p_j is a parameter to the type and each cons_i is a constructor. For each constructor cons_i there are one or more accessor functions acc_l ranging over type $T_{(i,l)}$. Each rec_i is a predicate on instances of the ADT acting as a recogniser for the associated constructor, cons_i, such that $\mathit{rec}_i(x)$ for x of type *name* is *true* iff x is constructed with cons_i.

Predicates are functions ranging over booleans and sets of a type are treated as predicates on elements of that type. Functions may be defined by primitive recursion and functions, and some operators, may be overloaded. A limited form of polymorphism is supported: if type T is an argument to a theory module then functions can be defined with types dependent on T; the argument T will be instantiated when the functions are used. A conditional is provided by an `if-then-else-endif` expression for which both branches must be provided. An expression for pattern matching on an instance of an ADT is available and is equivalent to a nested conditional using the recognisers of the ADT.

7.2 Development of the Theory

The PVS system was used both to develop and to verify the theory described in this thesis. The use of PVS was intended to ensure the correctness of the theory and in particular the theorems and rules needed to manipulate programs. The theory verified using PVS is made up of the definitions, lemmas and theorems of Chapters 3, 4 and 6. It also includes the rules for the substitution operator (Figure 3.3) and for the program logic (Figures 3.5, 4.7 and 6.1). The PVS system was also used to refine and develop the theory as errors were found and corrected. This process used the PVS system to experiment with alternatives to the models used as the theory was developed. More generally, the PVS system was used as a tool to manage and check the details of the manually developed theory.

The development of the theory described in this thesis proceeded as follows: first, the theory was developed without the use of PVS, by working out (on paper) an informal model of the components needed to verify object code (the expressions, commands, programs and regions). This included the methods needed to abstract from programs and the theorems which were required to establish the correctness of the methods. The informal model was then formally defined as a PVS specification. As each part was defined in the PVS specification language, attempts were made to prove the theorems and lemmas describing the properties required of that part of the theory. These attempts often exposed errors in the initial development which required amendments to the theory. The theory was refined by correcting and extending the models used, to allow additional properties to be established and used in proofs. These corrections were mainly carried out in PVS, as the result of experiments with extensions and alternative to the models. The completed development in PVS consists of specification files totalling approximately 8200 lines (172000 characters) and proof files made up of approximately 86000 lines (9900000 characters).

The result of the work with PVS is two theories, the first made up of the definitions and theorems described in this thesis and the second made up of the definitions and theorems defined in the PVS specification language. The differences between the two theories are in the precise form of the definitions and in the proofs of the two sets of theorems. The theories make the same assumptions and their theorems are equivalent; the two theories are therefore equivalent.

7.2.1 Mechanical and Mathematical Proofs

The work with PVS resulted in a set of proofs establishing the correctness of the theorems and lemmas making up the theory. These proofs are made up of the commands to the theorem prover needed to establish the truth of a theorem or lemma. As a consequence, the mechanical proof is highly dependent on the PVS theorem prover. However, a mathematical proof must be independently verifiable. A proof in PVS cannot be considered a mathematical proof since it is difficult to read and check that a PVS proof is correct without the use of PVS. Any theorem prover can contain errors and a proof which depends on the PVS theorem prover cannot be considered sufficient to establish the correctness of the theory. It can only be considered as increasing confidence in a separate, mathematical proof.

Separate mathematical proofs of the theorems and lemmas were developed and are given in Appendix C, Appendix D and Appendix E. Because these can be verified independently of a theorem prover, they are considered to be the main proof of the theory described in this thesis. The mathematical proofs are based on the structure of the PVS proof, which determines the general approach used to establish a property. The development of the mathematical proofs was otherwise independent of the PVS proofs. In particular, the mathematical proofs are not simply a human-readable transcript of the PVS proof. Consequently, the theory described in this thesis has been verified twice: once with an automated proof tool and once manually. The manual (mathematical) proof allows the correctness of the theory to be independently verified while the proof in PVS increases confidence in the mathematical proof and therefore in the theory.

7.3 Verifying the Theory in PVS

The development of the theory in PVS is based on a description in the PVS specification language of the definitions, the theorems and the lemmas which make up the theory. There are a number of differences between the theory described in the thesis and the PVS specification. These differences are principally in the methods used to define constructs of the language $\mathcal{L}$ in the PVS specification language. However, the differences are minor and the theory defined in the thesis is equivalent to that defined in PVS. There are greater differences between the lemmas used in the mechanical and mathematical proofs. A proof in a theorem prover is carried out using simpler concepts than is needed in a mathematical proof. This requires properties, which can be assumed in a mathematical proof, to be explicitly stated and proved for the mechanical proof. For example, the fact that any subset of a finite set is also finite (Levy, 1979) can be assumed in a mathematical proof but must be explicitly proved in a mechanical proof.

7.3.1 Specification of the Theory in PVS

The greater part of the theory described in this thesis translates immediately to the PVS specification language. In particular, the theory concerning the commands, the majority of Chapter 3, and programs and regions, that of Chapter 4, can be translated directly to the PVS specification language. The theory concerning the weakest precondition for regions and the data refinement between programs, in Chapter 6, can also be translated easily to PVS. The model of the expressions used in the PVS theory differs from that of Chapter 3 in the definition of the basic types, *Values*, *Names* and *Labels* and the definition of the expressions $\mathcal{E}$. As a consequence, the definitions in PVS of the functions on these types also differ from those given in Chapter 3. These differences are minor: the theories defined for PVS and those given in Chapters 3, 4 and 6 are equivalent. The remainder of this section will describe the differences between the definition in PVS of the theory and the definitions given in this thesis.

Model of the Basic Expressions

The basic expressions are constructed from an undefined type T, by which a processor represents data, and a register identifier type *RT*. The values, variables, labels and registers are contained in the sets *BValues*, *BVars*, *BLabels* and *BRegs* respectively. The sets are undefined but are assumed to be non-empty. The set *BRegs* has type *RT*, the sets *BValues* is of type T and the sets *BVars* and *BLabels* are subsets of *BValues*. An abstract data type, *Basic*, containing the basic expressions is the disjoint union of the sets *BValues*, *BVars* and *BLabels*.

$$\begin{array}{l} \textit{Basic} : \texttt{datatype} \\ \texttt{begin} \\ \textit{bval}(\textit{val} : \textit{BValues}) : \textit{bval?} \\ \textit{bvars}(\textit{var} : \textit{BVars}) : \textit{bvar?} \\ \textit{breg}(\textit{reg} : \textit{BRegs}) : \textit{breg?} \\ \texttt{end}\ \textit{Basic} \end{array}$$

The predicates *bval?*, *bvar?* and *breg?* are recognisers for elements of *Basic* representing the basic values, variables and registers respectively. These define the recognisers for the labels and names: a basic label is a basic value constructed from the set *BLabels*; a basic name is a basic variable or a basic register.

$$\begin{array}{l} \textit{blabel?}(x) \stackrel{\text{def}}{=} \textit{bval?}(x) \wedge \textit{val}(x) \in \textit{BLabels} \\ \textit{bname?}(x) \stackrel{\text{def}}{=} \textit{bvar?}(x) \vee \textit{breg?}(x) \end{array}$$

The type *Basic* represents the constants over which the functions of $\mathcal{L}$ range. The subtypes of *Basic* containing the basic names and the basic values define the set of states.

$$\begin{array}{l} \textit{BasicNames} \stackrel{\text{def}}{=} \{x : \textit{Basic} \mid \textit{bname?}(x)\} \\ \textit{BasicValues} \stackrel{\text{def}}{=} \{x : \textit{Basic} \mid \textit{bval?}(x)\} \end{array}$$

In the PVS theory, a state is function from basic names to values.

$$\textit{State} \stackrel{\text{def}}{=} \textit{BasicNames} \to \textit{BasicValues}$$

Model of the Expressions

The expressions $\mathcal{E}$ are defined as the basic expressions, the substitution expression, the application of a function to an expression and an expression pair. The functions are applied to a single expression and, as in Chapter 3, the set of function names is $\mathcal{F}$. The substitution operator is defined separately from the expressions, $\mathcal{E}$, and the substitution expression is a function from states to basic values. The set $\mathcal{E}_0$ contains all expressions.

$$\begin{array}{l} \mathcal{E}_0 : \texttt{datatype} \\ \texttt{begin} \\ \textit{base}(\textit{basic} : \textit{Basic}) : \textit{base?} \\ \textit{subst}(\textit{substfn} : [\textit{State} \to \textit{BasicValues}]) : \textit{subst?} \\ \textit{pair}(\textit{left} : \mathcal{E}_0, \textit{right} : \mathcal{E}_0) : \textit{pair?} \\ \textit{apply}(\textit{fn} : \mathcal{F}, \textit{arg} : \mathcal{E}_0) : \textit{apply?} \\ \texttt{end}\ name \end{array}$$

The set of expressions $\mathcal{E}$ is the subset of $\mathcal{E}_0$ in which the constructor *pair* occurs only as a sub-term of an expression.

$$\mathcal{E} \stackrel{\text{def}}{=} \{x : \mathcal{E}_0 \mid \neg \mathit{pair?}(x)\}$$

e.g. The expression $\mathit{apply}(f, \mathit{pair}(x, y)) \in \mathcal{E}$ but $\mathit{pair}(x, y) \notin \mathcal{E}$. The set of names, values and labels are defined as subtypes of $\mathcal{E}_0$.

$$\begin{aligned}
\mathit{Names} &\stackrel{\text{def}}{=} \{x : \mathcal{E}_0 \mid \mathit{base?}(x) \wedge \mathit{bname?}(\mathit{basic}(x))\}\\
\mathit{Values} &\stackrel{\text{def}}{=} \{x : \mathcal{E}_0 \mid \mathit{base?}(x) \wedge \mathit{bval?}(\mathit{basic}(x))\}\\
\mathit{Labels} &\stackrel{\text{def}}{=} \{x : \mathcal{E}_0 \mid \mathit{base?}(x) \wedge \mathit{blabel?}(\mathit{basic}(x))\}
\end{aligned}$$

A function is applied to a single expression and a function with arity greater than 1 is applied to an expression constructed from the function *pair* applied to elements of $\mathcal{E}_0$. e.g. The application of function f to arguments x, y, z, $f(x, y, z)$, can be represented as $\mathit{apply}(f, \mathit{pair}(x, \mathit{pair}(y, z)))$. The label and name expressions $\mathcal{E}_n$ and $\mathcal{E}_l$ are defined as in Chapter 3. The expressions resulting from application of the constructor *subst* are values and $\mathit{subst}(f) \notin \mathcal{E}_n$.

The semantics of the expressions are defined by primitive recursion on the type $\mathcal{E}_0$, unfolding the definition of $\mathcal{I}_n$ in the definition of $\mathcal{I}_e$. The interpretation of constructor *subst* in state s is the application of the argument of the constructor *subst* to s. The result is an element of *Basic* and an expression of $\mathcal{E}_0$ is obtained by the application of the constructor *base*:

$$\mathcal{I}_e(\mathit{subst}(f))(s) \stackrel{\text{def}}{=} \mathit{base}(f(s))$$

The interpretation of an expression pair is the pair constructed from the interpretation of the arguments.

$$\mathcal{I}_e(\mathit{pair}(x, y))(s) \stackrel{\text{def}}{=} \mathit{pair}(\mathcal{I}_e(x)(s), \mathcal{I}_e(y)(s))$$

The interpretation function $\mathcal{I}_n$ on name expressions is defined by applying the function $\mathcal{I}_e$ to the arguments of name functions.

$$\mathcal{I}_n(x)(s) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } \mathit{base?}(x) \\ \mathcal{I}_f(\mathit{fn}(x))(\mathcal{I}_e(\mathit{arg}(x))(s)) & \text{if } \mathit{apply?}(x) \end{cases}$$

The type *Alist* is an abstract data type and the definition follows that given in Chapter 3. The functions *find* and *update* are also as given in Chapter 3. The predicate *correct?* on correct assignment lists is that given in Section C.2.3 of the appendix. The substitution operator $\triangleleft$ is defined by recursion on the type $\mathcal{E}_0$. The substitution of al in an expression $x \in \mathcal{E}$ is formed from the constructor *subst*. The substitution of assignment list al in expression $\mathit{pair}(x, y)$ is the substitution of al in x and y.

$$\begin{aligned}
&_ \triangleleft _ : \mathcal{E}_0 \to \mathcal{E}\\
&x \triangleleft al \stackrel{\text{def}}{=} \begin{cases} \mathit{subst}(\lambda(s : \mathit{State}) : \mathcal{I}_e(x, \mathit{update}(s, al))) & \text{if } \neg\mathit{pair?}(x) \\ \mathit{pair}(\mathit{left}(x) \triangleleft al, \mathit{right}(x) \triangleleft al) & \text{otherwise} \end{cases}
\end{aligned}$$

The definition is well-founded since every expression constructed by *pair* is well-founded.

Models of the Commands and Programs

The commands and the programs are as defined in Chapters 3 and 4. The most significant variation is in the definition of sequential composition. In the PVS theory, the composition of an assignment command (Definition 3.24 and Definition 3.25) and the composition of labelled and conditional commands (Definition 3.23) are defined as two functions. The composition of an assignment command is by a function *compose_assign* with type

$$\textit{compose\_assign} : (\textit{Alist} \times \mathcal{E}_l \times \mathcal{C}_0) \to \mathcal{C}_0$$

which is defined as in Definition (3.24). For $al \in \textit{Alist}, l \in \mathcal{E}_l$ and $c \in \mathcal{C}_0$, the result of the composition $\textit{compose\_assign}(al, l, c)$ is the result of $(:= (al, l)); c$. Sequential composition of commands $c_1, c_2 \in \mathcal{C}_0$ is defined as $\textit{compose\_assign}(al, l, c_2)$ if $c_1 = (:= (al, l))$ for some $al \in \textit{Alist}$ and $l \in \mathcal{E}_l$, otherwise it is as defined in Definition (3.23).

The specification of programs and regions required a PVS theory for finite sets. This was obtained by translating part of a library provided with the HOL theorem prover to a PVS theory. This was then extended with the additional theorems and lemmas required to model the programs. Although the PVS system provides a library for finite sets, this is based on the cardinality of a set. This complicates reasoning about programs, which is based on the addition of commands to a program. Furthermore, the PVS library required more machine resources than was needed by the theory translated from the HOL theorem prover.

While there are other differences between the model of programs and regions defined in PVS and the model defined in Chapter 4 , these are minor. For example, the definition of *trace* in the PVS theory has type $(\textit{Set}(\mathcal{C}) \times \textit{State} \times \textit{State}) \to \textit{boolean}$ while in the definition of Chapter 3, the first argument has type $\mathcal{P}$.

7.4 Implementing the Theory

The methods described in this thesis are intended to allow the implementation of proof tools for verifying and abstracting object code programs. The usual approach to implementing proof tools is to extend an existing theorem prover with the logic and procedures needed to manipulate and reason about the domain of the proof tool (e.g. see Boulton, 1994). This allows the logic and the procedures of the proof tool to be based on the logic and procedures provided by the theorem prover. Methods for implementing a theory as a proof tool are well established (Gordon & Melham, 1993; Moore, 1994; Boulton, 1994). The implementation as a proof tool of the theory would not require any technique other than those which are commonly used in automated theorem proving. The requirements of a proof tool for verifying object code based on the work in this thesis will be briefly summarised here.

A proof tool for object code verification requires an assertion language, in which to specify programs, a representation of the language $\mathcal{L}$ and procedures for manipulating the expressions, commands and programs of $\mathcal{L}$. A first order logic together with the natural numbers and specification operators for commands and programs is sufficient for the assertion language used to specify programs. The assertion language must also provide proof rules, such as those of $\mathcal{A}$, for reasoning about the expressions, commands and programs of $\mathcal{L}$. Because program specifications are based on the $\mathcal{L}$ expressions modelling the data operations of a processor, a proof tool must also provide procedures to simplify logical formulas which include properties of the natural numbers. Such procedures are well known (Shostak, 1977; Moore, 1994; Boulton, 1994).

To simplify verification by abstracting programs, a proof tool must mechanise the application of the sequential composition operator and implement the transformations on regions. Implementing sequential composition is a straightforward application of its definition (see Section 3.3), specialised to the representation by the proof tool of $\mathcal{L}$ commands. The proof tool must also provide procedures for simplifying the result of abstracting commands. These can be based on the substitution rules of Figure (3.3) and on procedures for reasoning about equivalence relations and for rewriting (Duffy, 1991). The abstraction of programs will be based on sequential composition or on the region transformations T_1 and T_2. To implement these transformations requires the manipulation and analysis of flow-graphs. Efficient methods for representing and manipulating flow-graphs are described by Aho et al. (1986).

A proof tool for verification will normally be developed as an extension to an existing theorem prover. The theorem prover used must provide a high degree of control over the representation of the formulas of its logic. The proof tool must represent the language $\mathcal{L}$ and its assertion language in terms of the logic of the theorem prover. These representations will be used to determine the actions to be carried out by procedures of the proof tool. The theorem prover must therefore allow the terms of its logic to be examined and manipulated. It must also ensure that the result of such manipulation is consistent in the logic of the theorem prover, otherwise the proof tool will be unsound. Theorem provers which are suitable for implementing such a proof tool include Isabelle (Paulson, 1995a), HOL (Gordon & Melham, 1993) and COQ (Barras et al., 1997). These explicitly support the development of proof tools as extensions to the theorem prover. The PVS system is less suitable, since it does not provide the access required to the internal procedures and the representation of the logic used by the PVS theorem prover.

7.5 Conclusion

The differences between the theory defined in the PVS specification language and those defined in Chapters 3 and 4 are mainly concerned with the representation of the expressions. The definition of expressions use particular features of the PVS specification language. For clarity, the description in Chapter 3 uses a slightly different approach (which does not affect the correctness of the theory). The expressions of $\mathcal{L}$ are likely to cause the greatest difficulty if the theory is redefined in the specification language of an alternative theorem prover. The remainder of the theory, dealing with the commands and the program, is relatively straightforward and its definition in an alternative specification language should not be difficult.

The theory has been verified using the PVS theorem prover and a separate mathematical proof is given in the appendix of this thesis. The methods for proving the theory are based on the induction schemes for commands, programs and regions. Many of the proofs are based on the structure of the commands and regions and on the size of programs. The proofs of semantic properties of the transformation are indirect since the difference in the levels of abstraction between the transformed and the original region must be considered. These properties are established by relating the behaviour of a command in the transformed region to the behaviour of a sequence of commands in the original region.

Related work in automated theorem proving includes the definition of graph theory in the HOL theorem prover by Wong (1991) and Chin-Tsun Chou (1994). The definition of finite sets used in the work with PVS is similar to the methods described by Camilleri & Melham (1992) for inductive relations. Definitions of structured programming languages include the work by Gordon (1988) and Curzon (1992); Windley (1994) and Müller-Olm (1995) describe the definition of processor languages. The work at Computational Logic Inc. has resulted in the implementation of proof tools for the verification of both hardware and software (Bevier et al., 1989; Yuan Yu, 1992).

The work with the PVS theorem prover is intended to verify the correctness of the theory described in Chapter 3 and Chapter 4. It is not intended to define a proof tool for program verification, the definition of the theory in the specification language of a theorem prover will not result in an efficient proof tool. The development of a proof tool for the language $\mathcal{L}$ does not require any techniques other than those commonly used in theorem proving and in program optimisation. Such a proof tool can be used to verify any sequential object code program. The theory concerning the verification and abstraction of programs described in this thesis is not concerned with the techniques used to implement proof tools. Instead, it provides a method for applying existing techniques to construct proof tools which can simplify the task of verifying programs.

Chapter 8

Conclusion

This thesis has considered the verification of object code in a program logic. The methods used to construct the proof of correctness are the standard methods for program verification (Floyd, 1967; Manna, 1974; Burstall, 1974). The size of typical object code programs meant that it was necessary to provide a method for reducing the manual work needed to verify a program. The approach used is based on program abstraction by manipulating the text of a program. Because the manipulation of text is easily mechanised, this approach allows the efficient implementation of abstracting program transformations in a proof tool.

The transformation and verification of object code is made difficult by the number and extent of specialisation of processor instructions. Because of the number of instructions which must be considered, it is difficult to prove the correctness of a program transformation. The specialisation of instructions means that the simplifications which can be performed are restricted by the processor language rather than by the methods used to verify the program. The definition of a program logic for a processor language will also be complicated. A large number of inference rules will be required and the program logic will be difficult to reason with as well as being specialised to a single processor language.

These problems were solved by considering a processor language as an instance of an abstract language, $\mathcal{L}$. The language $\mathcal{L}$ is sufficiently expressive to define the semantics of instructions in sequential object code programs. Expressions of a processor language are defined as name or value expressions of the language $\mathcal{L}$ and the representation of a processor instruction as a command of $\mathcal{L}$ makes clear the action performed by the instruction. Defining the semantics of processor instructions in terms of the abstract language simplifies the definition of a program logic and of program transformations. In addition, the program logic and transformations defined for the abstract language can be applied to the programs of a range of processor languages.

A program of $\mathcal{L}$ can be simplified by the sequential composition of program commands. This was defined on the syntax of the commands by extending the algebraic rules of Hoare et al. (1987) to pointers and computed jumps. Sequential composition can be applied to arbitrary commands of a program and will result in an abstraction of the commands. Transformations T_1 and T_2 generalise sequential composition to regions of a program. The general transformation,

T_2, can be applied to an arbitrary region of a program and will also result in an abstraction of the region. Methods were described for applying sequential composition or the transformations to construct an abstraction of a program. These methods are consistent with the standard techniques for program verification.

A program logic can be defined for the language $\mathcal{L}$ using the techniques of Hoare (1969) and Dijkstra (1976). The program logic used in the examples is based on proof rules for a flow-graph language, similar to as those of Francez (1992), extended with a rule for the assignment command, similar to that of Cartwright and Oppen (1981). A program of $\mathcal{L}$ can be verified using standard proof methods for verification (Floyd, 1967; Manna, 1974; Burstall, 1974). The method of verification used in this thesis is described in Chapter 4 and based on the method of intermittent assertions (Manna, 1974; Burstall, 1974); alternative methods were described in Chapter 6.

8.1 Contribution of the Thesis

The main contributions of the work described in this thesis is a method for verifying any object code program and a method for reducing the manual work needed to carry out the verification. The methods use the abstract language $\mathcal{L}$ to model and abstract from object code programs. The use of an abstract language to model and manipulate object code programs is common in code optimisation techniques (Aho et al., 1986) but is novel for program verification. An abstract language allows the object code programs of a range of processor languages to be verified using the same tools and techniques. For example, the program logic $\mathcal{A}$ and method of verifying programs described in Chapter 4 were used to verify the different object code programs (of different processors) given in Chapter 5.

The ability to describe any object code program as a program of an abstract language with an equal number of commands is new. The language $\mathcal{L}$ is at least as expressive as any other sequential language used in verification: any sequential program which does not modify itself can be described as a program of $\mathcal{L}$. The language $\mathcal{L}$ is more expressive than the intermediate languages used in compilers. Any processor instruction which can be described as transforming a state can be described as a single command of $\mathcal{L}$. A single command of $\mathcal{L}$ can therefore describe the exact behaviour of a processor instruction. In compilers, a command of an intermediate language is translated to sequences of instructions; it cannot describe the behaviour of processor instructions. The ability to model object code as a program of $\mathcal{L}$ with an equal number of commands means that the difficulty of verifying the object code is not increased by its translation to $\mathcal{L}$.

The abstract language $\mathcal{L}$ provides the means for abstracting from programs by manipulating the text of a program. The method used to abstract program is an extension of the rules of (Hoare et al., 1987), which are limited to structured programs without pointers. The ability to abstract from programs with pointers and computed jumps and to describe the abstraction as a program is new. Since the result of abstraction is a $\mathcal{L}$ program, it is not necessary to distinguish between programs and their abstractions. This allows the abstraction of a program to be verified using the same methods as the original program.

Modelling Object Code for Verification

The features of $\mathcal{L}$ used to model object code are the commands (which model instructions) and the programs. The language $\mathcal{L}$ has three commands, which simplifies the development of techniques for verifying and transforming $\mathcal{L}$ programs. The programs of $\mathcal{L}$ are based on the execution model of object code, in which the flow of control is determined by instructions. A program of $\mathcal{L}$ is a set of $\mathcal{L}$ commands, each of which select their successor. This provides a flexible model of object code which simplifies the manipulation of $\mathcal{L}$ programs. It differs from the usual approach to modelling object code in which an object code program is described in terms of a structured language (Bowen and He Jifeng, 1994; Back et al., 1994).

The main features of the commands and expressions of $\mathcal{L}$ are the simultaneous assignment and the name and label expressions. The simultaneous assignment command permits assignments to arbitrary variables and pointers and is more general than the assignment commands previously considered. It is needed to allow a processor instruction to be modelled by a single $\mathcal{L}$ command. Pointers are modelled in $\mathcal{L}$ as name expressions, a novel approach which is simpler and more flexible than the models of arrays and pointers commonly used (Dijkstra, 1976; Manna & Waldinger, 1981). The execution model of $\mathcal{L}$ uses a program counter to select commands for execution. The value assigned to the program counter is the result of evaluating a label expression, providing the language $\mathcal{L}$ with computed jumps. The name and label expression are built up in the same way as the value expressions (from constants and functions). This means that methods for reasoning about name and label expressions are consistent with methods for reasoning about value expressions.

The expressions of $\mathcal{L}$ also support the definition of proof rules for $\mathcal{L}$; the assertion language $\mathcal{A}$ is an example of a program logic with such proof rules. The principal features of $\mathcal{A}$ are the specification of assignment commands and the treatment of control flow in programs. Assignments are specified using the substitution operator of $\mathcal{L}$, which generalises simultaneous textual substitution to allow its application in the presence of pointers. The proof rules for programs of $\mathcal{L}$ are based on the selection of commands by the use of the program counter. This approach is simpler than those previously proposed for flow-graph programs (Clint & Hoare, 1972; Jifeng He, 1983). Program logics for the language $\mathcal{L}$ can be used to verify any object code program (of any processor language) which is described in terms of $\mathcal{L}$. This provides a single, consistent approach to modelling and verifying the object code programs of any processor language.

Abstraction

The abstraction of $\mathcal{L}$ programs is based on the sequential composition of commands. This is defined as a function on the commands of $\mathcal{L}$; the definition generalises the rules for manipulating commands described by Hoare et al. (1987) to commands of a flow-graph language which include pointers and computed jumps. The interpretation of sequential composition used is new and consistent with the execution model of flow-graph languages (such as $\mathcal{L}$). The definition of sequential composition required the development of new constructs in the language $\mathcal{L}$, to take into account the undecidability introduced by pointers and computed jumps. The abstraction of assignment commands is based on a representation of assignment lists which allows the merger of assignment lists to be described syntactically. This is a new approach to abstracting assignment commands which avoids the aliasing problem (which is undecidable). The undecidability of computed jumps is avoided by the use of the the program counter to guard the result of composing two commands. This allows the application of sequential composition to any two commands in a program: the result will always be an abstraction of the commands.

The sequential composition operator of $\mathcal{L}$ is enough to abstract from any program of $\mathcal{L}$ but to mechanise program abstraction, it is necessary to define transformations on programs. To use a program transformation during verification, the transformation must be correct. A framework for defining and reasoning about transformations of $\mathcal{L}$ programs was described. The framework is based on regions, to analyse and transform programs, and traces, to reason about transformations. The regions are a novel concept; their properties include an induction scheme and the ability to define transformations by primitive recursion. This framework is more general than those used in code optimisation, since it can be applied to arbitrary programs regardless of the structure of the program's flow-graph. It is also more general than those used in program verification, which are developed for structured programs.

The framework was used to define transformations T_1 and T_2. Transformation T_1 is a straightforward application of sequential composition, which abstracts commands in a region in the order determined by the flow-graph of the region. The general transformation T_2 uses an analysis of a region's flow-graph to determine the cut-points in the regions. This analysis is more general than those used in code optimisation, since it is not limited by the structure of the flow-graph. Both transformations abstract from their arguments. Furthermore, the result of applying the transformations to a region r is semantically equivalent to r and preserves the failures of r. This ensures that, when verifying a program, it is enough to consider the abstraction of the program formed by applying the transformations.

Because the transformations and the sequential composition operator consider only the text of a program, mechanising program abstraction is both straightforward and efficient. The techniques needed to construct and manipulate regions are commonly used to develop compilers. The efficiency of these techniques is a consequence of the efficiency with which machines can carry out symbolic manipulation. For the same reason, verifying a program in a program logic is more efficient than reasoning about the semantics of a program. The work described in this thesis therefore provides a means for efficiently applying proof tools in the verification of a program. It also allows the application of proof tools to verify object code, regardless of the processor language in which the object code is written. This greatly reduces the manual work needed to verify an object code program.

8.2 Application

An object program can be verified either by the use of generic transformations and proof rules or by the use of transformations and rules specialised to the processor language. The first approach was used in the examples of Chapter 5 and has the advantage that the programs of a number of processors can be verified. The second allows the definition of rules and transformation to simplify the verification proof; examples of this approach are given in Chapter 6. The second approach is required for processor languages with a computation model in which the general transformations, T_1 and T_2, cannot be not applied directly. The SPARC processor is an example of such a language.

Abstractions of a program are constructed from the syntax of the program commands to allow their implementation in an automated tool for program verification. An approach to verifying programs with abstraction is described in Chapter 4. The use of sequential composition and transformations T_1 and T_2 in the verification of a program is consistent with the standard proof methods (Floyd, 1967; Manna, 1974; Burstall, 1974). Transformations T_1 and T_2 can reduce the number of commands in a region r, limited by the number of loops in r. If there are n loops then the result of $T_2(r)$ will contain no less then $n + 1$ commands, representing the n loops and the path from the head of r to the first loop in r. The result of constructing an abstraction will be a set of complex commands, made up of the expressions occurring in the original commands. These can be simplified using information about the values of names which occur in the commands and expressions.

To verify an object program, the processor instructions must be defined in terms of $\mathcal{L}$. Where an instruction is executed in a strictly sequential model, its definition in $\mathcal{L}$ is straightforward. However, some instructions of an otherwise sequential language imply a parallel execution model. For example, the Alpha AXP processor (Sites, 1992) defines a sequential execution model for application programs but includes floating point instructions whose semantics are described in a parallel execution model. The definition of the instructions can be modelled in terms of $\mathcal{L}$ but the resulting commands and programs will be complex and difficult to verify. A simpler alternative is to restrict an object program to ensure that the behaviour of these instructions can be defined in a sequential execution model. For example, to meet a standard for floating point operations, the Alpha processor manual recommends the use of floating point instructions in a form which forces sequential execution (Digital Equipment Corporation, 1996).

8.3 Related Work

Other work with verifying object code programs has mainly been concerned with showing the correctness of a compiler (see Hoare et al., 1993 and Müller-Olm, 1995) or verifying a processor design (Windley, 1994). The techniques used in these works are similar to the interpreter functions described by Boyer & Moore (1997). The models defined for an object code program are intended to show that a high-level command is correctly refined by a sequence of processor instructions or that an instruction is correctly implemented in hardware. The models are not intended for verifying the object code program.

Symbolic execution is a technique for mechanically constructing an abstraction of a program. The sequential composition of commands of $\mathcal{L}$ can be considered a form of symbolic execution. The transformation T_2 is the generalisation of sequential composition to a region of a program and is equivalent to the symbolic execution of the region. However, the result of both sequential composition and the transformations are defined as commands of $\mathcal{L}$ while the result of symbolic execution is a logical relation. Further transformations can be defined on the syntactic structure of commands and programs of $\mathcal{L}$ and these can be applied to both a program and its abstraction. A logical relation does not provide such a structure, transformations are less easily defined and cannot be applied to both program and abstraction.

Techniques for symbolic execution of programs are well known (see King, 1976) and include the execution of programs with pointers and multiple assignment commands (Colby, 1996). When the program includes pointers, the relation constructed by symbolic execution solves the aliasing problem by a series of comparisons between pointers. The properties described by the relation depend on the result of each comparison and this can lead to a large number of terms, making the relation difficult to manipulate. The relation constructed by symbolic execution is intended for use in program analysis and describes the semantic properties of commands. The structure of a verification proof will therefore depend on the properties of functions used in the definition of the semantics rather than on the syntax of the commands and flow of control through the program. In general, this will lead to a large and complex proof.

The work at Computational Logic Inc on program verification is also based on the semantic properties of instructions (Bevier et al., 1989; Yuan Yu, 1992). The techniques described by Boyer & Moore (1997) for object code verification are specialised to a particular method of proof and use interpreter functions to model the behaviour of a processor (Boyer & Moore, 1997). Verification is based on reasoning about the behaviour of the interpreter rather than the behaviour of the program. Because of the complexity of an interpreter function, this makes manual intervention in the proof difficult. The use of an interpreter function also hides the similarities between processor languages and leads to a replication of work when modelling object programs. To verify object programs of different processors, separate interpreter functions for the two processors must be constructed and the properties of these functions established.

In contrast, the use of the language $\mathcal{L}$ to model an object code program allows the standard techniques for program verification to be applied. The processor language does not play a great as role as in systems using an interpreter function. Once the instructions of an object programs are defined in terms of the commands of $\mathcal{L}$, the processor language does not feature in the verification. This allows the implementation of generic proof tools for object code verification and such a tool can be applied consistently to the programs of a range of processors.

8.4 Verifying Safety Properties

The techniques described in this thesis are intended to prove the liveness properties of a program. Safety properties are properties which must be true for every state produced by the program but, by a simple transformation of the program, a safety property can be verified as a liveness property. The transformation does not alter the basic requirement that every state produced by the program must be shown to have the property. However, it may reduce the number of steps in the proof by allowing the verification to be based on an abstraction of the program.

Assume that Ψ is a safety property which must be established by program p, property Ψ is an assertion on a state, $\Psi \in \mathcal{A}$. To transform the safety property to one of liveness, the program p must halt if a command does not preserve the property Ψ. A state is changed only by an assignment command and the program must halt if an assignment command beginning in state s, such that $\Psi(s)$, produces a state t such that $\neg\Psi(t)$. To achieve this, the assignment commands are made conditional on the preservation of the property. The conditional command of $\mathcal{L}$ is extended so that the test is an assertion of $\mathcal{A}$ rather than a boolean expression of $\mathcal{E}_b$. The extended constructor function will have type:

$$\textbf{if}\ \_\ \textbf{then}\ \_\ \textbf{else}\ \_ : (\mathcal{A} \times \mathcal{L}_0 \times \mathcal{L}_0) \to \mathcal{L}_0$$

and interpretation:

$$\mathcal{I}(\textbf{if}\ a\ \textbf{then}\ c_1\ \textbf{else}\ c_2)(s,t) \stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(c_1)(s,t) & \text{if } a(s) \\ \mathcal{I}(c_2)(s,t) & \text{otherwise} \end{cases}$$

For safety property Ψ, every assignment command $:= (al, l)$ in program p is transformed to the command:

$$\textbf{if}\ \Psi \triangleleft ((pc, l) \cdot al)\ \textbf{then}\ := (al, l)\ \textbf{else abort}$$

This command terminates only if the assignment establishes the safety property and halts otherwise.

Assume that the safety property Ψ is to be established by program p beginning at the command labelled l_1 and ending at command labelled l_2. Transforming the assignment commands of p results in program p'. The property to be proved is that the program p' beginning in a state satisfying Ψ terminates.

$$\vdash [(pc = l) \wedge \Psi]p'[pc = l_2]$$

The transformations for abstracting from the program can then be applied to program p'.

8.5 Extending the Work

The programs considered here are those of processor languages, which have relatively simple data types and constructs. High-level languages include more complex data types and the control constructs of the structured languages. A high-level program can be defined in terms of a processor language, and therefore in terms of $\mathcal{L}$. However, the verification of a high-level program will be simplified by augmenting the language $\mathcal{L}$ with the data types and control constructs of the high-level language and defining the program in terms of this augmented language. In general, the addition of high-level data types, and expressions of these types, to the language $\mathcal{L}$ is straightforward. The set *Values* is extended with a representation of the additional types and the expressions of $\mathcal{L}$ are defined to include the functions on the data types.

The language $\mathcal{L}$ excludes a form of expression, found in high-level languages, which are constructed from the application of functions to name expressions and which result in a value. For example, the language C includes an operator $\&$ with type $\mathcal{E}_n \rightarrow \mathit{Values}$ such that, for $x \in \mathit{Vars}$, $\&(\mathbf{ref}(x)) \equiv x$. To define such an expression in the language $\mathcal{L}$ requires an extension of the set $\mathcal{E}$ to include the application of functions from names to values. Such an extension is straightforward: a set of identifiers for these functions is defined as a subset of $\mathcal{F}$.

$$\mathcal{F}_{(\mathit{Names} \rightarrow \mathit{Values})} \subset \mathcal{F}$$

An interpretation function $\mathcal{I}_f{}'$ is defined on this subset, with type:

$$\mathcal{F}_{(\mathit{Names} \rightarrow \mathit{Values})} \rightarrow (\mathit{Names} \times \ldots \times \mathit{Names}) \rightarrow \mathit{Values}$$

The set of expressions $\mathcal{E}$ is extended to include the application of the functions to name expressions.

$$\frac{x \in \mathcal{E}_n \quad f \in \mathcal{F}_{(\mathit{Names} \rightarrow \mathit{Values})}}{f(x) \in \mathcal{E}}$$

The interpretation of expressions is also extended with the application of the interpretation $\mathcal{I}_f{}'$.

$$\mathcal{I}_e(f(x))(s) = \mathcal{I}_f{}'(f)(\mathcal{I}_n(x)(s)) \quad \text{if } f \in \mathcal{F}_{(\mathit{Names} \rightarrow \mathit{Values})}$$

The C operator $\&$ can then be defined, for $x \in \mathit{Names}$, $\mathcal{I}_f{}'(\&)(x) = \epsilon\{v : \mathit{Values} \mid x = \mathit{name}(v)\}$.

The control constructs of a high-level language are essentially those of a structured language together with a, possibly restricted, jump command. The transformations to construct an abstraction can, in general, be applied to a high-level program without modification. The iteration command of a high-level language forms a loop in the program and will be treated correctly by the transformation T_2. High-level languages include procedures and functions and these may complicate the abstraction of a program. Although it is likely that a region of a high-level program which included procedure calls could be transformed, this may require an extension to the methods which have been considered. A technique which might form the basis of such transformations is that of *procedure in-lining* in which the body of a procedure is substituted for the a call to the procedure.

Alternatives to the transformations T_1 and T_2 can be defined for particular processor languages or for sequence of instructions which commonly appear in an object program. For example, transformations can be specialised for the object code produced by a particular compiler.

These would use the translation rules of the compiler to re-construct the flow of control through the original program and to match the instructions of the object code with the commands of the source program. Where it is known that a particular compiler was used, the rules used for data refinement can also be used to simplify the expressions which occur in the object program.

Only sequential programs have been considered and therefore only the programs of the unprivileged mode of a processor. The definition of sequential composition depends on the fact that assignments are made to program variables by one command at a time. The privileged mode of a processor has a parallel execution model and a privileged mode program can allow two instructions to simultaneously assign values to the same variable. The order in which the assignments are actually made may be arbitrary and the value assigned to the variable will be unknown. While it is possible to model such a program, by simple extensions to $\mathcal{L}$, it is not generally possible to construct an abstraction of the program by the sequential composition of commands.

Abstractions of a parallel program can be constructed by partitioning the program into sub-programs which are executed sequentially and have exclusive access to a subset of the program variables. This models the parallel program as a set of sequential processes, similar to those described by Hoare (1985). A sub-program p exchanges data with other sub-programs at the beginning and end of the execution of p, by reading and writing variables, and each sub-program executes independently of other sub-programs. The parallel program must be verified by establishing liveness and safety properties for each sub-program. The liveness properties are those required by the program specification but the safety properties must ensure non-interference between sub-programs, by establishing that each sub-program writes only to the variables of that sub-program. The abstraction of the parallel program can then be constructed by constructing abstractions of each of the sequential sub-programs.

8.6 Summary of Results

This thesis describes a method for simplifying the verification of an object program by constructing an abstraction of the program. The techniques are based on the definition of processor instructions in terms of an abstract language. Arbitrary commands of this language can be combined by sequential composition. This results in a single command which is an abstraction of the original commands. The aliasing problem between program variables was solved by an extension to the assignment command. Abstractions of a computed jump were constructed by the use of conditional commands to determine the target of the jump. The method for abstracting from commands was extended to programs by defining transformations on the flow-graph of a program and the transformations were shown to construct an abstraction of the program. The transformations can be applied to the programs of a number of processor languages and the verification of an object program may use the standard methods for program verification. The methods for abstracting commands and programs are based on the syntax of the commands only. This suggests that the techniques may be efficiently mechanised as an automated tool for program verification.

Bibliography

Abadi, M. and Lamport, L. (1991). The existence of refinement mappings. *Theoretical Computer Science 82*(2), 253–284.

Aho, A., Sethi, R. and Ullman, J. (1986). *Compilers. Principles, Techniques and Tools*. Addison-Wesley.

Apt, K. R. (1981, October). Ten years of Hoare's logic: A survey-part I. *ACM Transactions on Programming Languages and Systems 3*(4), 431–483.

Back, R. J. R., Martin, A. J. and Sere, K. (1994). Specification of a microprocessor. Technical Report 148, Åbo Akademi University.

Back, R. J. R. and von Wright, J. (1989). Refinement calculus, part I: Sequential nondeterministic programs. In de Bakker, J. W., de Roever, W. P. and Rozenburg, G. (Eds.), *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness*, Volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Giménez, E., Herbelin, H., Huet, G., Noz, C. M., Murthy, C., Parent, C., Paulin, C., Saïbi, A. and Werner, B. (1997, August). The Coq proof assistant reference manual - version v6.1. Technical Report 0203, INRIA.

Barringer, H., Cheng, J. H. and Jones, C. B. (1984). A logic covering undefinedness in program proofs. *Acta Informatica 21*(3), 251 – 269.

Bevier, W. R., Warren A. Hunt, J., Moore, J. S. and Young, W. D. (1989, April). An approach to systems verification. Technical Report 41, Computational Logic Inc.

Bledsoe, W. W. (1974, December). The Sup-Inf Method in Presburger Arithmetic. Technical Report Memo ATP-18, Math. Dept, University of Texas at Austin, Austin,Texas.

Boulton, R. J. (1994). *Efficiency in a Fully-Expansive Theorem Prover*. Ph. D. thesis, University of Cambridge Computer Laboratory.

Bowen, J. (Ed.) (1994). *Towards Verified Systems*, Volume 2 of *Real-Time Safety Critical Systems*. Elsevier Science.

Bowen, J. and He Jifeng (1994). Specification, verification and prototyping of an optimized compiler. *Formal Aspects of Computing 6*(6), 643–658.

Boyer, R. S. and Moore, J. S. (1979). *A Computational Logic*. Academic Press.

Boyer, R. S. and Moore, J. S. (1997). Mechanized formal reasoning about programs and computing machines. In Veroff, R. (Ed.), *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, Chapter 4, pp. 147–176. MIT Press.

Boyer, R. S. and Yuan Yu (1996, January). Automated proof of object code for a widely used microprocessor. *Journal of the ACM 43*(1), 166–192.

Breuer, P. T. and Bowen, J. P. (1994, January). Decompilation: The enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems 11*(1), 147–167.

Bundy, A. (1983). *The Computer Modelling of Mathematical Reasoning*. Academic Press.

Burstall, R. M. (1974). Program proving as hand simulation with a little induction. In *Information Processing*, Volume 74, pp. 308–312. North-Holland.

Camilleri, J. and Melham, T. (1992). Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, Cambridge University.

Cartwright, R. and Oppen, D. (1981). The logic of aliasing. *Acta Informatica 15*, 365 – 384.

Chin-Tsun Chou (1994, September). A formal theory of undirected graphs in higher-order logic. See Melham & Camilleri (1994), pp. 144–157.

Church, A. (1940). A formulation of a simple theory of types. *Journal of Symbolic Logic 5*, 56–68.

Clarke, E. M., Emerson, E. A. and Sistla, A. P. (1986, April). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*(2), 244–263.

Clarke, E. M., Grumberg, O. and Long, D. E. (1994, September). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16*(5), 1512–1542.

Clarke, L. A. and Richardson, D. J. (1981). Symbolic evaluation methods for program analysis. See Muchnick & Jones (1981), Chapter 9, pp. 264–302.

Clint, M. and Hoare, C. A. R. (1972). Program proving: Jumps and functions. *Acta Informatica 1*, 214–224.

Colby, C. (1996). *Semantics-based Program Analysis via Symbolic Composition of Transfer Relations*. Ph. D. thesis, School of Computer Science, Carnegie Mellon University.

Cousot, P. (1981). Semantic foundations of program analysis. See Muchnick & Jones (1981), Chapter 10, pp. 303–342.

Cousot, P. (1990). Methods and logics for proving programs. In van Leeuwen, J. (Ed.), *Formal Models and Semantics*, Volume B of *Handbook of Theoretical Computer Science*, Chapter 15, pp. 841–994. Elsevier.

Cousot, P. and Cousot, R. (1987). Sometime $=$ always $+$ recursion $\equiv$ always - on the equivalence of the intermittent and invariant assertions methods for proving inevitability properties of programs. *Acta Informatica 24*, 1–31.

Curzon, P. (1992). A programming logic for a verified structured assembly language. In Voronkov, A. (Ed.), *Logic Programming and Automated Reasoning*, Volume 624 of *Lecture Notes in Computer Science*. Springer-Verlag.

Cyrluk, D., Möller, O. and Rueß, H. (1997). An efficient decision procedure for the theory of fixed-sized bit-vectors. In Grumberg, O. (Ed.), *Computer Aided Verification. 9th International Conference (CAV97). Haifa, Israel, June 22-25, 1997: Proceedings*, Volume 1254 of *Lecture Notes in Computer Science LNCS*, Berlin - Heidelberg - New York, pp. 60–71. Springer.

de Bakker, J. W. (1980). *Mathematical Theory of Program Correctness*. Prentice-Hall.

de Bruin, A. (1981). Goto statements: Semantics and deduction systems. *Acta Informatica 15*, 384 – 424.

Diefendorff, K. (1994, June). History of the PowerPC architecture. *Communications of the ACM 37*(6), 28–33.

Digital Equipment Corporation (1996, October). *Alpha Architecture Handbook*. Digital Equipment Corporation.

Dijkstra, E. (1976). *A Discipline of Programming*. Prentice-Hall.

Duffy, D. A. (1991). *Principles of Automated Theorem Proving*. John Wiley & Sons.

Fidge, C. J. (1997). Modelling program compilation in the refinement calculus. In *BCS-FACS Northern Formal Methods Workshop*.

Floyd, R. W. (1967). Assigning meanings to programs. In Schwartz, J. T. (Ed.), *Mathematical Aspects of Computer Science*, Volume 19 of *Symposia in Applied Mathematics*, pp. 19–32. Providence, RI: American Mathematical Society.

Ford, W. and Topp, W. (1988). *The MC68000: Assembly Language and Systems Programming*. D. C. Heath.

Francez, N. (1992). *Program Verification*. Addison Wesley.

Gordon, M. J. C. (1988). Mechanizing programming logics in higher order logic. Technical Report 145, Computer Laboratory, Cambridge University.

Gordon, M. J. C. (1994a). A mechanized Hoare logic of state transition. See Bowen (1994), Chapter 1, pp. 1–17.

Gordon, M. J. C. (1994b). State transition assertions: A case study. See Bowen (1994), Chapter 5, pp. 93–113.

Gordon, M. J. C. and Melham, T. F. (1993). *Introduction to HOL*. Cambridge University Press.

Gries, D. (1981). *The Science of Programming*. Springer-Verlag.

Grundy, J. (1993). *A Method of Program Refinement*. Ph. D. thesis, University of Cambridge.

Hayes, J. P. (1988). *Computer Architecture and Organization* (Second ed.). McGraw-Hill.

Hecht, M. (1977). *Flow Analysis of Computer Programs*. Elsevier.

Hennessy, J. L. and Patterson, D. A. (1990). *Computer Architecture: A Quantative Approach*. Morgan Kaufman.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM 12*, 576–580.

Hoare, C. A. R. (1972). Proofs of correctness of data representation. *Acta Informatica 1*(4), 271–281.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Hoare, C. A. R., Hayes, I. J., He Jifeng, Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sørensen, I. H., Spivey, J. M. and Sufrin, B. A. (1987, August). Laws of programming. *Communications of the ACM 30*(1), 672–686.

Hoare, C. A. R., He Jifeng and Sampaio, A. (1993). Normal form approach to compiler design. *Acta Informatica 30*(8), 701–739.

Jacobs, B., van den Berg, J., Huisman, M., van Berkum, M., Hensel, U. and Tews, H. (1998). Reasoning about java clases. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*. ACM SIGPLAN.

Jifeng He (1983). General predicate transformer and the semantics of a programming language with Go To statement. *Acta Informatica 20*, 35–57.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*. Prentice-Hall.

King, J. (1976, July). Symbolic execution and program testing. *Communications of the ACM 19*(7), 385–394.

King, J. C. (1971, November). Proving programs to be correct. *IEEE Transactions on Computers C-20*(11), 1331–1336.

Kleene, S. C. (1952). *Introduction to Meta-Mathematics*. North-Holland.

Lamport, L. (1994, May). The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*(3), 872–923.

Levy, A. (1979). *Basic Set Theory*. Springer-Verlag.

Loeckx, J. and Sieber, K. (1987). *The Foundations of Program Verification* (Second ed.). Wiley-Teubner.

Manna, Z. (1974). *Mathematical Theory of Computation*. McGraw-Hill.

Manna, Z. and Pnueli, A. (1981). Verification of temporal programs: The temporal framework. In Boyer, R. S. and Moore, J. S. (Eds.), *The Correctness Problem in Computer Science*, Chapter 5, pp. 215–275. Academic Press.

Manna, Z. and Pnueli, A. (1991). *The Temporal Logic of Reactive and Concurrent Systems. Specification*, Volume 1. Springer-Verlag.

Manna, Z. and Waldinger, R. (1981). Problematic features of programming languages: A situational-calculus approach. *Acta Informatica 16*, 371 – 426.

McCarthy, J. and Painter, J. (1967). Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, Volume 19 of *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society.

Melham, T. F. (1993). *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science. Cambridge University.

Melham, T. F. and Camilleri, J. (Eds.) (1994, September). *Higher Order Logic Theorem Proving and its Applications: 7th International Workshop*, Volume 859 of *Lecture Notes in Computer Science*. Springer-Verlag.

Milner, R. (1984). The use of machines in rigorous proof. *Phil. Trans. R. Soc. London, A 312*, 411–422.

Moore, J. S. (1994). Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning 6*(1), 33–45.

Morgan, C. (1990). *Programming from Specifications*. Prentice-Hall International.

Morris, J. M. (1987). A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming 9*, 287–306.

Morris, J. M. (1989). Laws of data refinement. *Acta Informatica 26*, 287–308.

Motorola (1986). *M68000 8-/16-/32-Bit Microprocessors, Programmer Reference Manual* (5 ed.). Prentice-Hall.

Motorola Inc. and IBM Corp. (1997). *PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors*. Motorola Inc. and IBM Corp. Revision 1.1.

Muchnick, S. and Jones, N. (1981). *Program Flow Analysis*. Prentice-Hall.

Müller-Olm, M. (1995, August). Structuring code generator correctness proofs by stepwise abstracting the machine language's semantics. Technical report, University of Kiel.

Necula, G. C. and Lee, P. (1996, September). Proof-carrying code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University.

Owre, S., Rushby, J. M. and Shankar, N. (1993). PVS: A Prototype Verification System. Technical Report SRI-CSL-93-04, SRI.

Paulson, L. C. (1985). Verifying the unification algorithm in LCF. *Science of Computer Programming 5*(2), 143–169.

Paulson, L. C. (1994a). *Isabelle: A Generic Theorem Prover*, Volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag.

Paulson, L. C. (1994b). Set theory for verification: I. from foundations to functions. Technical report, University of Cambridge.

Paulson, L. C. (1995a). Introduction to Isabelle. Distributed with the Isabelle system.

Paulson, L. C. (1995b). Set theory for verification: II. induction and recursion. Technical report, University of Cambridge.

Pavey, D. J. and Winsborrow, L. A. (1993). Demonstrating equivalence of source code and PROM contents. *The Computer Journal 36*(7), 654–667.

Polak, W. (1981). *Compiler Specification and Verification*, Volume 124 of *Lecture Notes in Computer Science*. Springer-Verlag.

Ramsey, N. and Fernández, M. F. (1997, May). Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems 19*(3), 492–524.

Sampaio, A. (1993). *An Algebraic Approach to Compiler Design*. Ph. D. thesis, University of Oxford.

Shostak, R. E. (1977, October). On the SUP-INF method for proving Presburger arithmetic. *Journal of the ACM 24*(4), 529–543.

Sites, R. L. (1992). Alpha AXP architecture. *Digital Technical Journal 4*(4), 1–17.

Sites, R. L. (1993, February). Alpha AXP architecture. *Communications of the ACM 36*(2), 33–44.

Skakkebaek, J. U. and Shankar, N. (1994). Towards a duration calculus proof assistant in PVS. Technical Report SRI-CSL-93-10, SRI.

Sreedhar, V. C., Gao, G. R. and Yong-Fong Lee (1996, November). Identifying loops in DJ graphs. *ACM Transactions on Programming Languages and Systems 8*(6), 649– 658.

Tarjan, R. E. (1981, July). A unified approach to path problems. *Journal of the ACM 28*(3), 577–593.

Tennent, R. D. (1991). *Semantics of Programming Languages*. Prentice-Hall.

Wakerly, J. F. (1989). *Microcomputer Architecture and Programming - The 68000 Family*. Wiley.

Weaver, D. L. and Germond, T. (Eds.) (1994). *The SPARC Architecture Manual, Version 9*. Prentice-Hall.

Wegbreit, B. (1977, July). Complexity of synthesizing inductive assertions. *Journal of the ACM 24*(3), 504–512.

Windley, P. J. (1994, September). Specifying instruction-set architectures in HOL: A primer. See Melham & Camilleri (1994).

Wolfe, M. (1991). Flow-graph anomalies: What's in a loop? Technical Report CS/E 92-012, Oregon Graduate Institute.

Wong, W. (1991, August). A simple graph theory and its application in railway signalling. In Archer, M., Joyce, J. J., Levitt, K. N. and Windley, P. J. (Eds.), *Proceedings of the 1991 International Workshop on HOL Theorem Proving System and its Applications*, Davis, California, USA, pp. 395–401. IEEE Computer Society Press.

Young, W. D. (1989, January). A mechanically verified code generator. Technical report, Computational Logic Inc.

Yuan Yu (1992). *Automated Proofs of Object Code for a Widely Used Microprocessor*. Ph. D. thesis, University of Texas at Austin.

Appendix A

Functions

The functions used in the examples of Chapter 5 are defined under the assumption that the values are the natural numbers, *Values* $= \mathbb{N}$. The general arithmetic functions, (equality, addition, subtraction, etc) are assumed to be as defined in Chapter 3. Bit-vector functions are the operators on bit-vectors of a given size *sz* and, in general, result in a bit-vector of size *sz*. A bit-vector is assumed to be represented as a natural number (in *Values*) and the size of the bit-vector is also a natural number. The bit-vector functions include signed and unsigned arithmetic operations and the equality and greater-than relations.

The only general function required in Chapter 5 which is not defined in Chapter 5 is the conditional expression **cond**$(b, x, y) \in \mathcal{E}$. This has the value of x if b is equivalent to **true** and is y otherwise.

Definition A.1 *Conditional expressions*

The conditional expression is a value function with name **cond** and arity 3.

$$\begin{aligned}&\mathbf{cond} \in \mathcal{F}_{\mathcal{V}}\\&\mathcal{I}_f(b, v_t, v_f) \stackrel{\text{def}}{=} \begin{cases} v_t & \text{if } \mathcal{B}(x) \\ v_f & \text{otherwise}\end{cases}\end{aligned}$$

□

A.1 Bit-Vector Functions

The bit-vector functions differ from the general functions in that the values are assumed to be the naturals, *Values* $= \mathbb{N}$, and in that the size of the bit-vector is an argument to the function. In general, the bit-vector functions are written $f(sz)(a_1, \ldots, a_n)$ where f is the function name, *sz* the size of the bit-vector and $a_1, \ldots, a_n$ are arguments. The function name f has arity $1 + n$, the notation is used only to distinguish particular arguments: $f(sz)(a_1, \ldots, a_n)$ is a synonym for the expression $f(sz, a_1, \ldots, a_n)$.

An accessor function for individual bits of a bit-vector, **bit** applied to a bit-vector a and index i results in the value of the ith bit of a. A constructor **mkBit** applied to an argument a has the result 0 iff a is equivalent to **false** and is 1 otherwise.

Definition A.2 *Bit operations*

The value of the ith bit of bit-vector a is obtained by applying the function **bit**.

$$\begin{aligned}&\mathbf{bit} \in \mathcal{F}_{\mathcal{V}}\\&\mathcal{I}_f(\mathbf{bit})(n)(a) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } (a \bmod 2^{n+1}) \geq 2^n \\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

The constructor **mkBit** applied to a value a results in 0 if a is equivalent to **false**.

$$\begin{aligned}&\mathbf{mkBit} \in \mathcal{F}_{\mathcal{V}}\\&\mathcal{I}_f(\mathbf{mkBit})(a) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \mathcal{I}_b(a) \\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

□

A.1.1 Bit-Vector Operations

The accessor function **B** applied to value i and a bit-vector a results in the ith byte of a. The accessor function **W** applied to value i and bit-vector a results in the ith word of a.

Definition A.3 *Accessors*

The accessor function for bytes is named **B**.

$$\begin{aligned}&\mathbf{B} \in \mathcal{F}_{\mathcal{V}}\\&\mathcal{I}_f(\mathbf{B})(i)(a) \stackrel{\text{def}}{=} (a \div 2^{i\times 8}) \bmod 2^8\end{aligned}$$

The accessor function for words is named **W**.

$$\begin{aligned}&\mathbf{W} \in \mathcal{F}_{\mathcal{V}}\\&\mathcal{I}_f(\mathbf{W})(i)(a) \stackrel{\text{def}}{=} (a \div 2^{i\times 16}) \bmod 2^{16}\end{aligned}$$

□

A.1.2 Arithmetic Operations

The arithmetic operators are addition, **plus**, negation, **neg**, subtraction, **minus**, unsigned multiplication, **mult**, and sign extension **ext**. All functions are applied to a bit-vector of a given size *sz* and, with the exception of sign extension, result in a bit-vector of size *sz*.

The function **plus** applied to size *sz* and arguments x, y is the result of $x + y$ which can represented in a bit-vector of size *sz*. The result of applying **neg** to size *sz* and argument x is the two's complement negation of x interpreted as a bit-vector of size *sz*. The function **minus** applied to size *sz* and expressions $x, y \in \mathcal{E}$ results in the two's complement representation, in a bit-vector of size *sz*, of $x - y$.

Definition A.4 *Addition and subtraction*

Addition **plus** and negation **neg** are functions in $\mathcal{F}_V$.

$$\begin{aligned}
&\textbf{plus}, \textbf{neg} \in \mathcal{F}_V \\
&\mathcal{I}_f(\textbf{plus})(sz)(x, y) \stackrel{\text{def}}{=} (x + y) \bmod 2^{sz} \\
&\mathcal{I}_f(\textbf{neg})(sz)(x) \stackrel{\text{def}}{=} (2^{sz-1} - x + 2^{sz-1}) \bmod 2^{sz}
\end{aligned}$$

Subtraction, **minus**, is an expression defined in terms of addition and negation.

$$\begin{aligned}
&\textbf{minus} : \textit{Values} \to (\mathcal{E} \times \mathcal{E}) \to \mathcal{E} \\
&\textbf{minus}(sz)(x, y) \stackrel{\text{def}}{=} \textbf{plus}(sz)(x, \textbf{neg}(sz)(y))
\end{aligned}$$

For $x, y \in \mathcal{E}$, $\textbf{plus}(sz)(x, y)$ is written $x +_{sz} y$ and $\textbf{minus}(sz)(x, y)$ is written $x -_{sz} y$. □

The result of function **mult** applied to arguments x, y of size *sz* is the representation of $x \times y$ in a bit-vector of size *sz*.

Definition A.5 *Unsigned multiplication*

Multiplication, **mult**, is a function in $\mathcal{F}_V$.

$$\begin{aligned}
&\textbf{mult} \in \mathcal{F}_V \\
&\mathcal{I}_f(\textbf{mult})(sz)(x, y) \stackrel{\text{def}}{=} (x \times y) \bmod 2^{sz}
\end{aligned}$$

For expressions $x, y \in \mathcal{E}$, $\textbf{mult}(sz)(x, y)$ will be written $x \times_{sz} y$. □

Sign extension, the function **ext**, is applied to a bit-vector a of size n and results in a bit-vector of size m.

Definition A.6 *Sign extension*

The function $\textbf{ext} \in \mathcal{F}_V$ sign extends a bit-vector.

$$\begin{aligned}
&\textbf{ext} \in \mathcal{F}_V \\
&\mathcal{I}_f(\textbf{ext})(n, m, a) \stackrel{\text{def}}{=} \begin{cases} ((2^{m-n} - 1) \times 2^n + a) \bmod 2^m, & \text{if } (a \bmod 2^n) \geq 2^{n-1} \\ a & \text{otherwise} \end{cases}
\end{aligned}$$

□

A.1.3 Shift and Rotate Operations

The shift and rotate operations on bit-vectors are defined in terms of functions for left and right shift operations described by Wakerly (1989). The shift left operation, **shiftl**, applied to bit-vector a, of size *sz*, and bit c shifts a left and sets the least significant bit of the result to the value of c. The shift right operation, **shiftr**, applied to bit-vector a, of size *sz*, and bit c shifts a right one position and sets the most significant bit of the result to the value of c.

Definition A.7 *Primitive shift functions*

The functions **shiftl** and **shiftr** are value functions of $\mathcal{F}_V$ with arity 3.

$$\begin{array}{l}\textbf{shiftl} \in \mathcal{F}_V \\ \mathcal{I}_f(\textbf{shiftl})(sz)(a, c) \stackrel{\text{def}}{=} ((a \times 2) + (c \bmod 2)) \bmod 2^{sz}\end{array}$$

$$\begin{array}{l}\textbf{shiftr} \in \mathcal{F}_V \\ \mathcal{I}_f(\textbf{shiftr})(sz)(a, c) \stackrel{\text{def}}{=} ((a \bmod 2^{sz-1}) \div 2) + ((c \bmod 2) \times (2^{sz-1}))\end{array}$$

□

Shift Operations

Processors provide both arithmetic and logical shifts. An arithmetic shift interprets the bit-vector as a two's complement number and an arithmetic shift to the left or right of bit-vector a is equivalent to the signed multiplication or division of a by 2. The arithmetic shift to the left and right are defined by **Ashiftl** and **Ashiftr** respectively.

Definition A.8 *Arithmetic shift functions*

Ashiftl applied to bit-vector a of size *sz* shifts a one position to the left. **Ashiftr** applied to bit-vector a of size *sz* shifts a one position to the right while preserving the sign of the bit-vector.

$$\begin{array}{l}\textbf{Ashiftl} : \mathit{Values} \to \mathcal{E} \to \mathcal{E} \\ \textbf{Ashiftl}(sz)(a) \stackrel{\text{def}}{=} \textbf{shiftl}(sz)(a, 0)\end{array}$$

$$\begin{array}{l}\textbf{Ashiftr} : \mathit{Values} \to \mathcal{E} \to \mathcal{E} \\ \textbf{Ashiftr}(sz)(a) \stackrel{\text{def}}{=} \textbf{shiftr}(sz)(a, \textbf{bit}(sz - 1, a))\end{array}$$

□

In a logical shift the bit-vector is interpreted as an unsigned number. A logical shift to the left is equivalent to an unsigned multiplication by 2 and a logical shift to the right is equivalent to an unsigned division by 2. The logical shift to the left and to the right are defined by **Lshiftl** and **Lshiftr** respectively.

Definition A.9 *Logical shift functions*

The logical shift to the left and right are defined by **Lshiftl** and **Lshiftr** respectively.

$$\mathbf{Lshiftl} : \mathit{Values} \to \mathcal{E} \to \mathcal{E}$$
$$\mathbf{Lshiftl}(\mathit{sz})(a) \stackrel{\text{def}}{=} \mathbf{shiftl}(\mathit{sz})(a, 0)$$

$$\mathbf{Lshiftr} : \mathit{Values} \to \mathcal{E} \to \mathcal{E}$$
$$\mathbf{Lshiftr}(\mathit{sz})(a) \stackrel{\text{def}}{=} \mathbf{shiftr}(\mathit{sz})(a, 0)$$

□

Rotate Operations

The shift operations on a bit-vector have the effect of setting the least or most significant bits to 0. A rotate operation sets this bit to the value of the most or least significant bit of the argument. For example, when a bit-vector is shifted to the left, the value of the most significant bit is lost. In a rotate to the left, this value is stored in the least significant bit.

Rotl applied to bit-vector a of size sz shifts a one position to the left and sets the least significant bit of the result to the most significant bit of the argument a. **Rotr** applied to bit-vector a of size sz shifts a one position to the right and sets the most significant bit of the result to the least significant bit of the argument a.

Definition A.10 *Rotate functions*

The rotation of a bit-vector by to the left and right is defined by **Rotl** and **Rotr** respectively.

$$\mathbf{Rotl} : \mathit{Values} \to \mathcal{E} \to \mathcal{E}$$
$$\mathbf{Rotl}(\mathit{sz})(a) \stackrel{\text{def}}{=} \mathbf{shiftl}(\mathit{sz})(a, \mathbf{bit}(\mathit{sz} - 1, a))$$

$$\mathbf{Rotr} : \mathit{Values} \to \mathcal{E} \to \mathcal{E}$$
$$\mathbf{Rotr}(\mathit{sz})(a) \stackrel{\text{def}}{=} \mathbf{shiftr}(\mathit{sz})(a, \mathbf{bit}(0, a))$$

□

A second form of rotation is on an *extended* bit-vector. A bit-vector is extended by the use of a variable x to store one or more additional bits. **Rotxl** applied to bit-vector a of size sz and extension variable x shifts a one position to the left and the least significant bit of the result is set to the value of x. **Rotxr** applied to bit-vector a of size sz and extension variable x shifts a one position to the right and the most significant bit of the result is set to the value of x.

Definition A.11 *Extended rotate functions*

The extended rotation to the left and right are defined by **Rotxl** and **Rotxr** respectively.

$$\textbf{Rotxl} : \textit{Values} \rightarrow \mathcal{E} \rightarrow \mathcal{E}$$
$$\textbf{Rotxl}(sz)(a, x) \stackrel{\text{def}}{=} \textbf{shiftl}(sz)(a, \textbf{mkBit}(x))$$

$$\textbf{Rotxr} : \textit{Values} \rightarrow \mathcal{E} \rightarrow \mathcal{E}$$
$$\textbf{Rotxr}(sz)(a, x) \stackrel{\text{def}}{=} \textbf{shiftr}(sz)(a, \textbf{mkBit}(x))$$

□

A.1.4 Memory Access

Functions to access the values stored in memory are defined in terms of name expression **ref**, as value expressions and as an assignment list. Assuming that each memory variable stores a byte, a long-word a is stored in memory by assigning each byte of a to one of four consecutive memory locations.

Definition A.12 Function **writel** applied to expressions a and e, constructs an assignment list in which the long word between a and $a +_{32} 3$ is assigned byte 3 to byte 0 of e.

$$\textbf{writel} : (\mathcal{E} \times \mathcal{E}) \rightarrow \textit{Alist}$$
$$\begin{aligned}\textbf{writel}(a, e) \stackrel{\text{def}}{=} \; & (\textbf{ref}(a), \textbf{B}(3)(e)) \cdot (\textbf{ref}(a +_{32} 1), \textbf{B}(2)(e)) \cdot \\ & (\textbf{ref}(a +_{32} 2), \textbf{B}(1)(e)) \cdot (\textbf{ref}(a +_{32} 3), \textbf{B}(0)(e)) \cdot \textit{nil}\end{aligned}$$

□

Appendix B

Processor Language Features

Processor languages include a number of features which must be modelled in the language $\mathcal{L}$ to enable an object program to be verified. As an example of such features, the method by which the Motorola 68000 interprets a value and the addressing modes of the M68000 are described in terms of expressions of $\mathcal{L}$.

B.1 Motorola 68000: Condition Codes

Data is interpreted by assigning a value to the condition codes (also called flags) of the M68000, the five least significant bits of the status register, **SR**. The condition codes are set by an instruction to reflect the result of an operation on data and the semantics of such an instruction will include an assignment to the status register. The expression assigned to **SR** can be calculated by applying the function **mkSR** to the five bits representing the values of the individual condition codes.

Definition B.1 The condition codes are set by a constructor **mkSR** applied to five arguments: x, the extend flag, n, the negative flag, z, the zero flag, v, the overflow and c, the carry flag, $\mathbf{mkSR}(x, n, z, v, c)$.

$$\mathbf{mkSR} \in \mathcal{F}_V$$
$$\mathcal{I}_f(\mathbf{mkSR})(x,n,z,v,c) \stackrel{\text{def}}{=} \begin{cases} (\mathbf{mkBit}(x) \times 2^4) + (\mathbf{mkBit}(n) \times 2^3) \\ +(\mathbf{mkBit}(z) \times 2^2) + (\mathbf{mkBit}(v) \times 2) + c \end{cases}$$

□

The method for calculating the individual condition codes and their used depends on the particular instruction. In general, instructions which perform similar operations have a similar interpretation and make similar use of the condition codes, e.g. the arithmetic instructions typically use the same interpretation of data. The condition codes are used by accessing the individual bits of the status register, **SR**.

Carry (C), Bit 0: Bit 0 is set by addition and subtraction operations to reflect the carry out of an addition operation. If the flag is set then the result of the operation was too large to represent in the size of bit-vector which was used. Since subtraction is defined in terms of addition, the interpretation of the carry flag is derived from that of addition. Other operations set the carry to 0.

The carry flag in an addition operation on operands of size *sz* with arguments x, y and result e, $e = x + y$, is set to 1 if $e \leq 2^{sz}$. For the M68000 processor, the flag is set if the most significant bit of both x and y are set or if either is set and the most significant bit of r is not. The most significant bit is determined from the size *sz* of the operations.

$$\mathbf{bit}(0)(\mathbf{SR}) = \begin{cases} \mathbf{bit}(sz -_a 1)(x) \textbf{ and } \mathbf{bit}(sz -_a 1)(y) \\ \textbf{ or not } \mathbf{bit}(sz -_a 1)(e) \textbf{ and } \mathbf{bit}(sz -_a 1)(x) \\ \textbf{ or not } \mathbf{bit}(sz -_a 1)(e) \textbf{ and } \mathbf{bit}(sz -_a 1)(y) \end{cases}$$

Overflow (V), Bit 1: In general, the overflow flag is set when the result of an operation is too large to be represented in the bit-vector size used. For addition, $e = x + y$, the overflow is set if the most significant bits of the arguments x and y differ from the most significant bits of the result e. The most significant bit is determined from the size *sz* of the operations.

$$\mathbf{bit}(1)(\mathbf{SR}) = \begin{cases} \mathbf{bit}(sz -_a 1)(x) =_a \mathbf{bit}(sz -_a 1)(y) \\ \textbf{ and not } (\mathbf{bit}(sz -_a 1)(e) =_a \mathbf{bit}(sz -_a 1)(x)) \end{cases}$$

Zero (Z), Bit 2: The zero flag is set when the result of an operation is equal to 0. The method used in the semantics of the M68000 to determine whether the result e is 0 is by the conjunction of the negated bits of the result. The size *sz* of the operation determines the number of bits which are compared.

$$\mathbf{bit}(2)(\mathbf{SR}) = \textbf{not } \mathbf{bit}(sz -_a 1)(e) \textbf{ and } \ldots \textbf{ and not } \mathbf{bit}(0)(e)$$

This is equivalent to a direct comparison with 0.

$$\mathbf{bit}(2)(\mathbf{SR}) = (e =_a 0)$$

Negative (N), Bit 3: The negative flag is set when the result of an operation, interpreted as a two's complement number, is less than 0. In the two's complement system, a number is negative if the most significant bit is set. Assume that the size of the operation is *sz* and that the result of the operation is e. The negative flag is assigned the value of bit $sz - 1$th of e.

$$\mathbf{bit}(3)(\mathbf{SR}) = \mathbf{bit}(sz -_a 1)(e)$$

Extend (X), Bit 4: In arithmetic operations this is set to the value of the carry flag. In other operations this is a general purpose flag. For example, the rotate operations used the extend flag as an extension to the bit-vector to be rotated.

B.1.1 Condition Code Calculation

Examples of common methods for calculating the condition codes are those of the data movement, `move`, addition, `add`, and comparison, `cmp`, instructions. The semantics of each instruction describes the operation to be performed and assigns a value to the status register, **SR**, reflecting the result of the operation. The instructions are written with the instruction size, *sz*, source argument, *src*, and destination argument, *dst*.

Data movement `move`.*sz* *src*, *dst*

The result of the operation is the value of the source argument, *src*, assigned to the destination argument *dst*. Condition code X is unaffected, N is set if the source operand is negative, Z is set if the source is 0 and V and C are cleared. Assume that the size of the operation is a byte, $sz = Byte$.

$$\begin{aligned} dst, \mathbf{SR} :=\,& \mathbf{Byte}(src),\\ & \mathbf{mkSR}(\mathbf{bit}(4)(\mathbf{SR}), \mathbf{bit}(sz -_a 1)(src), (src =_{sz} 0), \mathbf{false}, \mathbf{false}) \end{aligned}$$

Addition `add`.*sz* *src*, *dst*

The destination argument is assigned the value of $src + dst$. The destination must be a register or a memory variable. When the destination is an address register, the status register is unchanged. Assume that the destination is not an address register.

$$dst, \mathbf{SR} := e, \mathbf{mkSR}(C, N, Z, V, C)$$

$$\begin{aligned} \text{where } e = (dst +_{sz} src), N &= \mathbf{bit}(sz -_a 1)(e), Z = (e =_{sz} 0)\\ V =\ & \mathbf{bit}(sz -_a 1)(src) \textbf{ and } \mathbf{bit}(sz -_a 1)(dst) \textbf{ and not } \mathbf{bit}(isz -_a 1)(e)\\ & \textbf{or not } \mathbf{bit}(sz -_a 1)(src) \textbf{ and not } \mathbf{bit}(sz -_a 1)(dst) \textbf{ and } \mathbf{bit}(isz -_a 1)(e)\\ C =\ & \mathbf{bit}(sz -_a 1)(src) \textbf{ and } \mathbf{bit}(sz -_a 1)(dst)\\ & \textbf{or not } \mathbf{bit}(isz -_a 1)(e) \textbf{ and } \mathbf{bit}(sz -_a 1)(src)\\ & \textbf{or not } \mathbf{bit}(isz -_a 1)(e) \textbf{ and } \mathbf{bit}(sz -_a 1)(dst) \end{aligned}$$

Comparison `cmp`.*sz* *src*, *dst*

A comparison instruction performs a similar operation to a subtraction. The destination is subtracted from the source, $(dst -_{sz} src)$ and the condition flags are set according to the result. In the comparison instruction, only the status register is changed.

$$\mathbf{SR} := \mathbf{mkSR}(X, N, Z, V, C)$$

$$\begin{aligned}
\text{where } e = (\mathit{dst} -_{sz} \mathit{src}), X &= \textbf{bit}(4)(\textbf{SR}), N = \textbf{bit}(sz -_a 1)(e), Z = (e =_{sz} 0)\\
V = &\ \textbf{not bit}(sz -_a 1)(\mathit{src}) \textbf{ and bit}(sz -_a 1)(\mathit{dst}) \textbf{ and not bit}(isz -_a 1)(e)\\
&\ \textbf{or bit}(sz -_a 1)(\mathit{src}) \textbf{ and not bit}(sz -_a 1)(\mathit{dst}) \textbf{ and bit}(isz -_a 1)(e)\\
C = &\ \textbf{bit}(sz -_a 1)(\mathit{src}) \textbf{ and not bit}(sz -_a 1)(\mathit{dst})\\
&\ \textbf{or bit}(isz -_a 1)(e) \textbf{ and bit}(sz -_a 1)(\mathit{src})\\
&\ \textbf{or bit}(isz -_a 1)(e) \textbf{ and not bit}(sz -_a 1)(\mathit{dst})
\end{aligned}$$

B.1.2 Addressing Modes of the M68000

Addressing modes determine how an operand is obtained from arguments to a processor instruction. In general, an instruction argument is used to determine the variable name, a memory address or register, in which the operand is stored. The description of these addressing modes is by name expressions accessing individual bytes of memory. The exception is the *immediate* addressing mode in which the instruction argument is the operand and is described by a value expression.

Immediate addressing This mode applies only when the argument is a source and the argument is the operand. The instruction argument is written with a # prefix and is interpreted as a value. For example, in the instruction `inst` $\#1, d$ the operand is the value 1.

Absolute addressing The instruction argument is a value identifying the memory location in which the operand is stored. The instruction argument $x : sz$ denotes the address in memory x. The size of the bit-vector x is sz and is either a word or a long-word. When the bit-vector is a word, it is sign extended to a long-word.

$$\textbf{ref}(\textbf{ext}(\mathit{Word}, \mathit{Long}, x))$$

When the bit-vector is a long-word, it is used directly, $\textbf{ref}(x)$.

Data register direct The instruction argument is a data register. For instruction `inst` $\textbf{Dn}, \mathit{dst}$ the source is the value of **Dn**, $\mathit{dst} := f(\textbf{Dn})$. For instruction `inst` $\mathit{src}, \textbf{Dn}$ the destination is the register **Dn**, $\textbf{Dn} := f(\textbf{Dn})$.

Address register direct The instruction argument and the operand are an address register. For instruction `inst` $\textbf{An}, \mathit{dst}$ the source is the value of **An**, $\mathit{dst} := f(\textbf{An})$. In instruction `inst` $\mathit{src}, \textbf{An}$ the destination is the register **An**, $\textbf{An} := f(\mathit{src})$.

Address register indirect The operand is stored in memory at an address stored in an address register which is the instruction argument. For instruction `inst` $\textbf{An}@, \mathit{dst}$ the source is the value stored in memory at the address **An**, $\mathit{dst} := f(\textbf{ref}(\textbf{An}))$. In instruction `inst` $\mathit{src}, \textbf{An}@$ the destination is the memory location at address **An**, $\textbf{ref}(\textbf{An}) := f(\mathit{src})$

Address register with post-increment The operand is in memory at an address stored in an address register. After the operand has been used and updated, the address register is set to the address of the next operand in memory. The operand may be of size *Byte*, *Word* or *Long* and the change to the address register is by the addition of 1, 2 or 4. If the address register is the stack pointer (**A7**), then the increment is always a multiple of 2. With an operand is of size *Byte*, the increment is also by 2 bytes.

For instruction `inst`.*sz* *src*, $\mathbf{An}@+$ the destination is the memory location at address **An**, $\mathbf{ref}(\mathbf{An}) := f(\mathbf{ref}(\mathbf{An}))$. The change to the address register is of 1, 2, or 4 bytes: for a long-word, $\mathbf{An} := \mathbf{An} +_{32} 4$. The addition operates on a long-word since memory addressing is by long-words. The model of the instruction is the combination of the operation and the increment to the register.

$$\mathbf{ref}(\mathbf{An}), \mathbf{An} := f(\mathit{src}), \mathbf{An} +_{32} 4$$

When used with the stack pointer (**A7**), this mode implements a pop from a stack. The instruction `inst.b` $\mathbf{A7}@+$, *dst* uses the byte at the top of the stack and increments the stack pointer to the next word. The memory access of the M68000 must be aligned on a word boundary and the stack pointer is incremented by 2. The instruction is modelled as with the general case.

$$\mathit{dst}, \mathbf{An} := f(\mathbf{ref}(\mathbf{An})), \mathbf{An} +_{32} 2$$

Address register indirect with pre-decrement The instruction argument is an address register and the operand is in memory at the address immediately below that stored in the argument. The address register is decremented by 1, 2 or 4 depending on the operand size and the operand is the location identified by the new value of the address register. As with the address register indirect with pre-increment mode, the stack pointer **A7** is always decremented by a multiple of two.

For instruction `inst`.*sz* *src*, $\mathbf{An}@-$, the destination is calculated from the size of the operand. When the size is a byte, the operand is obtained by $\mathbf{ref}(\mathbf{An} -_{32} 2))$. The change to the address register is $\mathbf{An} -_{32} 2$. The two are combined to give the model of the instruction.

$$\mathbf{ref}(\mathbf{An} -_{32} 1), \mathbf{An} := f(\mathit{src}), \mathbf{An} -_{32} 1$$

The stack pointer is always aligned with a word boundary. For $n = 7$ the address register is decremented by two bytes.

$$\mathbf{ref}(\mathbf{An} -_{32} 2), \mathbf{An} := f(\mathit{src}), \mathbf{An} -_{32} 2)$$

Address register indirect with displacement The argument is an address register with a displacement of size *Word*. The operand is stored at the address calculated from the contents of the address register together with the sign-extended displacement.

In instruction `inst.`sz *src*, **An**$@(x)$, the destination argument is obtained by the expression **ref**(**An** $+_{32}$ **ext**(*Word*, *Long*, x)).

$$\mathbf{ref}(\mathbf{An} +_{32} \mathbf{ext}(\mathit{Word}, \mathit{Long}, x)) := f(\mathit{src})$$

Address register indirect with index The argument to the instruction is an address register, an index register (an address or data register), an integer displacement, an integer size and an integer scale. The address register, index register and displacement are optional.

The argument is written **An**$@(i, \mathbf{r} : sz : sc)$ where **An** is an address register, i is the integer displacement, sz the size of the data, sc the scale and **r** is a data or address register. The location in memory of the operand is calculated by the addition of i, the value stored at the location stored in **An** and the value stored of **r** multiplied by the size and scale.

$$\mathbf{readl}(\mathbf{An} +_{32} \mathbf{ext}(sz, \mathit{Long}, i) +_{32} (\mathbf{ext}(sz, \mathit{Long}, \mathbf{r}) \times_{32} sc))$$

The reference manual (Motorola, 1986) defines different forms of this addressing mode. When the displacement can be represented in a byte, the mode is called *8-bit displacement*; when the displacement is a word or long it is called a *base displacement*. These modes are also defined with the program counter **PC** replacing the address register and are referred to as *program counter indirect with index*.

Memory indirect post-indexed The operand is stored in a memory location whose address x is also stored in memory. The location at which address x is stored is determined from the argument which is written **An**$@(d_1)@(d_2, \mathbf{r} : sz : sc, d_2)$. **An** is an address register, d_1 and d_2 are integer displacements, **r** is an index register (an address or data register), sz is the operation size and sc is an integer scale.

The location of the address x is calculated by the addition of **An** and d_1. The address of the operand is calculated by addition of the address x, the scaled index register, $\mathbf{r} \times sc$, and the integer d_2. The integers d_1 and d_2 and the index register are sign extended to a long-word.

$$\begin{aligned}&\mathbf{ref}(\mathbf{readl}(\mathbf{An} +_{32} \mathbf{ext}(sz, \mathit{Long}, d_1))\\ &\quad +_{32} (\mathbf{ext}(sz, \mathit{Long}, \mathbf{r}) \times_{32} sc)\\ &\quad +_{32} \mathbf{ext}(sz, \mathit{Long}, d_2)))\end{aligned}$$

Memory indirect pre-indexed The operand is stored in a memory location whose address x is also stored in memory. The location at which address x is stored is determined from the argument which is written **An**$@(d_1, \mathbf{r} : sz : sc)@(d_2)$. **An** is an address register, d_1 and d_2 are integer displacements, **r** is an index register (an address or data register), sz is the operation size and sc is an integer scale.

The location of the address x is calculated by the addition of the address register **An**, the scaled index register, $\mathbf{r} \times sc$, and the integer d_1. The address of the operand is calculated by the addition of the address x and the integer d_2. The integers d_1 and d_2 and the index register are sign extended to a long-word.

$$\begin{array}{l}\mathbf{ref}(\mathbf{readl}(\mathbf{An} +_{32} \mathbf{ext}(sz, Long, d_1) +_{32} (\mathbf{ext}(sz, Long, \mathbf{r}) \times_{32} sc)) \\ \quad +_{32} \mathbf{ext}(sz, Long, d_2))\end{array}$$

Expressions defining memory access for word and long-words use similar calculation but must be defined in terms of functions such as **writel** and **readl**. For example, to obtain a long-word from instruction argument **A** and the address register indirect addressing mode, the expression used in $\mathbf{readl}(\mathbf{ref}(\mathbf{A}))$. To store a long-word x with argument **A** and using the address register indirect mode, an assignment list is constructed by $(\mathbf{A}, \mathbf{A} +_{32} 4) \cdot \mathbf{writel}(\mathbf{A}, x)$.

Appendix C

Proofs: Commands

This appendix contains proofs of the theorems and lemmas of Chapter 3, proofs for the substitution rules of Figure (3.3) and a definition for the predicate *correct?*. The order in which the proofs and definitions are presented is that of Chapter 3. The proofs for lemmas concerning the expressions are followed by the proofs for substitution rules. These are followed by the definition of *correct?* and the proofs for the sequential composition operator.

C.1 Expressions

C.1.1 Lemma (3.1)

Proof.

1. Immediate.
2. The definition of equivalence gives $\mathcal{I}_n(n_1)(s) = \mathcal{I}_n(n_2)(s)$. From the definition of $\mathcal{I}_e$, $\mathcal{I}_e(n_1)(s) = s(\mathcal{I}_n(n_1)(s))$ and $\mathcal{I}_e(n_2)(s) = s(\mathcal{I}_n(n_2)(s))$. By substitution, $s(\mathcal{I}_n(n_1)(s)) = s(\mathcal{I}_n(n_2)(s))$.
3. Note that if $\mathcal{I}_1 = \mathcal{I}_n$ (and $e_1, e_2 \in \mathcal{E}_n$) then it follows, as before, that $e_1 \equiv_{(\mathcal{I}_e,s)} e_2$.

   Case $\mathcal{I}_2 = \mathcal{I}_e$: By definition, the interpretation of the expressions is $\mathcal{I}_e(f(e_1)(s)) = \mathcal{I}_f(f)(\mathcal{I}_e(e_1)(s))$ and $\mathcal{I}_e(f(e_2)(s)) = \mathcal{I}_f(f)(\mathcal{I}_e(e_2)(s))$. The proof follows immediately from $e_1 \equiv_{(\mathcal{I}_e,s)} e_2$.

   Case $\mathcal{I}_2 = \mathcal{I}_n$ (with $f \in \mathcal{F}_n$): By definition, the interpretation of the expressions is $\mathcal{I}_e(f(e_1)(s)) = \mathcal{I}_f(f)(\mathcal{I}_e(e_1)(s))$ and $\mathcal{I}_e(f(e_2)(s)) = \mathcal{I}_f(f)(\mathcal{I}_e(e_2)(s))$. As before the proof follows immediately from $e_1 \equiv_{(\mathcal{I}_e,s)} e_2$.

$\square$

C.2 Substitution

Substitution in Assignment Lists

Lemma C.1 *For any name expression $x \in \mathcal{E}_n$, assignment list al and state s,*

$$\mathcal{I}_n(x)(s) \in_s al \Leftrightarrow x \in_s al$$

Proof. By induction on al:

Case $al = nil$: By definition, $\mathcal{I}_n(x)(s) \in_s nil = false = x \in_s nil$.

Case $al = (t, r) \cdot bl$ with hypothesis $\mathcal{I}_n(x)(s) \in_s bl \Leftrightarrow x \in_s bl$: From $\mathcal{I}_n(x)(s) \in$ *Names* and the definition of $\mathcal{I}_n$, $\mathcal{I}_n(\mathcal{I}_n(x)(s))(s) = \mathcal{I}_n(x)(s)$ and $\mathcal{I}_n(x)(s) \equiv_s t$ iff $x \equiv_s t$. If $x \equiv_s t$ then the equivalence of membership follows immediately. If $x \not\equiv_s t$ then the proof follows from the inductive hypothesis.

Case $al = bl \oplus cl$ with hypothesis $\mathcal{I}_n(x)(s) \in_s bl \Leftrightarrow x \in_s bl$ and $\mathcal{I}_n(x)(s) \in_s cl \Leftrightarrow x \in_s cl$: The proof of equivalence follows immediately from the definition of membership and the inductive hypothesis. □

Lemma C.2 *Properties of find*

For name expression x, assignment list al and state s,

$$\frac{x \notin_s al}{\mathit{find}(x,\ al)(s) = x}$$

Proof. By induction on al.

Case $al = nil$: $\mathit{find}(x, al)(s) = x$ by definition.

Case $al = (x_1, e_1) \cdot bl$ and $x \notin_s bl \Rightarrow \mathit{find}(x, bl)(s) = x$: if $x_1 \equiv_s x_2$ then $x \in_s al$ which is a contradiction. Since $x_1 \not\equiv_s x$, $\mathit{find}(x, al)(s) = \mathit{find}(x, bl, s) = x$ since if $x \in_s bl$ then $x \in_s al$.

Case $al = bl \oplus cl$ with $x \notin_s bl \Rightarrow \mathit{find}(x, bl)(s) = x$ and $x \notin_s cl \Rightarrow \mathit{find}(x, cl)(s) = x$: Since $x \notin_s cl$, $\mathit{find}(x, al)(s) = \mathit{find}(x, bl)(s)$. $x \notin_s bl$ and $\mathit{find}(x, al)(s) = x$. □

C.2.1 Lemma (3.2)

Proof.

Case $x \in$ *Names*: By definition, $x \lhd al = x$, and $\mathcal{I}_n(x)(\mathit{update}(al, s)) = x = \mathcal{I}_n(x)(s)$.

Case $x = f(a_1, \ldots, a_n)$ for $f \in \mathcal{F}n$: By definition of substitution, $f(a_1, \ldots, a_n) \lhd al = f(a_1 \lhd al, \ldots, a_n \lhd al)$. From the definition of $\mathcal{I}_n$, $\mathcal{I}_n(f(a_1 \lhd al, \ldots, a_n \lhd al)(s))$ is $\mathcal{I}_f(f)(\mathcal{I}_e(a_1 \lhd al)(s), \ldots, \mathcal{I}_e(a_n \lhd al)(s))$. For $1 \leq i \leq n$, $\mathcal{I}_e(a_i \lhd al)(s) = \mathcal{I}_e(a_i)(\mathit{update}(al, s))$. This leads to $\mathcal{I}_f(f)(\mathcal{I}_e(a_1)(\mathit{update}(al, s)), \ldots, \mathcal{I}_e(a_n)(\mathit{update}(al, s)))$ which is equivalent to the interpretation in the updated state, $\mathcal{I}_n(f(a_1, \ldots, a_n))(\mathit{update}(al, s))$. □

C.2.2 Theorem (3.1)

Membership of an assignment list is based on the equivalence of name expressions and membership in state s updated with assignment list al of a name expression x is equivalent to membership in s of $x \lhd al$.

Lemma C.3 *For name expression $x \in \mathcal{E}_n$, assignment lists $al, bl \in Alist$ and state s,*

$$x \in_{update(bl,s)} al = (x \lhd bl) \in_s al \lhd bl$$

Proof. By induction on al.

Case $al = nil$: By definition, both sides of the equation are *false*.

Case $al = (x_1, v_1) \cdot al'$ and the property is *true* for al': By definition $((x_1, v_1) \cdot al') \lhd bl = (x_1 \lhd bl, v_1 \lhd bl) \cdot (al' \lhd bl)$. From the definition of membership and substitution in a name expression, $x \equiv_{update(bl,s)} x_1$ iff $x \lhd bl \equiv_s x_1 \lhd bl$. The remainder of the proof is straightforward from the definition of membership and the inductive hypothesis.

Case $al = (cl_1 \oplus cl_2)$ and the property is *true* for both cl_1 and cl_2: The proof is straightforward from the definition of membership. □

When the expression e is a name expression, searching for the value associated with e in assignment list al in the state s updated with bl is equivalent to substituting bl in both e and al and performing the search in the state s.

Lemma C.4 *For name expression $n \in \mathcal{E}_n$, assignment lists $al, bl \in Alist$, and states $s, s' \in State$,*

$$\frac{s' = update(bl, s) \quad n \in_{s'} al}{\mathcal{I}_e(find(n, al)(s'))(s') = \mathcal{I}_e(find(n \lhd bl, al \lhd bl)(s))(s)}$$

Proof. By induction on al,

Case $al = nil$. By definition $n \in_{s'} nil = false$ contradicting the assumption.

Case $al = (x, v) \cdot al'$ with the property *true* for al'. By Lemma (C.3), $n \equiv_{update(bl,s)} x$ iff $n \lhd bl \equiv_s x \lhd bl$.

Assume $n \equiv_{update(bl,s)} x$, it follows that $n \lhd bl \equiv_s x \lhd bl$. By definition, $\mathcal{I}_e(find(n, al)(s'))(s') = \mathcal{I}_e(v)(s')$ and $\mathcal{I}_e(find(n \lhd bl, al \lhd bl)(s))(s) = \mathcal{I}_e(v \lhd bl)(s)$. The proof that $\mathcal{I}_e(v)(s') = \mathcal{I}_e(v \lhd bl)(s)$ is immediate from the definition of substitution and from $s' = update(bl, s)$.

Assume $n \not\equiv_{update(bl,s)} x$. By definition, $\mathcal{I}_e(find(n, al)(s'))(s') = \mathcal{I}_e(find(n, al')(s'))(s')$ and, from the inductive hypothesis, this is $\mathcal{I}_e(find(n \lhd bl, al' \lhd bl)(s))(s)$. Also by definition and since $n \lhd bl \not\equiv_s x \lhd bl$, $\mathcal{I}_e(find(n \lhd bl, al \lhd bl)(s))(s) = \mathcal{I}_e(find(n \lhd bl, al' \lhd bl)(s))(s)$ completing the proof for this case.

Case $al = (cl_1 \oplus cl_2)$ with the property *true* for cl_1 and cl_2.

Assume $n \in_{s'} cl_1$. From Lemma (reflem:2.1.1) it follows that $n \lhd bl \in_s cl_1 \lhd bl$. By definition, $\mathcal{I}_e(\mathit{find}(n, al, s'))(s') = \mathcal{I}_e(\mathit{find}(n, cl_1, s'))(s')$ and the property follows from the inductive hypothesis. Assume $n \notin_{s'} cl_1$, it follows that $n \lhd bl \notin_s cl_1 \lhd bl$ and (from the assumption $n \in_{s'} al$) $n \in_{s'} cl_2$. Consequently, $n \lhd bl \in_s cl_2 \lhd bl$ and, by definition, the property to be established is $\mathcal{I}_e(\mathit{find}(n, cl_2, s'))(s') = \mathcal{I}_e(\mathit{find}(n, cl_2 \lhd bl, s))(s)$ which is immediate from the inductive hypothesis. □

Proof. *Theorem (3.1)* By induction on al and by extensionality with $x \in \mathit{Names}$.

Case $al = \mathit{nil}$. From the definitions, and since $\mathit{nil} \lhd bl = \mathit{nil}$, $\mathit{update}(bl, s)(x) = \mathit{update}(bl, s)(x)$ is straightforward

Case $al = (x_1, v_1) \cdot al'$ and the property is *true* for al': By definition, $\mathit{update}(al, \mathit{update}(bl, s))(x)$ is $\mathcal{I}_e(\mathit{find}(x, al)(\mathit{update}(bl, s)))(\mathit{update}(bl, s))$. Also, $\mathit{update}(bl \oplus (al \lhd bl), s)(x)$ is $\mathcal{I}_e(\mathit{find}(x, bl \oplus (al \lhd bl))(s))(s)$.

Assume $x \in_{\mathit{update}(bl,s)} al$. From Lemma (C.3), it follows that $x \in_s al \lhd bl$ is *true* ($x \lhd bl = x$ since $x \in \mathit{Names}$ and the substitution is for name expressions). This reduces the equation to $\mathcal{I}_e(\mathit{find}(x, al)(\mathit{update}(bl, s)))(\mathit{update}(bl, s)) = \mathcal{I}_e(\mathit{find}(x, (al \lhd bl))(s))(s)$.

Assume that $x \equiv_{\mathit{update}(bl,s)} x_1$. By definition this is *true* iff $x \lhd bl \equiv_s x_1 \lhd bl$. The result of $\mathcal{I}_e(\mathit{find}(x, al)(\mathit{update}(bl, s)))(\mathit{update}(bl, s)$ will therefore be $\mathcal{I}_e(v_1)(\mathit{update}(bl, s))$ and the result of $\mathcal{I}_e(\mathit{find}(x, (al \lhd bl))(s)))(s)$ is $\mathcal{I}_e(v_1 \lhd bl)(s)$. The proof follows from the definition of substitution.

Assume that $x \not\equiv_{\mathit{update}(bl,s)} x_1$. That $x \in_{\mathit{update}(bl,s)} al'$ follows from the definition of membership and the proof is straightforward from the inductive hypothesis and the definitions.

Assume $x \notin_{\mathit{update}(bl,s)} al$. From Lemma (C.3), it follows that $x \notin_s al \lhd bl$. If x is assigned to by the assignment list it must be by bl. The equation reduces to $\mathit{update}(bl, s)(x) = \mathit{update}(bl, s)$ from the definitions.

Case $al = (cl_1 \oplus cl_2)$ with the property *true* for cl_1 and cl_2. The proof is similar to the previous case.

By definition, $\mathit{update}(al, \mathit{update}(bl, s))(x)$ is $\mathcal{I}_e(\mathit{find}(x, al)(\mathit{update}(bl, s)))(\mathit{update}(bl, s))$. Also by definition, $\mathit{update}(bl \oplus (al \lhd bl), s)(x)$ is $\mathcal{I}_e(\mathit{find}(x, bl \oplus (al \lhd bl))(s))(s)$.

Assume that $x \in_{\mathit{update}(bl,s)} al$. From Lemma (C.3), it follows that $x \in_s al \lhd bl$ (since $x \in \mathit{Names}$, $x \lhd bl = x$). Consequently, $\mathcal{I}_e(\mathit{find}(x, bl \oplus (al \lhd bl))(s))(s)$ is equivalent to $\mathcal{I}_e(\mathit{find}(x, al \lhd bl)(s))(s)$. From Lemma (C.4), this is equivalent to $\mathcal{I}_e(\mathit{find}(x, al)(\mathit{update}(bl, s)))(\mathit{update}(bl, s))$.

Assume that $x \notin_{\mathit{update}(bl,s)} al$. From Lemma (C.3), it follows that $x \notin_s al \lhd bl$. By definition, $\mathcal{I}_e(\mathit{find}(x, bl \oplus (al \lhd bl))(s))(s)$ is $\mathcal{I}_e(\mathit{find}(x, bl)(s))(s)$ and, by definition, this is equivalent to $\mathit{update}(bl, s)(x)$. From Lemma (C.2), $\mathit{find}(x, al)(\mathit{update}(bl, s)) = x$ and therefore $\mathcal{I}_e(\mathit{find}(x, al)(\mathit{update}(bl, s)))(\mathit{update}(bl, s))$ is $\mathcal{I}_e(x)(\mathit{update}(bl, s))$. By definition of $\mathcal{I}_e$ and from $x \in \mathit{Names}$, this is equivalent to $\mathit{update}(bl, s)(x)$ completing the proof. □

Substitution Rules

Corollary C.1 *Find and Update*

For $al, bl \in$ *Alist,* $s \in$ *State and* $x \in$ *Names:*

$$\mathcal{I}_e(x)(\textit{update}(al, s)) = \mathcal{I}_e(\textit{find}(x, al, s))(s)$$

Proof. Immediate, from definition of $\mathcal{I}_e$ and *update*. □

Proof. *Rules of Figure (3.3)*

Rules (sr1), (sr2), (sr3), (sr4) and (sr5): Straightforward from definitions of substitution, $\equiv_s$, $\mathcal{I}_e$ and *find*. For (sr3) note that the conclusion reduces to

$$\mathcal{I}_e(\textit{find}(x, (t, r) \cdot al)(s))(s)) = \mathcal{I}_e(r)(s)$$

From the definition of *find* and the assumption that $x \equiv_s t$, $\textit{find}(x, (t, r) \cdot al)(s) = r$.

For (sr4), the conclusion reduces to

$$\mathcal{I}_e(\textit{find}(x, (t, r) \cdot al)(s))(s)) = \mathcal{I}_e(\textit{find}(x, al)(s))(s)$$

From the definition of *find* and the assumption $x \not\equiv_s t$, $\textit{find}(x, (t, r) \cdot al)(s) = \textit{find}(x, al)(s)$.

Rules (sr6) and (sr7): From the definitions and the fact that f is a name function, the left hand side of the equivalence in the assumption reduces to

$$\mathcal{I}_e(\textit{find}(\mathcal{I}_n(f(a_1, \dots, a_n), (t, r) \cdot al)(s)))(s)$$

For rule (sr6): since the assumption is that the equivalence holds, $\mathcal{I}_n(f(a_1, \dots, a_n))(s) = \mathcal{I}_n(t)(s)$, the definition of *find* reduces the conclusion to $\mathcal{I}_e(r)(s) = \mathcal{I}_e(r)(s)$ which is trivially true.

For rule (sr7): the assumption is that the equivalence with t is not true. From the definition of $\mathcal{I}_n$ and of substitution, the left hand side of the assumption, reduces to

$$\mathcal{I}_e(\textit{find}(\mathcal{I}_f(f)(\mathcal{I}_e(a_1)(\textit{update}(al, s)), \dots, \mathcal{I}_e(a_n)(\textit{update}(al, s))), al)(s))(s)$$

For each v_i of the assumption, $\mathcal{I}_e(v_i)(s) = \mathcal{I}_e(a_i)(\textit{update}(al, s))$. This leads to

$$\mathcal{I}_e(\textit{find}(\mathcal{I}_f(f)(\mathcal{I}_e(v_1)(s), \dots, \mathcal{I}_e(v_n)(s)), \cdot al)(s))(s)$$

From the definition of equivalence, substitution and the interpretation function $\mathcal{I}_e$ and $\mathcal{I}_n$, this is

$$\mathcal{I}_e(f(v_1, \dots, v_n) \triangleleft al)(s)$$

completing the proof for the rule.

Rules (sr8) and (sr9): For any state s, $\mathcal{I}_e(e \lhd (al \oplus bl))(s)$ is $\mathcal{I}_e(e)(\mathit{update}(al \oplus bl, s))$.

For rule (sr8), $bl = \mathit{nil}$: by extensionality, $\mathit{update}(al \oplus \mathit{nil}, s) = \mathit{update}(al, s)$ and it follows by definition that $\mathcal{I}_e(e)(\mathit{update}(al, s)) = \mathcal{I}_e(e \lhd al)(s)$.

For rule (sr9), $al = \mathit{nil}$: since, for any $x \in \mathit{Names}$ and $t \in \mathit{State}$, $\mathit{find}(x, \mathit{nil}, s) = x$, the proof is as for rule (sr8). By extensionality, $\mathit{update}(\mathit{nil} \oplus bl, s) = \mathit{update}(bl, s)$ and $\mathcal{I}_e(e \lhd \mathit{nil} \oplus bl)(s) = \mathcal{I}_e(e \lhd bl)(s)$.

Rule (sr8): By definition $\mathcal{I}_e(x \lhd bl \oplus (t, r) \cdot al)(s)$ is $\mathcal{I}_e(x)(\mathit{update}(bl \oplus (t, r) \cdot al))$ and, by Corollary (C.1), this is equivalent to $\mathcal{I}_e(\mathit{find}(x, bl \oplus (t, r) \cdot al, s))(s)$. Since $x \equiv_s t$, it follows that $x \in_s (t, r) \cdot al$ and $\mathit{find}(x, (t, r) \cdot al, s) = r$. The conclusion of the rule is then straightforward from $\mathcal{I}_e(\mathit{find}(x, (t, r) \cdot al, s))(s) = \mathcal{I}_e(r)(s)$.

Rule (sr9): As for rule (sr8), the conclusion reduces to

$$\mathcal{I}_e(\mathit{find}(x, bl \oplus (t, r) \cdot al, s))(s) = \mathcal{I}_e(\mathit{find}(x, bl \oplus al, s))(s)$$

There are two cases to consider: for the first, assume $x \in_s (t, r) \cdot al$. Since $x \not\equiv_s t$, it follows that $x \in_s al$ which leads to $\mathcal{I}_e(\mathit{find}(x, (t, r) \cdot al, s))(s) = \mathcal{I}_e(\mathit{find}(x, al, s))(s)$. Since $x \not\equiv_s t$, this reduces, by definition of *find*, to $\mathcal{I}_e(\mathit{find}(x, al, s))(s) = \mathcal{I}_e(\mathit{find}(x, al, s))(s)$ which is trivially true. For the second case, assume $x \notin_s (t, r) \cdot al$. It follows that $x \notin_s al$ and therefore the conclusion of the rule reduces to $\mathcal{I}_e(\mathit{find}(x, bl, s))(s) = \mathcal{I}_e(\mathit{find}(x, bl, s))(s)$, which is trivially true.

Rule (sr10): Let $x = \mathcal{I}_n(f(a_1, \ldots, a_n) \lhd bl \oplus (t, r) \cdot al)(s)$. By definition,

$$x = \mathcal{I}_f(f)(\mathcal{I}_e(a_1)(\mathit{update}(bl \oplus (t, r) \cdot al, s)), \ldots, \mathcal{I}_e(a_n)(\mathit{update}(bl \oplus (t, r) \cdot al, s)))$$

Since each v_i of the rule is equivalent to $a_n \lhd bl \oplus (t, r) \cdot al$, it follows that

$$x = \mathcal{I}_f(f)(\mathcal{I}_e(v_1)(s), \ldots, \mathcal{I}_e(v_n)(s))$$

Also by definition (and since $f \in \mathcal{F}n$),

$$\begin{array}{rl} & \mathcal{I}_e(f(a_1, \ldots, a_n) \lhd bl \oplus (t, r) \cdot al)(s) \\ = & \mathcal{I}_e(\mathcal{I}_f(f)(\mathcal{I}_e(a_1)(\mathit{update}(bl \oplus (t, r) \cdot al, s)), \ldots, \mathcal{I}_e(a_n)(\mathit{update}(bl \oplus (t, r) \cdot al, s)))) \\ & \quad (\mathit{update}(bl \oplus (t, r) \cdot al, s)) \\ = & \mathcal{I}_e(x)(\mathit{update}(bl \oplus (t, r) \cdot al, s)) \end{array}$$

As with rule (sr8), this reduces to $\mathcal{I}_e(\mathit{find}(x, bl \oplus (t, r) \cdot al, s))(s)$. Since $x \equiv_s t$, this is $\mathcal{I}_e(\mathit{find}(x, (t, r) \cdot al, s))(s)$ which reduces to $\mathcal{I}_e(r)(s)$, completing the proof.

Rule (sr11): As in the proof of rule (sr10), let $x = \mathcal{I}_n(f(a_1, \ldots, a_n) \lhd bl \oplus (t, r) \cdot al)(s)$. As before, $\mathcal{I}_e(f(a_1, \ldots, a_n) \lhd bl \oplus (t, r) \cdot al)(s)$ is equivalent to $\mathcal{I}_e(x)(\mathit{update}(bl \oplus (t, r) \cdot al, s))$. Note also that $\mathcal{I}_e(f(v_1, \ldots, v_n) \lhd bl \oplus al)(s)$ is equivalent to $\mathcal{I}_e(\mathcal{I}_f(f)(v_1, \ldots, v_n))(\mathit{update}(bl \oplus al, s))$, since the v_i are constants ($v_i \in \mathit{Values}$).

Note that

$$\begin{array}{ll} & \mathcal{I}_e(x)(\mathit{update}(bl \oplus (t,r) \cdot al, s)) = \mathcal{I}_e(f(v_1, \ldots, v_n))(\mathit{update}(bl \oplus (t,r) \cdot al, s)) \\ \text{and} & \mathcal{I}_e(x)(\mathit{update}(bl \oplus al, s)) = \mathcal{I}_e(f(v_1, \ldots, v_n))(\mathit{update}(bl \oplus al, s)) \end{array}$$

The proof therefore requires that

$$\mathcal{I}_e(x)(\mathit{update}(bl \oplus (t,r) \cdot al, s)) = \mathcal{I}_e(x)(\mathit{update}(bl \oplus al, s))$$

and as before, $\mathcal{I}_e(x)(\mathit{update}(bl \oplus (t,r) \cdot al, s))$ reduces to $\mathcal{I}_e(\mathit{find}(n, bl \oplus (t,r) \cdot al, s))(s)$ and $\mathcal{I}_e(x)(\mathit{update}(bl \oplus al, s)$ is $\mathcal{I}_e(\mathit{find}(x, bl \oplus al, s))(s)$.

There are two cases to consider: for the first, $x \in_s (t,r) \cdot al$. By definition, $\mathit{find}(x, bl \oplus (t,r) \cdot al, s) = \mathit{find}(x, (t,r) \cdot al, s)$. Since $x \not\equiv_s t$, it must be that $x \in_s al$ from which it follows that $\mathit{find}(x, (t,r) \cdot al, s) = \mathit{find}(x, al, s)$ and that $\mathit{find}(x, bl \oplus al, s) = \mathit{find}(x, al, s)$. This completes the proof for this case. For the second case, assume $x \notin_s (t,r) \cdot al$. As as consequence, $\mathit{find}(x, bl \oplus (t,r) \cdot al, s) = \mathit{find}(x, bl, s)$. It also follows that $x \notin_s al$ and therefore $\mathit{find}(x, bl \oplus al, s) = \mathit{find}(x, bl, s)$ completing the proof. □

C.2.3 Correct Assignment Lists

The definition of *correct?* used in the PVS theories differs from that given in Chapter 3. The correctness of an assignment list al is asserted by the predicate *assign?* which is defined by recursion on al.

Definition C.1 *Correct assignment lists*

The function *assign?* has type

$$(\mathit{Alist} \times \mathit{Alist}) \to \mathit{State} \to \mathit{boolean}$$

and definition

$$\begin{array}{rcl} \mathit{assign?}(\mathit{nil}, cl)(s) & \stackrel{\text{def}}{=} & \mathit{true} \\ \mathit{assign?}((x,e) \cdot al, cl)(s) & \stackrel{\text{def}}{=} & \begin{cases} e \equiv_s \mathit{find}(x, cl)(s) & \\ \wedge \mathit{assign?}(al, (x,e) \cdot cl)(s) & \text{if } v \in_s cl \\ \mathit{assign?}(al, (x,e) \cdot cl)(s) & \text{otherwise} \end{cases} \\ \mathit{assign?}(al \oplus bl, cl)(s) & \stackrel{\text{def}}{=} & \mathit{assign?}(al, \mathit{nil})(s) \wedge \mathit{assign?}(bl, \mathit{nil})(s) \end{array}$$

The predicate *correct?* is defined in terms of *assign?*.

$$\mathit{correct?}(al)(s) \stackrel{\text{def}}{=} \mathit{assign?}(al, \mathit{nil})(s)$$

□

The action of *assign?* for a simple list is to retain the names and values which have been encountered in the list cl. To determine whether (x, e) is a correct assignment, the last value assigned to x is found in cl and compared with e. If the two are equivalent then the assignment (x, e) is possible provided that the remainder of the assignment list al is correct.

The properties of Definition (3.18) follow from Lemma (C.5) and Lemma (C.6) below. Note that the first property is proved from Lemma (C.5). For $n \in \mathit{Names}$ and $s \in \mathit{State}$, when $bl = \mathit{nil}$, $n \in_s bl = \mathit{false}$.

Lemma C.5 *For simple list $al \in \mathit{Alist}$, $\mathit{simple?}(al)$, assignment list $bl \in \mathit{Alist}$ and state $s \in \mathit{State}$,*

$$\mathit{assign?}(al, bl)(s) \Leftrightarrow \begin{pmatrix} \forall(n : \mathit{Names}) : \\ \quad \exists(e : \mathcal{E}) : \forall(e_1 : \mathcal{E}) : e_1 \in \mathit{Assoc}(n, al)(s) \Rightarrow e \equiv_s e_1 \\ \quad \wedge n \in_s bl \Rightarrow e \equiv_s (\mathit{find}(n, bl)(s)) \end{pmatrix}$$

Proof. Let Φ be the property on the right hand side.

$$\begin{aligned} \Phi(al, bl, s) \stackrel{\text{def}}{=} \forall(n : \mathit{Names}) :& \\ &\exists(e : \mathcal{E}) : \forall(e_1 : \mathcal{E}) : e_1 \in \mathit{Assoc}(n, al)(s) \Rightarrow e \equiv_s e_1 \\ &\ \wedge n \in_s bl \Rightarrow e \equiv_s (\mathit{find}(n, bl)(s)) \end{aligned}$$

($\Rightarrow$) by induction on al with any $n \in \mathit{Names}$. Since al is a simple list, there are two cases.

Case $al = \mathit{nil}$: The proof follows immediately from definitions.

Case $al = (x, v) \cdot al'$ and inductive hypothesis $\mathit{assign?}(al', (x, v)bl, s) \Rightarrow \Phi(al', (x, v) \cdot bl, s)$:

If $x \not\equiv_s n$, the proof is immediate from the inductive hypothesis therefore assume that $x \equiv_s n$ and $x \in_s bl$. By definition of *assign?*, $v \equiv_s \mathit{find}(x, bl)(s)$. Assume that $x \in_s al'$, then from the assumption that $\mathit{assign?}(al', (x, v) \cdot bl, s)$ (from $\mathit{assign?}(al, bl, s)$) and from the inductive hypothesis there is a value e such that $e \equiv_s \mathit{find}(x, (x, v) \cdot bl)(s)$ and for all e_1 such that $e_1 \in \mathit{Assoc}(x, al')(s)$, $e_1 \equiv_s e$. By definition of *find*, $e \equiv_s v$ and therefore $v \equiv_s e_1$. From this, the assertion $\exists(e : \mathcal{E}) : \forall(e_1 : \mathcal{E}) : e_1 \in \mathit{Assoc}(n, al)(s) \Rightarrow e \equiv_s e_1$ follows immediately and completes the proof for this case.

Assume that $x \notin_s bl$. The proof is similar to that of the previous case.

($\Leftarrow$) by induction on al.

Case $al = \mathit{nil}$: The proof follows immediately from definitions.

Case $al = (x, v) \cdot al'$ and inductive hypothesis $\Phi(al', (x, v) \cdot bl, s) \Rightarrow \mathit{assign?}(al', (x, v) \cdot bl, s)$:

Assume that $x \in_s bl$. From the definition of $\Phi(al, bl, s)$ and $v \in \mathit{Assoc}(x, al)(s)$, $v \equiv_s \mathit{find}(x, bl)(s)$ is immediate. If the assertion $\Phi(al, bl, s) \Rightarrow \Phi(al', (x, v) \cdot bl, s)$ is *true* then $\mathit{assign?}(al', (x, v) \cdot bl, s)$ will be established by the inductive hypothesis.

To show $\Phi(al', (x, v) \cdot bl, s)$, assume $n \in \mathit{Names}$ such that $n \in_s al'$. By definition of $\Phi(al, bl, s)$, there is an expression $e \in \mathcal{E}$ such that $\forall e_1 \in \mathit{Assoc}(n, al)(s) : e \equiv_s e_1$ and $n \in_s bl \Rightarrow e \equiv_s$

$\mathit{find}(n, bl)(s)$. The first conjunct of $\Phi(al', (x, v) \cdot bl, s)$, $\forall e_1 \in \mathit{Assoc}(n, al')(s) : e \equiv_s e_1$, is straightforward from the definition of *Assoc* and *occs?*.

Assume that $x \equiv_s n$. From $x \in_s al$, it follows that $e \equiv_s v$ and that $e \equiv_s \mathit{find}(n, (x, v) \cdot bl)(s)$, completing the proof of $\Phi((x, v) \cdot al', bl, s)$ for this case.

Assume that $x \not\equiv_s n$. From $\Phi(al, bl, s)$ and $n \not\equiv_s x$, it follows that $e \equiv_s \mathit{find}(n, (x, v) \cdot bl)(s)$, completing the proof of $\Phi((x, v) \cdot al', bl, s)$. □

Lemma C.6 *For any assignment lists $al, bl \in \mathit{Alist}$ and state $s \in \mathit{State}$,*

$$\mathit{assign?}(al, bl)(s) \Leftrightarrow \begin{pmatrix} \mathit{assign?}(\mathit{initial}(al), bl)(s) \\ \wedge\forall(cl : \mathit{Alist}) : cl \ll al \wedge \mathit{combine?}(cl) \Rightarrow \mathit{assign?}(cl, \mathit{nil})(s) \end{pmatrix}$$

Proof. Let Φ be the property

$$\begin{aligned}\Phi(al, bl, s) \stackrel{\text{def}}{=} & \mathit{assign?}(\mathit{initial}(al), bl)(s) \\ & \wedge \forall(cl : \mathit{Alist}) : cl \ll al \wedge \mathit{combine?}(cl) \Rightarrow \mathit{assign?}(cl, \mathit{nil})(s)\end{aligned}$$

The proof is by induction on al.

Case $al = \mathit{nil}$: The proof is immediate from definitions.

Case $al = (x, v) \cdot al'$ with inductive hypothesis $\mathit{assign?}(al', (x, v) \cdot bl)(s) \Leftrightarrow \Phi(al', (x, v) \cdot bl, s)$:

($\Rightarrow$), the proof is similar for ($\Leftarrow$). From the assumption $\mathit{assign?}(al, bl)(s)$, and by definition, it follows that $\mathit{assign?}(al, (x, v) \cdot bl)(s)$ and, if $x \in_s bl$, $v \equiv_s \mathit{find}(x, bl)(s)$. From the definition of *initial*, $\mathit{assign?}(\mathit{initial}(al), bl)(s)$ iff $\mathit{assign?}((x, v) \cdot \mathit{initial}(al'), bl)(s)$. From the definition of *assign?*, this requires $v \equiv_s \mathit{find}(x, bl)(s)$, if $x \in_s bl$, and $\mathit{assign?}(\mathit{initial}(al'), (x, v) \cdot bl)(s)$. The assumption $\mathit{assign?}(al, bl)(s)$ establishes $v \equiv_s \mathit{find}(x, bl)(s)$, satisfying the first requirement. The second requirement, $\mathit{assign?}(\mathit{initial}(al'), (x, v) \cdot bl, s)$, is established by the inductive hypothesis $\mathit{assign?}(al', (x, v) \cdot bl, s) \Rightarrow \Phi(al', (x, v) \cdot bl, s)$ and the definition of Φ.

This establishes the first conjunct of $\Phi(al, bl, s)$. The second conjunct is straightforward from the inductive hypothesis, $\mathit{assign?}(al', (x, v) \cdot bl, s) \Rightarrow \Phi(al', (x, v) \cdot bl, s)$ and the assumption $\mathit{assign?}(al, bl, s)$.

Case $al = (cl_1 \oplus cl_2)$ with the inductive hypothesis $\mathit{assign?}(cl_1, bl_1)(s) \Leftrightarrow \Phi(cl_1, bl_1, s)$ and $\mathit{assign?}(cl_2, bl_2)(s) \Leftrightarrow \Phi(cl_2, bl_2, s)$ for any $bl_1, bl_2 \in \mathit{Alist}$: The proof is straightforward from $\mathit{assign?}(al, bl)(s) = \mathit{assign?}(cl_1, \mathit{nil})(s) \wedge \mathit{assign?}(cl_2, \mathit{nil})(s)$, with $bl_1 = bl_2 = \mathit{nil}$. □

C.2.4 Theorem (3.2)

The proof for *correct?* is based on that for *assign?*.

Theorem C.1 *For assignment lists $al, bl, cl \in \mathit{Alist}$ and state s,*

$$\mathit{assign?}(al, cl, \mathit{update}(bl, s)) \Leftrightarrow \mathit{assign?}(al \lhd bl, cl \lhd bl, s)$$

Proof. By induction on al. *Case* $al = nil$: By definition $al \lhd bl = nil$ and $assign?(nil, cl, s)$ is *true* for any $s \in State$ and $cl \in Alist$.

Case $al = (x, v) \cdot al'$ and with inductive hypothesis $assign?(al', (x, v) \cdot cl, update(bl, s))$ iff $assign?(al' \lhd bl, ((x, v) \cdot cl) \lhd bl, s)$:

($\Rightarrow$), the proof for ($\Leftarrow$) is similar.

From the assumption $assign?(al, cl, update(bl, s))$: if $x \in_{update(bl,s)} cl$ then $v \equiv_{update(bl,s)} find(x, cl)(update(bl, s))$. By definition this is $\mathcal{I}_e(v)(update(bl, s)) = find(x, cl)(update(bl, s))$. From Lemma (C.4), $find(x, cl)(update(bl, s))$ is equivalent to $find(x \lhd bl, cl \lhd bl)(s)$. From the definition of *find* this is $\mathcal{I}_e(v \lhd bl)(s) = \mathcal{I}_e(v)(update(bl, s))$ completing the proof the first requirement of *assign?*. Note that If $x \notin_{update(bl,s)} cl$ then from Lemma (C.3), $x \lhd bl \notin_s cl \lhd bl$.

The remainder of the proof, for $assign?(al' \lhd bl, (x, v) \cdot cl \lhd bl, s)$ follows from the assumption $assign?(al, cl, update(bl, s))$. By definition, $assign?(al', (x, v) \cdot cl, update(bl, s))$ is *true* and the proof is immediate from the inductive hypothesis.

Case $al = cl_1 \oplus cl_2$ and assuming the property is *true* for both cl_1 and cl_2:

The proof is straightforward from the definition of *assign?*, Lemma (C.3) and the inductive hypothesis. □

Proof. *Theorem (3.2)*

By definition, $correct?(al)(s) = assign?(al, nil, s)$ and the proof is immediate from Theorem (C.1) with $cl = nil$. □

C.3 Composition

Properties of the sequential composition of commands c_1 and c_2, $c_1; c_2$ are established in two steps. The first assume that c_1 is an assignment command and the proof is, generally, by induction on c_2. The second step establishes the properties for any command by induction on c_1.

C.3.1 Composition and Assignment

Lemma C.7 *Composition and assignment*

Assume commands $c, c_1, c_2 \in \mathcal{L}_0$, assignment lists $al, bl \in Alist$, label $l \in Labels$, label expressions $l_1, l_2 \in \mathcal{E}_l$ and states $s, t, u \in State$.

Result of composition. *The result of the composition of an assignment command with any command is an abstraction.*

$$\frac{\mathcal{I}(:= (al, l_1))(s, u) \quad \mathcal{I}(c)(u, t)}{\mathcal{I}(:= (al, l_1); c)(s, t)}$$

Reverse of composition. *If the composition of two assignments commands c_1 and c_2 begins in state s and ends in state t then there is an intermediate state in which c_1 ends and c_2 begins.*

$$\frac{\mathcal{I}(:= (al, l_1); := (bl, l_2))(s, t)}{\exists u : \mathcal{I}(:= (al, l_1))(s, u) \wedge \mathcal{I}(:= (bl, l_2))(u, t)}$$

Composition with a labelled command. *If assignment command c_1 begins in state s and the successor expression of c_1 is not equivalent in s to the label of $l : c$ then the composition of c_1 with $l : c$ is equivalent to the command c_1.*

$$\frac{\mathcal{I}(:= (al, l_1); l : c)(s, t) \quad l_1 \not\equiv_s l}{\mathcal{I}(:= (al, l_1))(s, t)}$$

Composition with a regular labelled command *If assignment command c_1 begins in state s and the successor expression of c_1 is equivalent in s to the label of $l : c$ and $l : c$ is a regular command, then there is an intermediate state between c_1 and $l : c$.*

$$\frac{\mathcal{I}(:= (al, l_1); l : c)(s, t) \quad \textit{regular?}(l : c) \quad l_1 \equiv_s l}{\exists u : \mathcal{I}(:= (al, l_1))(s, u) \wedge \mathcal{I}(l : c)(u, t)}$$

Proof. *Result of composition*

By induction on c. The cases when c is not an assignment command are straightforward from the inductive hypothesis.

Case $c = (l' : c')$ with inductive hypothesis $\mathcal{I}(:= (al, l_1); c')(s, t)$:

From $\mathcal{I}(:= (al, l_1))(s, u)$, $\mathcal{I}_e(pc)(u) = \mathcal{I}_e(l_1)(s)$ and, from Corollary (3.2), $\mathcal{I}_b(\textbf{equal}(pc, l_1))(s)$. From the definition of composition with a labeled command and of the interpretation of a conditional command, the conclusion is equivalent to $\mathcal{I}(:= (al, l_1); c')(s, t)$ and is straightforward from the inductive hypothesis.

Case $c = (\textbf{if}\ b\ \textbf{then}\ c_t\ \textbf{else}\ c_f)$ with the inductive hypothesis $\mathcal{I}(:= (al, l_1); c_t)(s, t)$ and $\mathcal{I}(:= (al, l_1); c_f)(s, t)$: The proof is straightforward from the definition of composition with a conditional and from the assumptions.

Case $c = (:= (bl, l_2))$: From $\mathcal{I}(:= (al, l_1))(s, u)$, *correct?*$(al)(s)$ and $u =$ *update*$((pc, l_1) \cdot al, s)$. From $\mathcal{I}(:= (bl, l_2))(u, t)$, *correct?*$(bl)(u)$ and $t =$ *update*$((pc, l_2) \cdot bl, u)$. The definition of composition leads to

$$\mathcal{I}(:= (((pc, l_1) \cdot al) \oplus (bl \triangleleft (pc, l_1) \cdot al), l_2 \triangleleft ((pc, l_1) \cdot al)))(s, t)$$

as the conclusion to be proved. From the definition of $\mathcal{I}$, there are two requirements: the correctness of the assignment list and the change to the state.

Correctness: The definition of $\mathcal{I}$ requires

$$\textit{correct?}((pc, l_2 \triangleleft ((pc, l_1) \cdot al)) \cdot (((pc, l_1) \cdot al) \oplus (bl \triangleleft (pc, l_1) \cdot al)))(s)$$

This reduce to the three correctness statements *correct?*$((pc, l_2 \lhd al) \cdot nil)(s)$, *correct?*$((pc, l_1) \cdot al, s)$ and *correct?*$(bl \lhd (pc, l_1) \cdot al)(s)$ (see the Corollary 3.1 and Definition 3.18 of *correct?*). *correct?*$((pc, l_2 \lhd al) \cdot nil)(s)$ is trivially *true* and *correct?*$((pc, l_1) \cdot al, s)$ follows immediately from $\mathcal{I}(:= (al, l_1))(s, u)$. From Theorem (3.2), *correct?*$(bl \lhd (pc, l_1) \cdot al)(s)$ is equivalent to *correct?*$(bl)(update((pc, l_1) \cdot al, s))$. From $\mathcal{I}(:= (al, l_1))(s, u)$, $u = update((pc, l_1) \cdot al, s)$ and *correct?*$(bl)(u)$ follows from $\mathcal{I}(:= (al, l_1))(u, t)$.

State update: The definition of $\mathcal{I}$ requires

$$t = update((pc, l_2 \lhd ((pc, l_1) \cdot al)) \cdot (((pc, l_1) \cdot al) \oplus (bl \lhd (pc, l_1) \cdot al)), s)$$

From the assumption $\mathcal{I}(:= (al, l_1))(s, u)$, $u = update((pc, l_1) \cdot al, s)$. Also from assumption $\mathcal{I}(:= (bl, l_2))(u, t)$, $t = update((pc, l_2) \cdot bl, u)$. Replacing u gives, $t = update((pc, l_2) \cdot bl, update((pc, l_1) \cdot al, s))$ and the proof is straightforward by extensionality. □

Proof. *Reverse of composition*

The intermediate state u is $update((pc, l_1) \cdot al, s)$. The correctness of the assignment lists, *correct?*$((pc, l_1) \cdot al)(s)$ and *correct?*$((pc, l_2) \cdot bl)(u)$ is straightforward from Theorem (3.2) and the correctness of the assignment list in the assumption. The state update $t = update((pc, l_2) \cdot bl, update((pc, l_1) \cdot al, s))$ is also straightforward from the assumption and Theorem(3.1). □

Proof. *Composition with a labelled command*

The conclusion follows from the assumption $l_1 \not\equiv_s l$, Corollary (3.2), the definition of composition and the interpretation of a conditional command. □

Proof. *Composition with a regular labelled command*

From the assumption $l_1 \equiv_s l$, the definition of the composition and the interpretation of the conditional command, the assumption reduces to $\mathcal{I}(:= (al, l_1); c)(s, t)$.

The intermediate state u for all cases is $u = update((pc, l_1) \cdot al, s)$. From the definition of composition, the interpretation of a conditional command and the assumption $l_1 \equiv_s l$, the conclusion requires that $\mathcal{I}(c)(u, t)$.

By induction on c. When c is an assignment command, the proof is as for the previous property. When c is a conditional command, the proof is straightforward from the inductive hypothesis (that the property is *true* for the branches of the conditional).

Assume that $c = l' : c'$ and the property is *true* for c'. Note that if $l : c$ is a regular command then so is $l : l' : c'$: every labelled command which is a sub-term of $l : c$ is labelled with l and $l' = l$. From the interpretation of an assignment command, $\mathcal{I}(:= (al, l_1))(s, u)$ implies $pc \equiv_u l_1$ and, from the assumption, $pc \equiv_u l$. From the inductive hypothesis, the property is *true* for c' and there is a state u' such that $\mathcal{I}(:= (al, l_1))(s, u')$ and $\mathcal{I}(c')(u', t)$. From Lemma (3.3), $u' = u$ and, since $\mathcal{I}_e(pc)(u) = l'$, it follows that $\mathcal{I}(l' : c')(u, t)$ is *true*. □

C.3.2 Theorem (3.3)

Proof. By induction on c_1. The case when c_1 is a conditional command follows from definition of composition and the assumption that the property is *true* for the branches of the conditional. If c_1 is a labeled command $l' : c'$, and assuming the property is *true* for c', the conclusion follows from the definition of composition, the assumption that $l' = l$ (c is a regular command) and the inductive hypothesis.

Assume c is the assignment command $:= (al, l_1)$. The proof is by induction on c_2. When c_2 is a conditional command, the conclusion follows from the definition of composition and the assumption that the property holds for the branches of the conditional. When c_2 is an assignment command, the conclusion is immediate from the definition of composition. Assume c_2 is the labeled command $l' : c'$ and that the property holds for c'. From the definition of composition, $l : c_1; c_2$ is l : **if equal**(l_1, l') **then** $c_1; c'$ **else** c_1. From the inductive hypothesis, *regular?*$(l : c_1) \Rightarrow$ *regular?*$((l : c_1); c')$. Since every sub-term of $l : c_1; c'$ is also regular (for every sub-term labeled with l_2, $l_2 = l$), so is $c_1; c'$. The conclusion that $l : c_1; c'$ is regular follows since the only other labeled command occurring in $l : c_1; c'$ is c_1 and therefore every labeled command in $l : c_1; c'$ is labeled with l. □

C.3.3 Theorem (3.4)

Proof.

By induction on c_1.

Case $c_1 =$ **if** b **then** c_t **else** c_f and the property is *true* for c_t and c_f. From the assumption there is a state u such that $\mathcal{I}(c_1)(s, u)$ and $\mathcal{I}(c_2)(u, t)$. From the definition of composition $c_1; c_2$ is **if** b **then** $c_t; c_2$ **else** $c_f; c_2$. The conclusion to be proved is that if $\mathcal{I}_b(b)(s)$ then $\mathcal{I}(c_t; c_2)(s, t)$ and if $\neg\mathcal{I}_b(b)(s)$ then $\mathcal{I}(c_f; c_2)(s, t)$. From the assumption $\mathcal{I}(c_1)(s, u)$, either $\mathcal{I}_b(b)(s)$ and $\mathcal{I}(c_t)(s, u)$ or $\neg\mathcal{I}_b(b)(s)$ and $\mathcal{I}(c_f)(s, u)$. Assume $\mathcal{I}_b(b)(s)$, the inductive hypothesis is $(\exists u' : \mathcal{I}(c_t)(s, u') \wedge \mathcal{I}(c_2)(u', t)) \Rightarrow \mathcal{I}(c_t; c_2)(s, t)$ and the conclusion for this case follows with $u' = u$. The case for $\neg\mathcal{I}_b(b)(s)$ is similar.

Case $c_1 = l' : c'$ with the assumption that the property is *true* for c'. The proof is similar to that of the conditional command.

Case $c_1 = (:= (al, l))$. The proof is immediate from Lemma (C.7). □

C.3.4 Theorem (3.5)

Proof.

By induction on c_1, the case for the conditional and labelled commands is straightforward from the inductive hypothesis.

Assume $c_1 = (:= (al, l_1))$. By definition, $c_1; (l : c_2)$ is **if equal**(l_1, l) **then** $c_1; c_2$ **else** c_1. From the assumption $\mathcal{I}(c_1)(s,t)$, $\mathcal{I}_e(pc)(t) = \mathcal{I}_e(l_1)(s)$. From $pc \not\equiv_t l$ and $l \in \mathit{Labels}$, $l_1 \not\equiv_s l$ and $\mathcal{I}_b(\textbf{equal}(l_1, l))(s)$ is *false*. Therefore $\mathcal{I}(\textbf{if equal}(l_1, l)\ \textbf{then}\ c_1; c_2\ \textbf{else}\ c_1)(s,t)$ is $\mathcal{I}(c_1)(s,t)$. □

C.3.5 Theorem (3.6)

The proof is by induction on c_1, the case for the assignment command is proved by induction on c_2.

Lemma C.8 *For assignment command* $c_1 = (:= (al, l))$*, command* $c_2 \in \mathcal{L}_0$ *and states* s, t*,*

$$\frac{\mathcal{I}(c_1; c_2)(s,t)}{(\exists (u : \mathit{State}) : \mathcal{I}(c_1)(s,u) \wedge \mathcal{I}(c_2)(u,t)) \vee \mathcal{I}(c_1)(s,t)}$$

Proof.

From the interpretation of an assignment command, if there is an intermediate state u, it is $u = \mathit{update}((pc, l) \cdot al, s)$, otherwise $t = \mathit{update}((pc, l) \cdot al, s)$.

By induction on c_2.

Case $c_2 =$ **if** b **then** c_t **else** c_f and assuming the property is *true* for $c_1; c_t$ and $c_1; c_f$. By definition, $c_1; c_2$ is the conditional command **if** $b \lhd ((pc, l) \cdot al)$ **then** $c_1; c_t$ **else** $c_1; c_f$. Assume $\mathcal{I}_b(b \lhd ((pc, l) \cdot al))(s)$, then $\mathcal{I}(c_1; c_2)(s,t)$ is $\mathcal{I}(c_1; c_t)(s,t)$. Assume there is no intermediate state u such that $\mathcal{I}(c_1)(s,u)$ and $\mathcal{I}(c_t)(u,t)$, then from the hypothesis, $\mathcal{I}(c_1)(s,t)$ and the case is proved. Assume that there is an intermediate state u such that $\mathcal{I}(c_1)(s,u)$ and $\mathcal{I}(c_t)(u,t)$. The value of the boolean condition b in u is $\mathcal{I}_b(b)(\mathit{update}(al, s))$ which is equivalent to $\mathcal{I}_b(b \lhd bl)(s)$. Therefore $\mathcal{I}(c_2)(u,t)$ is equivalent to $\mathcal{I}(c_t)(u,t)$ completing the proof for this case. The case when $\neg\mathcal{I}_b(b \lhd ((pc, l) \cdot al))(s)$ is similar.

Case $c_2 = (l : c)$ and assuming the property is *true* for c. The proof is similar to that of the conditional command.

Case $c_2 = (:= bl, l_2)$, the proof follows from Lemma (C.7). □

Proof. *Theorem (3.6)*

Straightforward by induction on c_1 and from Lemma (C.8). □

C.3.6 Theorem (3.7)

The proof is similar to that of Theorem (3.6). The proof is by specializing the property when c_1 is an assignment command by labeling the command c_2.

Lemma C.9 *For assignment command $c_1 = (:= (al, l))$ command $c_2 \in \mathcal{L}$, label $l_2 \in$ Labels and states s, t,*

$$\frac{\mathcal{I}(c_1; (l_2 : c_2))(s,t) \quad \textit{regular?}(l_2 : c_2)}{(\exists (u : \textit{State}) : \mathcal{I}(c_1)(s,u) \wedge \mathcal{I}(l_2 : c_2)(u,t)) \vee (\mathcal{I}(c_1)(s,t) \wedge \textit{pc} \not\equiv_t l_2}$$

Proof.

By induction on c_2. Note that since $l_2 : c_2$ is a regular command, every sub-term c' of c_2, $l_2 : c'$ is also regular. Also, from the definition of composition, $c_1; (l_2 : c_2)$ is the conditional **if equal**(l, l_2) **then** $c_1; c_2$ **else** c_1. If $\neg\mathcal{I}_b(\mathbf{equal}(l, l_2))(s)$ then from the assumption, it follows that $\mathcal{I}(c_1)(s,t)$, $\mathcal{I}_e(pc)(t) = \mathcal{I}_e(l)(s)$ and $l \not\equiv_s l_2$. The conclusion, $\mathcal{I}(c_1)(s,t)$, and *pc* $\not\equiv_s l_2$ follows immediately. For the remainder, assume that $l \equiv_s l_2$.

Case $c_2 =$ **if** b **then** c_t **else** c_f and assuming the property is *true* for c_t and c_f. The proof is straightforward from the inductive hypothesis and is similar to that given for Lemma (C.8).

Case $c_2 = l' : c'$ and assuming the property is true for c'. Since $l_2 : c_2$ is a regular command, $l' = l_2$. From the inductive hypothesis, either there is an intermediate state u such that $\mathcal{I}(c_1)(s,u)$ and $\mathcal{I}(l_2 : c')(u,t)$ or $\mathcal{I}(c_1)(s,t)$ and *pc* $\not\equiv_t l_2$. Assume there is an intermediate state u: from $\mathcal{I}(l_2 : c')(u,t)$ and from $l' = l_2$ it follows that $\mathcal{I}(l_2 : l' : c')(u,t)$. The conclusion that there is a state u such that $\mathcal{I}(c_1)(s,u)$ and $\mathcal{I}(l_2 : l' : c')(u,t)$ is therefore *true*. Assume there is no intermediate state. From the inductive hypothesis, $\mathcal{I}(c_1)(s,t)$ and *pc* $\not\equiv_t l_2$ is trivially *true* completing that proof for this case.

Case $c_2 := (:= (bl, l_3))$. Since $l \equiv_s l_2$, the assumption reduces to $\mathcal{I}(c_1; c_2)(s,t)$ and the conclusion follows from Lemma (C.7). □

Proof. *Theorem (3.7)*

Straightforward by induction on c_1 and by Lemma (C.9) for the case when c_1 is an assignment command. □

C.3.7 Theorem (3.8)

Proof.

From the assumption, $c_1; c_2$ is enabled in s and there is no $t \in$ *State* such that $\mathcal{I}(c_1; c_2)(s,t)$.

From composition, *label*(c_1) = *label*$(c_1; c_2)$ and c_1 is enabled in s. Assume that there is a $t_1 \in$ *State* such that $\mathcal{I}(c_1)(s,t_1)$. Also assume that $\neg pc \equiv_{t_1}$ *label*(c_2). From Theorem (3.5), $\mathcal{I}(c_1; c_2)(s,t_1) = \mathcal{I}(c_1)(s,t_1)$ contradicting the assumption that there is no such t_1.

Assume that *pc* $\equiv_{t_1}$ *label*(c_2) and let state u of the conclusion be t_1, $u = t_1$; command c_2 is enabled in u. Assume that there is a state t_2 such that $\mathcal{I}(c_2)(u,t_2)$: t_1 is an intermediate state such that $\mathcal{I}(c_1)(s,t_1)$ and $\mathcal{I}(c_2)(t_1,t_2)$. It follows, from Theorem (3.4), that $\mathcal{I}(c_1; c_2)(s,t_2)$. This contradicts the assumption that there is no state t such that $\mathcal{I}(c_1; c_2)(s,t)$ and completes the proof. □

C.3.8 Theorem (3.9)

Proof.

1. c_1 halts.

   From composition, $c_1; c_2$ is enabled in s iff c_1 is enabled in s. The property to be proved is therefore that if $\forall t_1 : \neg\mathcal{I}(c_1)(s, t_1)$ then $\forall t : \neg\mathcal{I}(c_1; c_2)(s, t)$. Assume that there is a state t such that $\mathcal{I}(c_1; c_2)(s, t)$. From Theorem (3.6) there is a state u such that $\mathcal{I}(c_1)(s, u)$ which is a contradiction.

2. c_2 halts.

   Note that if c_2 halts in state t then it is enabled in t. Assume that there is a state u such that $\mathcal{I}(c_1; c_2)(s, u)$. From Theorem (3.7) either $\mathcal{I}(c_1)(s, u)$ and *pc* $\not\equiv_u$ *label*(c_2) or there is a state t' such that $\mathcal{I}(c_1)(s, t')$ and $\mathcal{I}(c_2)(t', u)$.

   Assume that $\mathcal{I}(c_1)(s, u)$ and *pc* $\not\equiv_u$ *label*(c_2): The commands are deterministic (Lemma 3.3) and $u = t$. From the assumption *halt?*$(c_2)(t)$, it follows that *enabled*$(c_2)(t)$ is *true*. This contradicts the assumption that *pc* $\not\equiv_u$ *label*(c_2).

   Assume there is a t' such that $\mathcal{I}(c_1)(s, t')$ and $\mathcal{I}(c_2)(t', u)$. The commands are deterministic and from Lemma (3.3), $t = t'$. Therefore $\mathcal{I}(c_2)(t, u)$ is *true* contradicting the assumption that c_2 halts in t.

$\square$

Appendix D

Proofs: Programs

This appendix contains proofs of the theorems and lemmas of Chapter 4, the definition of a path through a program and additional properties of programs, generalized sequential composition and the relationship between loops and traces. The order in which the proofs and definitions are presented follows that of Chapter 4.

D.1 Program Syntax and Semantics

The syntactic properties of the programs include the rules which describe how programs may be constructed.

D.1.1 Rules of Construction for Programs

Lemma D.1 *Rules of construction*

1. *The empty set is a program, program?*$(\{\})$.
2. *Any subset of a program is also a program.*

$$\frac{\mathit{program?}(p_1) \quad p_2 \subseteq p_1}{\mathit{program?}(p_2)}$$

3. *The addition of a command* $c \in \mathcal{L}$ *to a program results in a program.*

$$\frac{\mathit{program?}(p)}{\mathit{program?}(p + c)}$$

1. Since the empty set has no commands, the proof is immediate.

2. Let p be a program. If p' is a subset of p which is not a program then there are commands $c_1, c_2 \in p'$ such that *label*$(c_1) =$ *label*$(c_2))$. Since $p' \subseteq p$, both c_1 and c_2 are members of p contradicting the assertion that p is a program.

3. *Case* $\exists c_1 \in p$: *label*$(c_1) =$ *label*(c): By definition, $p + c = p$ and by the assumption, *program?*$(p + c)$.

   Case $\neg\exists c_1 \in p$: *label*$(c_1) =$ *label*(c): Let p' be the set of commands $p \cup \{c\}$. Assume that p' is not a program, then there is a command c_1 in p' such that *label*$(c_1) =$ *label*(c) and $c_1 \neq c$. This contradicts the assumption that there is no such command.

D.1.2 Theorem (4.1): Program Induction

The proof for both induction schemes is by induction on a finite set. In the proof, Φ_p will be the property of Theorem (4.1) and Φ_f the property of the finite induction schemes.

Proof. *Induction*

By finite induction. Let Φ_f be $\lambda(p : \textit{FiniteSet}(\mathcal{L}) : \textit{program?}(p) \Rightarrow \Phi_p(p)$.

$\Phi_f(\{\})$: From the assumptions of Theorem (4.1), $\Phi_p(\{\})$, and since the empty set is a program, $\Phi_f(\{\})$ is *true*.

$\forall b : \Phi_f(b) \Rightarrow (\forall x : x \notin b \Rightarrow \Phi_f(b \cup \{x\}))$:

To show $\Phi_f(b \cup \{x\}))$, it is necessary to show that *program?*$(b \cup \{x\}) \Rightarrow \Phi_p(b \cup \{x\})$. Note that b is a program since it is a subset of $b \cup \{x\}$. Also, $(b + x) = b \cup \{x\}$ since otherwise there is a $c \in b$ such that *label*$(c) =$ *label*(x) and $b \cup \{x\}$ is not a program. The proof therefore follows From the assumption of Theorem (4.1), $(\forall(p : \mathcal{P}) : \Phi_p(p) \Rightarrow \forall(c : \mathcal{L}) : \Phi_p(p + c))$ with $p = b$ and $c = x$. □

Proof. *Strong Induction*

By strong finite induction. Let Φ_f be $\lambda(p_1 : \textit{FiniteSet}(\mathcal{L})) : \textit{program?}(p_1) \Rightarrow (\forall p'_1 : p'_1 \subseteq p_1 \Rightarrow \Phi_p(p'_1))$.

Let p_1 and p'_1 be p. Since *program?*(p) and $p \subseteq p$, the property $\Phi_p(p)$ follows from the property Φ_f established by the finite induction scheme.

To show that the assumptions of the finite induction scheme are satisfied, the property $\Phi_f(b)$ must be established. Since *program?*(b) (from $\Phi_f(b)$), any subset p'_1 of b is also a program. If $p'_1 = b$ then the property $\Phi_p(b)$ follows from the assumption that $\Phi_p(p_1)'$ (definition of Φ_f). Assume that $p'_1 \subset b$, then from Φ_f, $\Phi_p(p'_1)$ and from the assumption of strong program induction $(\forall p : (\forall p_1 : p_1 \subset p_1 \Rightarrow \Phi_p(p_1)) \Rightarrow \Phi_p(p))$, $\Phi_p(b)$ is immediate. □

D.1.3 Lemma (D.2) and Lemma (4.1)

Lemma D.2 *Selection of commands*

For program p *and behaviour* σ*,*

$$\frac{\mathcal{I}_p(p)(\sigma)}{\forall(n:\mathbb{N}) : \mathcal{I}_c(\mathit{at}(p, \mathcal{I}_e(\mathit{pc})(\sigma_n(0))))(\sigma_n(0), \sigma_n(1))}$$

Proof. *Lemma (D.2)*

Since $\mathcal{I}(p)(\sigma)$, for every $i \in \mathbb{N}$ there is a command $c \in p$ such that $\mathcal{I}(c)(\sigma(i), \sigma(i+1))$. Command c is labeled ($p \subset \mathcal{L}$) and from the semantics of the command, $\mathcal{I}_e(\mathit{pc})(\sigma(i)) = \mathit{label}(c)$. Since p is a program, there is no other command $c' \in p$ such that $\mathit{label}(c) = \mathit{label}(c')$, therefore $\mathit{at}(p, \mathcal{I}_e(\mathit{pc})(\sigma(i))) = c$. □

Proof. *Lemma (4.1)*

($\Rightarrow$), the proof is similar for ($\Leftarrow$): Assume $\mathcal{I}(p)(\sigma)$. For every $i \in \mathbb{N}$, there is a command $c \in p$ such that $\mathcal{I}(c)(\sigma(i), \sigma(i+1))$. From the assumption there is also a command $c' \in p'$ such that $\mathcal{I}(c')(\sigma(i), \sigma(i+1))$ and the proof for this case is complete. □

D.2 Additional Induction Schemes

For the transitive closure of a relation two induction schemes are assumed in addition to the scheme which follows from its definition.

Theorem D.1 *Induction Schemes for Transitive Closure*

Assume type T*, property* $\Phi : (\mathit{Set}(T), T, T) \to \mathit{boolean}$*, relation* $R : (\mathit{Set}(T), T, T) \to \mathit{boolean}$*, set* $a : \mathit{Set}(T)$ *and* $x, y, z : T$*. To establish the property* $\forall a, x, y : R^+(a, x, y) \Rightarrow \Phi(a, x, y)$ *it is necessary to establish either of the properties* Left Induction *or* Right Induction *below.*

Left induction:

$$\frac{R(a,x,y)}{\Phi(a,x,y)} \qquad \frac{R(a,x,y) \quad R^+(a,y,z) \quad \Phi(a,y,z)}{\Phi(a,x,z)}$$

Right induction:

$$\frac{R(a,x,y)}{\Phi(a,x,y)} \qquad \frac{R^+(a,x,y) \quad \Phi(a,x,y) \quad R(a,y,z)}{\Phi(a,x,z)}$$

The proofs for these induction schemes can be derived from induction on the transitive closure of R.

D.3 Transition Relations

The transition relations are the *leads-to* relation and the trace relations through a program. A number of basic properties of the trace relations are used to establish the lemmas and theorems of Chapter 4.

D.3.1 Theorem 4.2

Proof.

By induction on $\leadsto$.

Case $\exists c \in p \land s \overset{c}{\leadsto} t$: From the definition of $\mathcal{I}(p)(\sigma)$ there is a command c' such that $\mathcal{I}(c')(\sigma(0), \sigma(1))$. Since p is a program, all commands are labelled, c' and c are enabled in s, $c' = c$. Since the commands are deterministic, $t = \sigma(1)$.

Case $\exists u : s \overset{p}{\leadsto} u \land u \overset{p}{\leadsto} t$ and the property holds between states s and u and between u and t: From the assumptions that $\mathcal{I}(p)(\sigma)$ and that the property holds between s and u, there is an $i > 0$ such that $u = \sigma(i)$. From Corollary (4.1), $\mathcal{I}(p)(\sigma_i)$ follows from the assumption $\mathcal{I}(p)(\sigma)$. This together with $u = \sigma_i(0)$ satisfies the assumptions of the inductive hypothesis between u and t. The conclusion that there is an $m > 0$ such that $t = m$ follows from the inductive hypothesis. □

D.3.2 Traces

The traces relation is stronger than *leads-to* and the maximal traces are stronger than both of these.

Lemma D.3 *For states s, t and program p, if there is a trace from s to t through p then s leads to t through p. If there is a maximal trace from s to t then there is a trace from state s to state t.*

$$\frac{\textit{trace}(p, s, t)}{s \overset{p}{\leadsto} t} \quad \frac{\textit{mtrace}(p, s, t)}{\textit{trace}(p, s, t)} \quad \frac{\textit{rmtrace}(p, s, t)}{\textit{trace}(p, s, t)}$$

Proof. $\textit{trace}(p, s, t) \Rightarrow s \overset{p}{\leadsto} t$ is straightforward by induction on *trace*. $\textit{mtrace}(p, s, t) \Rightarrow \textit{trace}(p, st,)$ and $\textit{rmtrace}(p, s, t) \Rightarrow \textit{trace}(p, st,)$ are immediate by definition. □

Some basic properties of the trace relations are used in the proofs.

Lemma D.4 *For program p and states s, t,*

$$\textit{tset}(p, s, t) \subseteq p$$

Proof. Immediate by induction on p and definition of $\textit{tset}(p, s, t)$. □

Lemma D.5 *For $p \in \mathcal{P}$, $c \in p$ and $s, t, u \in$ State,*

$$\frac{s \overset{c}{\rightsquigarrow} u}{(\mathit{tset}(p, s, t) - \{c\}) = \mathit{tset}(p - \{c\}, u, t)}$$

Proof.

By extensionality: with $c' \in \mathcal{L}$, $c' \in (\mathit{tset}(p, s, t) - \{c\}) \Leftrightarrow c' \in \mathit{tset}(p - \{c\}, s, t)$. Note that for every $c_1 \in p$, $u' \in \mathit{State}$ such that $s \overset{c_1}{\rightsquigarrow} u'$, $c_1 = c$ and $u = u'$.

($\Rightarrow$): Assume $s \overset{c'}{\rightsquigarrow} t$. It follows that $c' = c$, $c' \notin \mathit{tset}(p, s, t) - \{c\}$ and, by Lemma (D.4), $c' \notin \mathit{tset}(p - \{c\}, u, t)$. Assume $\exists u' : s \overset{c'}{\rightsquigarrow} u' \wedge \mathit{trace}(p - \{c'\}, u, t)$. It follows that $c' = c$, $c' \notin (\mathit{tset}(p, s, t) - \{c\})$ and, by Lemma (D.4), $c' \notin \mathit{tset}(p - \{c\}, u, t)$. Assume $\exists u', c_1 : s \overset{c_1}{\rightsquigarrow} u' \wedge c' \in \mathit{tset}(p - \{c_1\}, u, t)$. It follows that $c_1 = c$, $u' = u$ and $c' \in \mathit{tset}(p - \{c\}, u, t)$ is straightforward.

($\Leftarrow$): If $c' \in \mathit{tset}(p - \{c\}, u, t)$ then $s \overset{c'}{\rightsquigarrow} u'$ is *false* for all $u' \in \mathit{State}$ and $c' \neq c$. From the definition of *tset*, it follows that $c' \in \mathit{tset}(p, s, t) - \{c\}$. □

Lemma D.6 *For $p, p' \in \mathcal{P}$, and $s, t \in$ State,*

$$\frac{p \subseteq p' \quad \mathit{trace}(p, s, t)}{\mathit{trace}(p', s, t)}$$

Proof. Straightforward, by induction and from the observation that every command in p is also in p' □

Lemma D.7 *For $p \in \mathcal{P}$ and $s, t \in State$,*

$$\frac{\mathit{tset}(p, s, t) \neq \{\}}{\mathit{trace}(p, s, t)}$$

Proof.

By strong induction on p.

Since $\mathit{tset}(p, s, t)$ is not empty, there is a $c \in \mathit{tset}(p, s, t)$.

Case $s \overset{c}{\rightsquigarrow} t$: $\mathit{trace}(p, s, t)$ follows immediately.

Case $\exists u : s \overset{c}{\rightsquigarrow} u \wedge \mathit{trace}(p - \{c\}, u, t)$: $\mathit{trace}(p, s, t)$ follows immediately from definition.

Case $\exists (u \in State, c' \in p) : c' \neq c \wedge s \overset{c'}{\rightsquigarrow} u \wedge c \in \mathit{tset}(p - \{c'\}, u, t)$ and the property holds for $p - \{c'\}$: Since $c \in \mathit{trace}(p - \{c'\}, u, t)$, $\mathit{trace}(p - \{c'\}, u, t) \neq \{\}$ and there is a trace $\mathit{trace}(p - \{c'\}, u, t)$. $\mathit{trace}(p, s, t)$ follows from the definition of *trace*. □

Lemma D.8 *For $p \in \mathcal{P}$, $c \in \mathcal{L}$ and $s, t, u \in State$,*

$$\frac{\textit{trace}(p, s, u) \quad \textit{program?}(p \cup \{c\}) \quad c \notin p \quad u \overset{c}{\leadsto} t}{\textit{trace}(p \cup \{c\}, s, t)}$$

Proof.

By induction on $\textit{trace}(p, s, u)$.

Case $c' \in p$ and $s \overset{c'}{\leadsto} u$: Since $c \notin p$, $c \in (p \cup \{c\}) - \{c'\}$. From $u \overset{c}{\leadsto} t$ it follows that there is a trace $\textit{trace}(p \cup \{c\} - \{c'\}, u, t)$ and $\textit{trace}(p \cup \{c\}, s, t)$ follows from the definitions.

Case $c' \in p$, $u' \in State$, $s \overset{c'}{\leadsto} u'$, $\textit{trace}(p - \{c'\}, u', u)$ and the property is true for $\textit{trace}(p - \{c'\}, u', u)$: Since $c \notin p$, $c \notin p - \{c'\}$ and the inductive hypothesis leads to $\textit{trace}(p - \{c'\} \cup \{c\}, u', t)$. The conclusion $\textit{trace}(p \cup \{c\}, s, t)$ follows from the definition of *trace* and $p - \{c'\} \cup \{c\} = (p \cup \{c\}) - \{c'\}$. □

D.3.3 Lemma (4.3)

Proof.

($\Rightarrow$) By induction on *trace*.

Case: $c \in p$ and $s \overset{c}{\leadsto} t$: Immediate from definitions.

Case: $c \in p$ and $s \overset{c}{\leadsto} u$ and $\textit{trace}(p - \{c\}, u, t)$ with inductive hypothesis $\textit{trace}(\textit{tset}(p - \{c\}, u, t), u, t)$: By definition of *tset* and from the assumptions, $c \in \textit{tset}(p, s, t)$ and $s \overset{c}{\leadsto} u$. From Lemma (D.5), $\textit{trace}(\textit{tset}(p - \{c\}, u, t), u, t)$ iff $\textit{trace}(\textit{tset}(p, s, t) - \{c\}, u, t)$. The conclusion $\textit{trace}(\textit{tset}(p, s, t), s, t)$ follows from the definition of *trace*.

($\Leftarrow$): By Lemma (D.4) it follows that $\textit{tset}(p, s, t) \subseteq p$ and the conclusion $\textit{trace}(p, s, t)$ is straightforward by Lemma (D.6). □

Theorem (D.2)

The proof uses the following Lemma.

Lemma D.9 *For $p \in \mathcal{P}$, $c \in \mathcal{L}$ and $s, u, t \in State$,*

$$\frac{\textit{trace}(p, s, u) \quad c \in p \quad u \overset{c}{\leadsto} t \quad c \notin \textit{tset}(p, s, u)}{\textit{trace}(p, s, t)}$$

Proof.

By induction on $\textit{trace}(p, s, u)$.

Case $c_1 \in p$ and $s \overset{c_1}{\leadsto} u$: If $c_1 = c$ then $c \in \mathit{tset}(p, s, u)$, contradicting the assumptions. Assume $c_1 \neq c$. It follows that $c \in p - \{c_1\}$ and $u \overset{c}{\leadsto} t$, there is a trace, $\mathit{trace}(p - \{c_1\}, u, t)$. Since $s \overset{c_1}{\leadsto} u$, there is also a trace $\mathit{trace}(p, s, t)$.

Case $\exists u', c_1 : c_1 \in p \wedge s \overset{c_1}{\leadsto} u' \wedge \mathit{trace}(p - \{c_1\}, u', u)$ and the property holds for $\mathit{trace}(p - \{c_1\}, u', u)$: The inductive hypothesis has assumptions $c \notin \mathit{tset}(p - \{c_1\}, u', u)$, and $c \in p - \{c_1\}$ and conclusion $\mathit{trace}(p - \{c_1\}, u', t)$. For the first assumption, note that if $c \in \mathit{tset}(p - \{c_1\}, u', u)$ then $c \in \mathit{tset}(p, u', u)$ and therefore $c \in \mathit{tset}(p, s, u)$, contradicting the assumptions. For the second, if $c = c_1$ then $c \in \mathit{tset}(p, s, u)$ which is also a contradiction. Therefore, the conclusion of the inductive hypothesis holds and $\mathit{trace}(p, s, t)$ follows from $s \overset{c_1}{\leadsto} u'$, $\mathit{trace}(p - \{c_1\}, u', t)$ and the definition of *trace*. □

The *leads-to* relation can be defined in terms of the trace relations: if state s leads to state t through program p, then it does so by the transitive closure of *mtrace* to an intermediate state u followed by a trace to t.

Theorem D.2 *For program* p *and states* s, t*,*

$$\frac{s \overset{p}{\leadsto} t}{\mathit{trace}(p, s, t) \vee (\exists u : \mathit{mtrace}^{+}(p, s, u) \wedge \mathit{trace}(p, u, t))}$$

Proof.

By right induction on $s \overset{p}{\leadsto} t$.

Case $c \in p$ and $s \overset{c}{\leadsto} t$: $\mathit{trace}(p, s, t)$ follows from the definition of *trace*.

Case $c \in p$, $s \overset{p}{\leadsto} u$, $u \overset{c}{\leadsto} t$ and either $\mathit{trace}(p, s, u)$ or $\exists u' : \mathit{mtrace}^{+}(p, s, u) \wedge \mathit{trace}(p, u, t)$:

Assume $\mathit{trace}(p, s, u)$, the proof is similar for the case $\exists u' : \mathit{mtrace}^{+}(p, s, u') \wedge \mathit{trace}(p, u', u)$.

Assume that $c \in \mathit{tset}(p, s, u)$, then $\mathit{mtrace}(p, s, u)$ and $\mathit{trace}(p, u, t)$ completing the proof. Assume that $c \notin \mathit{tset}(p, s, u)$ then by Lemma (D.9), $\mathit{trace}(p, s, t)$ and the proof is completed. □

Corollary D.1 *For program* p *and states* s, t*,*

$$\frac{s \overset{p}{\leadsto} t \quad \mathit{final?}(p)(t)}{\mathit{mtrace}^{+}(p, s, t)}$$

Proof. From Theorem (D.2) and the assumption $s \overset{p}{\leadsto} t$, there are two cases, assume $\mathit{trace}(p, s, t)$. From $\mathit{final?}(p)(t) \Rightarrow \mathit{final?}(p - \mathit{tset}(p, s, t))(t)$ and the definition of *mtrace*, it follows that the conclusion $\mathit{mtrace}(p, s, t)$ is *true*. The proof is similar for the case $(\exists u : \mathit{mtrace}^{+}(p, s, u) \wedge \mathit{trace}(p, u, t))$. □

D.4 Refinement

D.4.1 Theorem (4.5)

Proof. *Composition of commands*

Let $p' = p \uplus \{c_1; c_2\}$. By definition of $\sqsubseteq$, the property to be proved is

$$\frac{c_1, c_2 \in p \quad \forall s, t : s \overset{p'}{\leadsto} t}{\forall s, t : s \overset{p}{\leadsto} t}$$

By induction on $s \overset{p'}{\leadsto} t$.

Case $c \in p \uplus \{c_1; c_2\}$ and $s \overset{c}{\leadsto} t$:

By definition of $c_1; c_2$ and $\uplus$ ($\mathit{label}(c_1; c_2) = \mathit{label}(c_1)$), $c \in p - \{c_1\}$ or $c = c_1; c_2$. Assume $c \in p - \{c_1\}$. By Lemma (4.2), $s \overset{p}{\leadsto} t$ is immediate. Assume $c = c_1; c_2$. By Theorem (3.6) (and $s \overset{c}{\leadsto} t = \mathcal{I}(c)(s, t)$), either $s \overset{c}{\leadsto} t$ or there is a state u such that $s \overset{c_1}{\leadsto} u$ and $u \overset{c_2}{\leadsto} t$. In either case, since $c_1 \in p$ and $c_2 \in p$, $s \overset{p}{\leadsto} t$ is straightforward from the definition of *leads-to*.

Case $s \overset{p'}{\leadsto} u$ and $u \overset{p'}{\leadsto} t$ for $u \in \mathit{State}$ and assume that the property holds between states s and u and between states u and t: Since the property holds between s and u and between u and t, $s \overset{p}{\leadsto} u$ and $u \overset{p}{\leadsto} t$, the proof is immediate from the definition of transitive closure. □

Proof. *Refinement and subprograms*

The property to prove is $\forall s, t : s \overset{p_1}{\leadsto} t \Rightarrow s \overset{p}{\leadsto} t$. If $s \overset{p_1}{\leadsto} t$, then $s \overset{p_2}{\leadsto} t$ by the assumption and by definition of refinement. Since $p_2 \subseteq p$, $s \overset{p}{\leadsto} t$ follows from Lemma (4.2). □

Proof. *Programs and abstraction*

By definition of $\sqsubseteq$, the property to be proved is

$$\frac{\forall s, t : s \overset{p_1}{\leadsto} t \Rightarrow s \overset{p_2}{\leadsto} t}{\forall s, t : s \overset{p_1 \uplus p_2}{\leadsto} t \Rightarrow s \overset{p_2}{\leadsto} t}$$

By induction on $s \overset{p_1 \uplus p_2}{\leadsto} t$.

Case $c \in p_1 \uplus p_2$ and $s \overset{c}{\leadsto} t$:

By definition of $p_1 \uplus p_2$, $c \in p_1$ or $c \in p_2$.

Assume $c \in p_1$. By definition, $s \overset{p_1}{\leadsto} t$ and $s \overset{p_2}{\leadsto} t$ follows from the assumption that $p_1 \sqsubseteq p_2$. Assume $c \in p_2$, the proof is immediate from definition of $s \overset{p_2}{\leadsto} t$.

Case $s \overset{p_1 \uplus p_2}{\leadsto} u$ and $u \overset{p \uplus p_2}{\leadsto} t$ for $u \in \mathit{State}$ and assume that the property holds between states s and u and between states u and t: Since the property holds between s and u and between u and t, $s \overset{p_2}{\leadsto} u$ and $u \overset{p_2}{\leadsto} t$, the proof is immediate from the definition of transitive closure. □

D.5 Control Flow Properties

D.5.1 Theorem (4.6)

Proof. *Composition (Left).*

The proof is by induction on c_1.

Case $c_1 = (:= (al, l_1))$: By definition, $c_1; c_2$ is **if equal**$(pc, \mathit{label}(c_2))$ **then** c' **else** c_1 where c' is the composition of c_1 and c_2. From the definition of $\longmapsto$, since c_1 occurs in $c_1; c_2$ and $c_1 \longmapsto c_3$, it follows that $c_1; c_2 \longmapsto c_3$.

Case $c_1 = l : c$ and the property is *true* for c: By definition of $\longmapsto$, $c_1 \longmapsto c_3$ iff $c' \longmapsto c_3$. By definition of composition, $c_1; c_2 = l : (c; c_2)$ and the property is immediate from the inductive hypothesis and the definition of $\longmapsto$.

Case $c_1 =$ **if** b **then** c_t **else** c_f and the property is *true* for c_t and for c_f: By definition of $\longmapsto$, $c_1 \longmapsto c_3$ iff $c_t \longmapsto c_3 \vee c_f \longmapsto c_3$. By definition of composition, $c_1; c_2$ is the command **if** b **then** $c_t; c_f$ **else** $c_f; c_2$ and the property is immediate from the inductive hypothesis and the definition of $\longmapsto$. □

Proof. *Composition (Right).*

Immediate from definition of composition and Corollary (4.2). □

The proof for the reverse of composition is in two steps: the first when c_1 is an assignment command, the second when c_1 is a labelled or conditional command.

Proof. *Reverse (assignment command)*

Let $c_1 = (:= (al, l))$. The proof is by induction on c_2.

Case $c_2 = (:= (bl, l_2))$: By definition, the composition of $c_1; c_2$ is $:= ((pc, l) \cdot al \oplus ((pc, l_2 \cdot bl) \triangleleft al), l_2 \triangleleft (pc, l) \cdot al)$. The successor expression of $c_1; c_2$ is $l_2 \triangleleft (pc, l) \cdot al$ and since $c_1; c_2 \longmapsto c_3$, there is a state t such that $\mathcal{I}_l(l_2 \triangleleft (pc, l) \cdot al)(t) = \mathit{label}(c_3)$ By substitution, this is equivalent to $\mathcal{I}_l(l_2)(\mathit{update}((pc, l) \cdot alt))$ and there is a state in which l_2 is equivalent to $\mathit{label}(c_3)$. Therefore $c_2 \longmapsto c_3$.

Case $c_2 = l : c$ and the property holds for c: By definition, $c_1; c_2$ is the conditional command **if equal**(pc, l) **then** $c_1; c$ **else** c_1. Since $c_1; c_2 \longmapsto c_3$, either $c_1 \longmapsto c_3$ (and the proof is complete) or $c_1; c \longmapsto c_3$ and either $c_1 \longmapsto c_3$ or $c \longmapsto c_3$. From the definition of *reaches*, if $c \longmapsto c_3$ then $l : c \longmapsto c_3$ is also *true* and the proof is complete.

Case $c_2 =$ **if** b **then** c_t **else** c_f and the property holds for c_t and c_f: By definition, if $c_1; c_2 \longmapsto c_3$ then either $c_1; c_t \longmapsto c_3$ or $c_1; c_f \longmapsto c_3$. In either case, the proof follows from the inductive hypothesis. □

Proof. *Reverse (labeled and conditional commands*

Case $c_1 = l : c$ and the assumption holds for c: By definition, $(c_1; c_2) = l : (c; c_2)$ and $c; c_2 \longmapsto c_3$. From the inductive hypothesis either $c_2 \longmapsto c_3$ or $c \longmapsto c_3$. The proof follows from the definition of *reaches* for a labelled command.

Case $c_1 =$ **if** b **then** c_t **else** c_f and the property holds for c_t and c_f: By definition, $c_1; c_2$ is **if** b **then** $c_t; c_2$ **else** $c_f; c_2$ and either $c_t; c_2 \longmapsto c_3$ or $c_f; c_2 \longmapsto c_3$. In both cases the proof follows from the inductive hypothesis. □

D.5.2 Theorem (4.8)

Proof. *Immediate successors*

Let $c_1 = l : c$. The proof is by induction on c. The cases when c is a labeled or a conditional command are straightforward from the inductive hypothesis. Let $c = (:= (al, l))$. By definition of $\mathcal{I}(c)(s,t)$, $t = \mathit{update}((pc, l) \cdot al)$ and since $\mathit{enabled}(c_2)(t)$, $\mathit{label}(c_2) = \mathcal{I}_l(pc)(t)$. From definition of *update*, $\mathcal{I}_l(pc)(t) = \mathcal{I}_l(l)(t)$ and there is a state such $\mathcal{I}_l(l)(t) = \mathit{label}(c_2)$. □

The proof of the second part of the Theorem uses the following property of the *leads-to* relation.

Lemma D.10 *For* $p \in \mathcal{P}$, $s, t \in$ *State,*

$$\frac{s \overset{p}{\leadsto} t}{\exists (u : \mathit{State}), (c : \mathcal{L}) : c \in p \land s \overset{c}{\leadsto} u}$$

Proof.

Straightforward, by induction on $s \overset{p}{\leadsto} t$. □

Proof. *Reaches through a program*

By induction on $s \overset{p}{\leadsto} t$.

Case $c \in p$ and $s \overset{c}{\leadsto} t$: The proof is similar to that for the immediate successor.

Case $s \overset{p}{\leadsto} u$ and $u \overset{p}{\leadsto} t$ and the property holds between s and u and between u and t: From Lemma (D.10) and $u \overset{p}{\leadsto} t$, there is a $c \in p$ and $u' \in$ *State* such that $u \overset{c}{\leadsto} u'$. Command c is enabled in u therefore, from the inductive hypothesis (between s and u), $c_1 \overset{p}{\longmapsto} c$. Also from the inductive hypothesis (between u and t), $c \overset{p}{\longmapsto} c_2$. The conclusion $c_1 \overset{p}{\longmapsto} c_2$ follows immediately from the definition of *reaches*. □

D.6 Paths

A path is defined inductively as establishing the relation *reaches* through a set. If for commands c_1, c_2 of a set a, the relation $c_1 \stackrel{a}{\longmapsto} c_2$ holds, then c_1 is said to reach c_2 in the set a and c_1 and c_2 are the end-points of the path. For the *reaches* relation, both the end-points and all intermediate commands must be contained in the set a. For a path through set a, only the intermediate commands must be contained in a.

Definition D.1 *Paths*

For commands c_1, c_2 and set a, the relation *path?* has type

$$(\mathit{Set}[\mathcal{L}], \mathcal{L}, \mathcal{L}) \to \mathit{boolean}$$

and is defined by induction.

$$\frac{c_1 \longmapsto c_2}{\mathit{path?}(a, c_1, c_2)} \qquad \frac{c \in a - \{c_1\} \quad c_1 \longmapsto c \quad \mathit{path?}(a - \{c_1\}, c, c_2)}{\mathit{path?}(a, c_1, c_2)}$$

□

The definition of a path is similar to that of the reaches relation with the exception that the path is through the set a while the *reaches* relation is in the set a. The relation *path?* holds even if either c_1 or c_2 is not a member of a. However, every intermediate point must be taken from a. The relation *path?* defines a path since each intermediate command is use at most once.

Lemma D.11 *For commands $c, c_1, c_2 \in \mathcal{L}$ and sets $a, b\mathit{Set}(\mathcal{L})$,*

$$\frac{\mathit{path?}(a, c_1, c_2)}{\mathit{path?}(a \cup b, c_1, c_2)}$$

Proof. By induction on *path?*(a, c_1, c_2):

Base case $c_1 \longmapsto c_2$: *path?*$(a \cup b, c_1, c_2)$ follows immediately from definition.

Inductive case, $c \in a - \{c_1\}$, $c_1 \longmapsto c$ and *path?*$(a - \{c_1\}, c, c_2)$ and hypothesis $\forall b' : \mathit{path?}((a - \{c_1\}) \cup b', c, c_2)$: From the hypothesis and $(a - \{c_1\}) \cup b - \{c_1\} = (a \cup b) - \{c_1\}$, *path?*$((a \cup b) - \{c_1\}, c, c_2)$ and *path?*$(a \cup b, c_1, c_2)$ follows from the assumptions and definition. □

For commands c_1, c_2 and set a, if *path?*(a, c_1, c_2) then any command $c \in a$ is either unnecessary for the path from c_1 to c_2 or c_1 is unnecessary for the path from c to c_2.

Lemma D.12 *For $c, c_1, c_2 \in \mathcal{L}$ and set $a \in \mathit{Set}(\mathcal{L})$,*

$$\frac{\mathit{path?}(a, c_1, c_2) \quad c \in a}{\mathit{path?}(a - \{c\}, c_1, c_2) \vee \mathit{path?}(a - \{c_1\}, c, c_2)}$$

Proof. By induction on *path?*(a, c_1, c_2):

Base case $c_1 \longmapsto c_2$: *path?*$(a - \{c\}, c_1, c_2)$ follows immediately from definition.

Inductive case $c \in a - \{c_1\}$, $c_1 \longmapsto c$ and *path?*$(a - \{c_1\}, c, c_2)$. Hypothesis:

$$\forall c' : c' \in a - \{c\} \Rightarrow \textit{path?}(a - \{c_1\} - \{c'\}, c_1, c_2) \vee \textit{path?}(a - \{c_1\} - \{c\}, c', c_2)$$

Case $c = c_2$: *path?*$(a - \{c\}, c_1, c_2)$ follows immediately from assumptions and definition.

Assume $c \neq c_2$, the proof is by cases.

Case $c_2 = c_1$: Let $c' = c$ in the inductive hypothesis. From the assumptions and $c \neq c_2$, $c \notin (a - \{c_1\})$ and $c_1 \longmapsto c$, the hypothesis leads to *path?*$(a - \{c_1\} - \{c\}, c_1, c_2)$ From Lemma (D.11) this is *path?*$((a - \{c_1\} - \{c\}) \cup, c, c_2)$ and from $(a - \{c_1\} - \{c\}) \cup (a - \{c_1\}) = (a - \{c_1\})$ this is *path?*$(a - \{c_1\}, c, c_2)$ completing the proof for this case.

Case $c_2 \neq c_1$: Let $c' = c_2$ in the hypothesis. From the assumptions and $c_2 \neq c_1$, $c_2 \notin (a - \{c_1\})$ and the hypothesis leads to *path?*$(a - \{c_1\} - \{c_2\}, c_1, c_2)$. The proof is similar for the case when $c_2 = c_1$ and leads to *path?*$(a - \{c_1\}, c, c_2)$ completing the proof. □

If there is a path from command c_1 to command c_2 through a set a then c_1 reaches c_2 in the set $a \cup \{c_1, c_2\}$.

Lemma D.13 *For commands* $c_1, c_2 \in \mathcal{L}$ *and set* $a \in \textit{Set}(\mathcal{L})$*,*

$$\frac{\textit{path?}(a, c_1, c_2)}{c_1 \overset{a \cup \{c_1, c_2\}}{\longmapsto} c_2}$$

Proof. Straightforward by induction on *path?*(a, c_1, c_2). □

If command c_1 reaches command c_2 through a set a then there is a path from c_1 to c_2 through set a.

Lemma D.14 *For commands* $c_1, c_2 \in \mathcal{L}$ *and set* $a \in \textit{Set}(\mathcal{L})$*,*

$$\frac{c_1 \overset{a}{\longmapsto} c_2}{\textit{path?}(a, c_1, c_2)}$$

Note that the *reaches* relation through a set is equivalent to the transitive closure of $\lambda(a : \textit{Set}(\mathcal{L}), c_1, c_2 : \mathcal{L}) : c_1 \in a \wedge c_2 \in a \wedge c_1 \longmapsto c_2$.

Proof. By left induction of $c_1 \overset{a}{\longmapsto} c_2$.

Base case $c_1 \longmapsto c_2$: *path?*(a, c_1, c_2) follows immediately from definition.

Inductive case $c, c_1, c_2 \in a$, $c_1 \longmapsto c$ and $c \overset{a}{\longmapsto} c_2$ with hypothesis *path?*(a, c, c_2): Assume that $c \neq c_1$ since otherwise the conclusion follows immediately from the hypothesis. From

Lemma (D.12), $c_1 \in a$ and from the hypothesis, either *path?*$(a - \{c_1\}, c, c_2)$ or *path?*$(a - \{c\}, c_1, c_2)$.

Case path?$(a - \{c_1\}, c, c_2)$: From $c \in a - \{c_1\}$, $c_1 \longmapsto c$ and the definition of *path?*, the conclusion *path?*(a, c_1, c_2) follows from the assumptions.

Case path?$(a - \{c\}, c_1, c_2)$: The conclusion *path?*(a, c_1, c_2) follows from Lemma (D.11). □

Command c_1 reaches command c_2 through a set a, $c_1 \overset{a}{\longmapsto} c_2$, iff there is a path from c_1 to c_2. Either c_2 is an immediate successor of c_1 or there is an immediate successor of c_1 which is distinct from c_1 and which reaches c_2 in $a - \{c\}$.

Theorem D.3 *Paths*

For commands c_1, c_2, c *and set* a*,*

$$\frac{c_1 \neq c_2 \quad c_1 \overset{a}{\longmapsto} c_2}{c_1 \longmapsto c_2 \vee (\exists c \in (a - \{c_1\}) : c_1 \longmapsto c \wedge c \overset{a-\{c_1\}}{\longmapsto} c_2)}$$

Proof.

From the assumption and Lemma (D.14), it follows that *path?*(a, c_1, c_2) is *true*. From the definition of *path?*, either $c_1 \longmapsto c_2$, and the proof is complete, or there is a $c \in a - \{c_1\}$ such that $c_1 \longmapsto c$ and *path?*$(a - \{c\}, c, c_2)$. From Lemma (D.13), this is enough to establish $c \overset{a'}{\longmapsto} c_2$ where $a' = (a - \{c_1) \cup \{c, c_2\}$. From the definition of *reaches*, it follows that $c_2 \in a$ and, since $c_2 \neq c_1$, $c_2 \in a - \{c_1\}$. Since c is also in $a - \{c_1\}$, it can be established that $a' = a$ and therefore $c \overset{a-\{c_1\}}{\longmapsto} c_2$. □

When a is a program, Theorem (D.3) can be used in a proof by strong induction on a. If there is an immediate successor, c, of c_1 which reaches c_2 in $a - \{c_1\}$ then there is a flow-graph constructed from $a - \{c\}$ beginning at c. This is a proper subset of the flow-graph constructed from a to which the inductive hypothesis applies.

D.7 Loops and *mtrace*

There is a loop in a program p if during an execution of p, a command $c \in p$ is selected for execution more than once. For each loop in the program, there is a command which begins the loop. This command, the head of the loop, can be determined by a property of a maximal trace through a program. If there is a maximal trace through a program p from state s to state t, then any command $c \in p$ enabled in t is the head of a loop in p. Moreover, either c is enabled in s or there is an intermediate state u and a subset of p $p_1 \subseteq p$ such that there is a trace from s to u in p_1 and a trace from u to t in $(p - \{p_1\}) \cup \{c\}$.

Theorem D.4 *For $p \in \mathcal{P}$, $c_1, c_2 \in p$, $s, t, u \in$ State and $a \in \mathcal{P}$,*

$$\frac{\mathit{trace}(p,s,t) \quad \mathit{enabled}(c_1)(s) \quad \mathit{enabled}(c_2)(t) \quad c_2 \in \mathit{tset}(p,s,t) \quad c_1 \neq c_2}{(\exists a, u : a \subseteq p \wedge \mathit{enabled}(c_2)(u) \wedge \mathit{trace}(a,s,u) \wedge \mathit{trace}((p-a) \cup \{c_2\}, u, t))}$$

Proof.

By induction on *trace*(p, s, t).

Case $s \overset{c_1}{\leadsto} t$: Since $c_1 \neq c_2$, there are two cases of *tset*(p, s, t). Assume there is a state u such that $s \overset{c_2}{\leadsto} u$. Since c_2 is enabled in s, it shares a label with c_1 and, from the definition of programs, $c_1 = c_2$ which is a contradiction.

Assume there is a command $c' \in p$ and state u such that $s \overset{c'}{\leadsto} u$ and $c_2 \in$ *tset*$(p - \{c'\}, u, t)$. Since c' is enabled in state s, $c' \in p$ and $c_1 \in p$, $c' = c_1$. Since $s \overset{c_1}{\leadsto} t$ and $s \overset{c_1}{\leadsto} u$, by Lemma (3.3) it follows that $u = t$. Therefore $c_2 \in$ *tset*$(p - \{c_1\}, t, t)$, *tset*$(p - \{c_1\}, t, t)$ is not empty, and, by Lemma (D.7), there is a trace *trace*$(p - \{c_1\}, t, t)$.

Let $a = \{c_1\}$ and $u = t$. It follows that $a \subseteq p$, *enabled*$(c_2)(t)$ and *trace*(a, s, u) (from $s \overset{c_1}{\leadsto} t$). Also $(p - a) \cup \{c_2\} = p - \{c_1\}$ and *trace*$((p - a) \cup \{c_2\}, u, t)$ is *true*, completing the proof for this case.

Case $\exists(u' \in$ *State*$)$: $s \overset{c_1}{\leadsto} u' \wedge$ *trace*$(p - \{c_1\}, u', t)$ with the assumption that the property is *true* for *trace*$(p - \{c_1\}, u', t)$: Since there is a trace from u' to t through $p - \{c_1\}$, there is a command $c' \in p - \{c_1\}$ enabled in u'. If $c_2 = c'$ then the proof is as for the previous case with $a = \{c_1\}$ and $u = u'$. Assume $c_2 \neq c'$. Since $c_2 \in$ (*tset*$(p, s, t) - \{c_1\}$), by Lemma (D.5), $c_2 \in$ *tset*$(p - \{c_1\}, u', t)$ and from the assumptions, *enabled*$(c_2)(t)$. Also since *enabled*$(c')(u')$ and *trace*$(p - \{c_1\}, s, u')$, the assumptions of the inductive hypothesis are satisfied.

From the inductive hypothesis, there is a set $a_1 \subseteq p - \{c_1\}$ and state u_1 such that *trace*(a_1, u', u_1), *enabled*$(c_2)(u')$ and *trace*$((p - \{c_1\} - a_1) \cup \{c_2\}, u_1, t)$. Let a be $a_1 \cup \{c_1\}$ and $u = u_1$. *trace*(a, s, u) follows from $s \overset{c_1}{\leadsto} u'$ and *trace*$(a - \{c_1\}, u', u)$. *enabled*$(c_2)(u)$ is immediate as is $a \subseteq p$ and *trace*$((p - a) \cup \{c_2\}, u, t)$ follows from $(a_1 - p - \{c_1\}) = ((a_1 \cup \{c_1\}) - p)$, completing the proof. □

The property of Theorem (D.4) is used in the proofs for transformation T_2 to show that a command enabled after a maximal trace through a region is the head of a loop (a member of *lpheads*). The commands beginning a loop in a region are determined by the predicate *lphead?* which is defined in terms of the *reaches* relation. The property of Theorem (D.4) is re-stated in terms of *reaches* in two steps. The first step (Theorem D.5 below) describes the property in terms of the *leads-to* relation between state and the second step describes this property in terms of *reaches* (Theorem D.6 below).

Theorem D.5 *For $p, p_1 \in \mathcal{P}$, $c \in \mathcal{L}$ and $s, t, u \in$ State,*

$$\frac{\mathit{mtrace}(p,s,t) \quad \neg \mathit{final?}(p)(t)}{\begin{array}{l}(\exists c, p_1, u : p_1 \subseteq p \wedge c \in p \wedge \mathit{enabled}(c)(u) \wedge \mathit{enabled}(c)(t) \\ \quad \wedge\, s \overset{p_1}{\leadsto} u \wedge u \overset{\{c\} \cup (p - p_1)}{\leadsto} t) \\ \vee (\exists c \in p : \mathit{enabled}(c)(s) \wedge \mathit{enabled}(c)(t) \wedge s \overset{p}{\leadsto} t)\end{array}}$$

Proof.

From the definition of $\mathit{mtrace}(p,s,t)$, $\mathit{trace}(p,s,t)$ and $\mathit{final?}(p - \mathit{tset}(p,s,t))$ are both *true*. From the definition of *final?*, and the assumption $\neg\mathit{final?}(p)(t)$ there is a command $c' \in p$ such that $\mathit{enabled}(c')(t)$. if $c' \notin \mathit{tset}(p,s,t)$ then $c' \in p - \mathit{tset}(p,s,t)$, contradicting the assumption $\mathit{final?}(p - \mathit{tset}(p,s,t))(t)$.

Assume $c' \in \mathit{tset}(p,s,t)$. From the definition of *trace* and the assumption, $\mathit{trace}(p,s,t)$ there is a command $c_1 \in p$ and $\mathit{enabled}(c_1)(s)$. By Theorem (D.4), either $c_1 = c'$ or there is a program $a \subseteq p$ and state u such that $\mathit{trace}(a,s,u)$, $\mathit{enabled}(c')(u)$ and $\mathit{trace}(p - a \cup \{c'\}, u, t)$. Assume $c_1 = c'$. From $\mathit{trace}(p,s,t)$ and Lemma (D.3), $s \overset{p}{\leadsto} t$ follows immediately. $\mathit{enabled}(c_1)(s)$ and $\mathit{enabled}(c_1)(t)$ follow from the assumptions, completing the proof for this case.

For the second case, from $\mathit{trace}(a,s,u)$ it follows that $s \overset{a}{\leadsto} u$ and from $\mathit{trace}(p - a \cup \{c'\}, u, t)$ it follows that $u \overset{p-a\cup\{c_1}{\leadsto} t$ (both by Lemma D.3). Command c' is enabled in s and t and therefore the proof is complete. □

Theorem D.6 *For* $p, p_1 \in \mathcal{P}$, $c \in \mathcal{L}$ *and* $s,t,u \in \mathit{State}$,

$$\frac{\mathit{mtrace}(p,s,t) \quad c \in p \quad \mathit{enabled}(c)(s) \quad \neg\mathit{final?}(p)(t)}{\begin{array}{r} c \overset{p}{\longmapsto} c \vee (\exists c_1, p_1, u : p_1 \subseteq p \wedge c \in p \wedge \mathit{enabled}(c_1)(t) \\ \wedge\, c \overset{p_1}{\longmapsto} c_1 \wedge c_1 \overset{\{c_1\}\cup(p-p_1)}{\longmapsto} c_1) \end{array}}$$

Proof.

The proof is straightforward from Theorem (D.6) and Theorem (4.8). Note that because c is enabled in s, $\mathit{enabled}(c)(s)$, for every command $c' \in p$ such that $\mathit{enabled}(c')(s)$, $c' = c$. □

D.8 Regions

Theorem (4.9)

The proof is by induction on a program. Let Φ_r be the property of Theorem (4.9) and Φ_p be the property of the strong induction scheme on programs.

Proof.

By strong program induction. Let Φ_p be $\lambda(p : \mathcal{P}) : (\forall(r : \mathcal{R}) : \mathit{body}(r) \subseteq p \Rightarrow \Phi_r(r))$. Let $p = \mathit{body}(r)$, the conclusion of Theorem (4.9), $\forall r : \Phi_r(r)$, follows from the conclusion of the program induction scheme, $\forall p : \Phi_p(p)$. The proof is therefore that the assumption of the program induction scheme is satisfied by the assumption of Theorem (4.9).

The assumption of the program induction scheme is $(\forall p : (\forall p' : p' \subset p \Rightarrow \Phi_p(p')) \Rightarrow \Phi_p(p))$. Let p be any program, to prove $\Phi_p(p)$ it must be shown that for any region r such that $\mathit{body}(r) \subseteq p$, $\Phi_r(r)$ is *true*.

From the assumption of Theorem (4.9), if, for every proper sub-region $r' \subset r$, $\Phi_r(r')$ is *true* then $\Phi_r(r)$ is also *true*. Let r' be a proper sub-region of r, $r' \subset r$. Because *body*$(r) \subseteq p$ and $r' \subset r$, it follows that *body*$(r') \subset p$. Therefore, from the assumption of the program induction scheme, $\Phi_p($*body*$(r'))$ is *true*. From the definition Φ_p and *body*$(r') \subseteq$ *body*(r'), it follows that $\Phi_r(r')$ is also *true*. The proof of $\Phi_r(r)$ therefore follows from the assumption of Theorem (4.9) and the proof of $\Phi_p(p)$ is complete. □

D.8.1 Corollary (4.5)

Proof.

The proof is in two steps.

1. *label*$(r_1) =$ *label*(r_3) and *body*$(r_1) \sqsubseteq$ *body*(r_3):
   From the definition of $\sqsubseteq$ between regions, *label*$(r_1) =$ *label*(r_2) and *label*$(r_2) =$ *label*(r_3). Also by definition, *body*$(r_1) \sqsubseteq$ *body*(r_2) and *body*$(r_2) \sqsubseteq$ *body*(r_3). By Theorem (4.3), *body*$(r_1) \sqsubseteq$ *body*(r_3).
2. $\forall s, t : s \overset{r_1}{\leadsto} t \wedge$ *final?*$(r_1)(t) \Rightarrow$ *final?*$(r_3)(t)$:
   By definition of $\sqsubseteq$ between programs (*body*$(r_1) \sqsubseteq$ *body*(r_2)), $s \overset{r_2}{\leadsto} t$ and *final?*$(r_2)(t)$. Since $r_2 \sqsubseteq r_3$, $s \overset{r_3}{\leadsto} t$ and *final?*$(r_3)(t)$ follows immediately.

□

D.9 Composition Over a Set

Lemma D.15 *For $c \in \mathcal{L}$ and set $a \in$ FiniteSet$(\mathcal{L})$,*

$$\textit{label}(c; a) = \textit{label}(c)$$

Proof.

Straightforward from strong finite induction on a and from the definition of composition. □

If command c ends in state t and there is no command $c' \in A$ enabled in t then the command $(c; A)$ also ends in t.

Theorem D.7 *For commands $c, c_1 \in \mathcal{L}$, set $a \in$ FiniteSet$(\mathcal{L})$ and states $s, t, u \in$ State,*

$$\frac{\mathcal{I}(c)(s,t) \quad \forall(c_1 \in a) : \neg(\exists u : \mathcal{I}(c)(s,u) \wedge \textit{enabled}(c_1)(u))}{\mathcal{I}(c; a)(s,t)}$$

Proof.

By strong finite induction on a. If $a = \{\}$, then by definition, $\mathcal{I}(c; a)(s,t) = \mathcal{I}(c)(s,t)$ and the proof is complete.

Assume $a \neq \{\}$ and that the property holds for all $a' \subset a$. Let $c' = \epsilon a$, from the epsilon axiom, $c' \in a$ and $a - \{c'\} \subset a$. The property to prove is that from $\mathcal{I}(c)(s,t)$ it follows that $\mathcal{I}((c; a - \{c'\}); c')(s,t)$ (definition of composition over a set).

Case for all $c_2 \in a - \{c'\}$, there is no $u_2 \in$ *State* such that $\mathcal{I}(c)(s,u_2)$ and *enabled*$(c_2)(u_2)$: From the inductive hypothesis, $\mathcal{I}(c; a - \{c'\})(s,t)$. From the assumptions, *pc* $\not\equiv_t$ *label*(c') since otherwise there would command $c' \in a$ is enabled in t, *enabled*$(c')(t)$ contradicting the assumptions. From Theorem (3.5), the composition of $c; a-\{c'\}$ and c' is equivalent to $c; a-\{c'\}$ and $\mathcal{I}((c; a - \{c'\}); c')(s,t)$ follows from $\mathcal{I}(c; a - \{c'\})(s,t)$.

Case there is a $c_2 \in a-\{c'\}$ and $u_2 \in$ *State* such that $\mathcal{I}(c)(s,u_2)$ and *enabled*$(c_2)(u_2)$: It follows that there is a command in a, c_2, and a state, u_2, such that $\mathcal{I}(c)(s,u_2)$ and *enabled*$(c_2)(u_2)$, contradicting the assumptions. $\square$

Theorem (D.7) describes the behaviour when no command in the set a is enabled. This is extended by Theorem (D.8) below for the case when a single command of a program p is selected. If command c begins in state s and ends in state u, there is a command $c' \in p$ which begins in state u and ends in state t and no other command in a is enabled in t then the command $c; a$ begins in state s and ends in state t.

Theorem D.8 *For commands $c, c_1, c_2 \in \mathcal{L}$, program $p \in \mathcal{P}$ and states s, t, u,*

$$\frac{(\exists c_1 \in p : \mathcal{I}(c)(s,u) \wedge \mathcal{I}(c_1)(u,t)) \quad (\forall (c_2 \in p) : \neg \mathit{enabled}(c_2)(t))}{\mathcal{I}(c;p)(s,t)}$$

Proof.

By strong finite induction on p. Note that every subset of p is also a program and that $p \neq \{\}$ since $c_1 \in p$.

Let $c' = \epsilon p$. By definition of composition over a set, the property to prove is that $\mathcal{I}((c; p - \{c'\}); c')(s,t)$.

Case $c' = c_1$: Assume for some $c_2 \in p - \{c_1\}$, there is a u' such that $\mathcal{I}(c)(s,u')$ and that *enabled*$(c_2)(u')$. From the assumptions $\mathcal{I}(c)(s,u)$ and from Lemma (3.3) $u = u'$. Since c_1 is also enabled in u, $c_1, c_2 \in p$ and p is a program it follows that $c_1 = c_2$ contradicting the assumption $c_2 \in p - \{c_1\}$.

Assume there is no state u' such that $\mathcal{I}(c)(s,u')$ and *enabled*$(c_2)(u')$, for some $c_2 \in p - \{c_1\}$. From Theorem (D.7), $\mathcal{I}(c)(s,u)$ leads to $\mathcal{I}(c; p - \{c_1\})(s,u)$. From Theorem (3.4), $\mathcal{I}(c; p - \{c_1\})(s,u)$ and $\mathcal{I}(c_1)(u,t)$ it follows that $\mathcal{I}((c; p - \{c_1\}); c_1)(s,t)$ is *true*.

Case $c' \neq c_1$: From the assumptions, there is a command $c_1 \in p - \{c'\}$ such that $\mathcal{I}(c_1)(u,t)$ and for all $c_2 \in p - \{c'\}$, $\neg$*enabled*$(c_2)(t)$. Therefore, from the inductive hypothesis, $\mathcal{I}(c; p -$

$\{c'\})(s,t)$ is *true*. From the fact that $c' \in p$, the assumption that no command of p is enabled in t and Theorem (3.5), the conclusion $\mathcal{I}((c; p - \{c'\}); c')(s,t)$ follows immediately. □

For command $c \in \mathcal{L}$ and set $a \in \mathit{FiniteSet}(\mathcal{L})$, if $c; a$ reaches command c_1 then there is a command $c_2 \in a \cup \{c\}$ such that $c_2 \longmapsto c_1$.

Theorem D.9 *For* $c, c_1, c_2 \in \mathcal{L}$ *and* $a \in \mathit{FiniteSet}(\mathcal{L})$*,*

$$\frac{(c; a) \longmapsto c_1}{\exists c_2 : c_2 \in a \cup \{c\} \wedge c_2 \longmapsto c_1}$$

Proof.

By strong finite induction on a.

Case $a = \{\}$: By definition, $(c; a) = c$ therefore $c \longmapsto c_1$ and $c \in a \cup \{c\}$ are immediate.

Case $a \neq \{\}$: Let $c' = \epsilon a$. By definition of composition, $(c; a - \{c'\}); c' \longmapsto c_1$. From Theorem (4.6), either $(c; a - \{c'\}) \longmapsto c_1$ or $c' \longmapsto c_1$. Assume $c' \longmapsto c_1$. Since $\epsilon a \in a$ the conclusion follows immediately. Assume $(c; a - \{c'\}) \longmapsto c_1$. From the inductive hypothesis, there is a command $c_3 \in (a - \{c'\}) \cup \{c\}$ such that $c_3 \longmapsto c_1$. The conclusion follows immediately from $c_3 \in a \cup \{c\}$. □

For command $c \in \mathcal{L}$, program $p \in \mathcal{P}$ and states $s, t \in \mathit{State}$, if $\mathcal{I}(c; p)(s,t)$ then there is a trace through $p \cup \{c\}$ from s to t.

Theorem D.10 *For* $c \in \mathcal{L}$*,* $p \in \mathcal{P}$ *and* $s, t \in \mathit{State}$*,*

$$\frac{c \notin p \quad \mathcal{I}(c; p)(s,t) \quad \mathit{program?}(p \cup \{c\})}{\mathit{trace}(p \cup \{c\}, s, t)}$$

Proof.

By strong finite induction on p. Note that every subset of $p \cup \{c\}$ is also a program.

Case $p = \{\}$: By definition, $\mathcal{I}(c; p)(s,t) = \mathcal{I}(c)(s,t)$ and $\mathit{trace}(p \cup \{c\}, s, t)$ follows from the definition of *trace*.

Case $p \neq \{\}$: Let $c' = \epsilon p$. From the assumptions, $\mathcal{I}((c; p - \{c'\}); c')(s,t)$ is *true*. From Theorem (3.5), either there is a state u such that $\mathcal{I}(c; p - \{c'\})(s,u)$ and $\mathcal{I}(c')(u,t)$ or $\mathcal{I}(c; p - \{c'\})(s,t)$ and $\mathit{pc} \not\equiv_t \mathit{label}(c')$.

Assume $\mathcal{I}(c; p - \{c'\})(s,t)$ and $\mathit{pc} \not\equiv_t \mathit{label}(c')$. From the assumption, $c \notin p$ and therefore $c \notin p - \{c'\}$. Since $p - \{c'\} \subset p$, the inductive hypothesis leads to $\mathit{trace}(p - \{c'\} \cup \{c\}, s, t)$. From Lemma (D.6) and $(p - \{c'\} \cup \{c\}) \subseteq (p \cup \{c\})$, it follows that $\mathit{trace}(p \cup \{c\}, s, t)$ is *true*.

Assume $\mathcal{I}(c; p - \{c'\})(s,u)$ and $\mathcal{I}(c')(u,t)$. From the inductive hypothesis, $\mathit{trace}(p - \{c'\} \cup \{c\}, s, u)$ is *true*. Since $c' \notin (p - \{c'\} \cup \{c\})$, and by Lemma (D.8), $\mathit{trace}(p \cup \{c\}, s, t)$ is also *true*, completing the proof. □

For any program p, command c and state s, t, if $\mathcal{I}(c; p)(s, t)$ then either $\mathcal{I}(c)(s, t)$ or there is a u such that $\mathcal{I}(c)(s, u)$ and *trace*(p, u, t).

Theorem D.11 *For $p \in \mathcal{P}$, $c \in \mathcal{L}$ and $s, t, u \in State$,*

$$\frac{\mathcal{I}(c; p)(s, t)}{\mathcal{I}(c, s, t) \vee (\exists u : \mathcal{I}(c, s, u) \wedge \mathit{trace}(p, u, t))}$$

Proof.

By strong finite induction on p.

Case $p = \{\}$: By definition $\mathcal{I}(c; p)(s, t) = \mathcal{I}(c)(s, t)$ and the proof is complete for this case.

Case $p \neq \{\}$: Let $c' = \epsilon p$. By definition, $\mathcal{I}(c; p)(s, t)$ is $\mathcal{I}((c; p - \{c'\}); c')(s, t)$. From Theorem (4.5) either there is a t_1 such that $\mathcal{I}(c; p - \{c'\})(s, t_1)$ and $\mathcal{I}(c')(t_1, t)$ or $\mathcal{I}(c; p - \{c'\})(s, t)$.

Assume $\mathcal{I}(c; p - \{c'\})(s, t)$. From the inductive hypothesis, either $\mathcal{I}(c)(s, t)$ (and the case is proved) or there is a u such that $\mathcal{I}(c)(s, u)$ and *trace*$(p - \{c'\}, u, t)$. From Lemma (D.6), and $(p - \{c'\} \subset p)$, it follows that *trace*(p, u, t) is *true* completing the proof for the case.

Assume a state $t_1 \in State$ such that $\mathcal{I}(c; p - \{c'\})(s, t_1)$ and $\mathcal{I}(c')(t_1, t)$. From the inductive hypothesis either $\mathcal{I}(c)(s, t_1)$ or there is a u such that $\mathcal{I}(c)(s, u)$ and *trace*$(p - \{c'\}, u, t_1)$. Assume $\mathcal{I}(c)(s, t_1)$, then from $\mathcal{I}(c')(t_1, t)$, $c' \in p$ and the definition of *trace*, *trace*(p, t_1, t) is *true* and the proof is completed for this case. Assume a state u such that $\mathcal{I}(c)(s, u)$ and *trace*$(p - \{c'\}, u, t_1)$. From $\mathcal{I}(c')(t_1, t)$, $c' \notin p - \{c'\}$ and Lemma (D.8), *trace*$(p - \{c'\} \cup \{c'\}, u, t)$. The conclusion that there is a state u such that $\mathcal{I}(c)(s, u)$ and *trace*(p, u, t) follows immediately. □

For program $p \in \mathcal{P}$, commands $c \notin p$, $c_1 \in p$ and states $s, t \in State$, if command c begins in state s to produce state t and c_1 halts in t then $(c; p)$ halts in state s.

Theorem D.12 *For $p \in \mathcal{P}$, $c, c_1 \in \mathcal{L}$ and $s, t, t_1, t_2 \in State$,*

$$\frac{c \notin p \quad c_1 \in p \quad \mathcal{I}(c)(s, t) \quad \mathit{enabled}(c_1)(t) \quad \forall t_1 : \neg\mathcal{I}(c_1)(t, t_1)}{\forall t_2 : \neg\mathcal{I}(c; p)(s, t_2)}$$

Proof.

By strong finite induction on p. Since $c_1 \in p$, $p \neq \{\}$ and there is only one case. Let $c' = \epsilon p$ and assume that there is a state $t_2 \in State$ such that $\mathcal{I}(c; p)(s, t_2)$. The proof is by showing that this assumption leads to a contradiction. From the definition of composition over a set, $\mathcal{I}(c; p)(s, t_2)$ is $\mathcal{I}((c; p - \{c'\}); c')(s, t_2)$.

Case $c' = c_1$: From Theorem (3.7) either there is a state u such that $\mathcal{I}(c; p - \{c'\})(s, u)$ and $\mathcal{I}(c')(u, t_2)$ or $\mathcal{I}(c; p - \{c'\})(s, t_2)$ and $\neg$*enabled*$(c')(t)$. From the assumptions, *enabled*$(c')(t)$ is *true* and there is only the case when $\mathcal{I}(c; p - \{c'\})(s, u)$ and $\mathcal{I}(c')(u, t_2)$. Since p is a program and *enabled*$(c_1)(t)$, there is no command $c_3 \in p - \{c_1\}$ such that *enabled*$(c_3)(t)$. From the assumption $\mathcal{I}(c)(s,t)$ and Theorem (D.7), $\mathcal{I}(c; p - \{c'\})(s,t)$ is *true*. From the assumption $\mathcal{I}(c; p - \{c'\})(s,u)$ and from Lemma (3.3), $u = t$ and therefore $\mathcal{I}(c')(t, t_2)$. This contradicts the assumption that there is no $t_1 \in \mathit{State}$ such $\mathcal{I}(c_1)(t, t_1)$ and completes the proof for this case.

Case $c' \neq c_1$: From Theorem (3.7) either there is a state u such that $\mathcal{I}(c; p - \{c'\})(s,u)$ and $\mathcal{I}(c')(u, t_2)$ or $\mathcal{I}(c; p - \{c'\})(s, t_2)$ and $\neg \mathit{enabled}(c')(t)$. For both cases, from $c_1 \in (p - \{c'\})$ and from the inductive hypothesis it follows that $\forall t_3 : \neg \mathcal{I}(c; p - \{c'\})(s, t_3)$ which is a contradiction.

□

D.10 Path Transformation

The properties of the path transformation T_1 are based on the behaviour of the commands constructed by composition over a set and the syntactic properties established by the transformation. The proof of the semantic properties of the transformation are based on basic properties of the command constructed by the transformation. Note that for region r, $\mathit{label}(T_1(r)) = \mathit{label}(\mathit{head}(r))$ and that for all $r' \in \mathit{rest}(r)$ there is a $c \in r$ such that $\mathit{label}(c) = \mathit{label}(r')$.

For any region r and command c, if $T_1(r)$ reaches c then there is a $c_1 \in r$ such that c_1 reaches c through $\mathit{body}(r) \cup \{c\}$.

Theorem D.13 *For* $r \in \mathcal{R}$, $c, c_1 \in \mathcal{L}$,

$$\frac{T_1(r) \longmapsto c}{\exists c_1 : c_1 \in \mathit{body}(r) \land c_1 \overset{\mathit{body}(r) \cup c}{\longmapsto} c}$$

Proof.

By strong induction on r.

Case unit?(r): By definition, $T_1(r) = \mathit{head}(r)$ and from the assumptions, $\mathit{head}(r) \longmapsto c$. The conclusion $\mathit{head}(r) \overset{\mathit{body}(r) \cup \{c\}}{\longmapsto} c$ follows from $\mathit{head}(r) \in r$ and the definition of the *reaches* relation through a set.

Case $\neg$*unit?*(r): By definition, $T_1(r) = (\mathit{head}(r); T_1(\mathit{rest}(r)))$. From Theorem (D.9), there is a command $c_2 \in T_1(\mathit{rest}(r)) \cup \{\mathit{head}(r)\}$ such that $c_2 \longmapsto c$. If $c_2 = \mathit{head}(r)$ then the proof is as before. Assume $c_2 \in T_1(\mathit{rest}(r))$, then there is a region $r' \in \mathit{rest}(r)$ such that $T_1(r') \longmapsto c$. Since $r' \in \mathit{rest}(r)$, $r' \subset r$ and from the inductive hypothesis, $\exists c_1 \in \mathit{body}(r') \land c_1 \overset{\mathit{body}(r') \cup \{c\}}{\longmapsto} c$. From $\mathit{body}(r') \subset \mathit{body}(r)$ and $c_1 \in \mathit{body}(r')$ it follows that $c_1 \in \mathit{body}(r)$. Also, from $(\mathit{body}(r') \cup \{c\}) \subseteq (\mathit{body}(r) \cup \{c\})$, and from Theorem (4.7), it follows that $c_1 \overset{\mathit{body}(r') \cup \{c\}}{\longmapsto} c$ is *true*. □

If the head of a single loop r is enabled in state s, there is maximal trace from state s to state t through the body of r and there is a command $c \in r$ enabled in t then c is the head of r.

Lemma D.16 *For $r \in \mathcal{R}$, $c \in \mathcal{L}$ and $s, t \in State$,*

$$\frac{\textit{single?}(r) \quad \textit{enabled}(\textit{head}(r))(s) \quad c \in r \quad \textit{enabled}(c)(t) \quad \textit{mtrace}(\textit{body}(r), s, t)}{c = \textit{head}(r)}$$

Proof.

From the assumption *enabled*$(c)(t)$, t is not final for *body*(r). From Theorem (D.5), if there is a maximal trace through *body*(r) from s to t, and t is not final, then there are two cases.

Assume there is a command $c' \in \textit{body}(r)$ such that *enabled*$(c')(s)$ and *enabled*$(c')(t)$. Since *head*$(r) \in$ *body*(r), *enabled*(*head*(r))(s) and *body*(r) is a program it follows that $c' = \textit{head}(r)$. Also since *enabled*$(c)(t)$ and $c \in \textit{body}(r)$, it also follows that $c' = c$ completing the proof for this case.

Assume there is a program $p \subseteq \textit{body}(r)$, a state u and a command $c' \in \textit{body}(r)$ such that *enabled*$(c')(u)$, *enabled*$(c')(t)$, $s \overset{p}{\leadsto} u$ and $u \overset{(\textit{body}(r)-p)\cup\{c'\}}{\leadsto} t$. From *enabled*$(c)(t)$ and from the assumption *enabled*$(c')(t)$, it follows that $c' = c$. If $c = \textit{head}(r)$ then the case is proved, therefore $c \neq \textit{head}(r)$ and, since *enabled*(*head*(r))(s), *head*$(r) \in p$ and *head*$(r) \notin (\textit{body}(r) - p \cup \{c\})$. Therefore there is a subset p' of $\textit{body}(r) - \{\textit{head}(r)\}$ such that $u \overset{p'}{\leadsto} t$. From Theorem (4.8), this means that $c \overset{p'}{\longmapsto} c$ is *true* and it follows that $c \overset{\textit{body}(r)-\{\textit{head}(r)\}}{\longmapsto} c$, contradicting the assumption *single?*(r). □

If the head of r is enabled in state s and there is a maximal trace from s to state t through r, then $\mathcal{I}(T_1(r))(s, t)$.

Theorem D.14 *For any region $r \in \mathcal{R}$ and states $s, t \in State$,*

$$\frac{\textit{single?}(r) \quad \textit{enabled}(\textit{head}(r))(s) \quad \textit{mtrace}(\textit{body}(r), s, t)}{\mathcal{I}(T_1(r))(s, t)}$$

Proof.

By strong region induction on r.

Case unit?(r): By definition, $\textit{body}(r) = \{\textit{head}(r)\}$ therefore *mtrace*(*body*(r))(s, t) is equivalent to $\mathcal{I}(\textit{head}(r))(s, t)$. By definition, $T_1(r) = \textit{head}(r)$ and the conclusion $\mathcal{I}(T_1(r))(s, t)$ is straightforward.

Case $\neg$*unit?*(r) with the property true for every proper sub-region of r:

By definition, $T_1(r) = (\textit{head}(r); T_1(\textit{rest}(r)))$. From the definition of *mtrace*, there is a trace from s to t, *trace*(*body*(r), s, t). By definition of *trace*, there are two cases. For the first case, assume there is a command $c \in \textit{body}(r)$ and $s \overset{c}{\leadsto} t$. Since c is enabled in s it follows that $c = \textit{head}(r)$. From the definition of *mtrace*(*body*(r), s, t), t is final for *body*(r) − *tset*(*body*(r), s, t). From Lemma (D.16) and the assumption *mtrace*(*body*(r), s, t), state t is also final for *body*(r) −

$\{head(r)\}$. Since no region $r' \in rest(r)$ shares a label with $head(r)$, it follows that there is no region $r' \in rest(r)$ which is enabled in t. Consequently there is no command $c' \in T_1(rest(r'))$ which is enabled in t. The conclusion $\mathcal{I}(head(r); T_1(rest(r)))(s,t)$ follows from Theorem (D.7).

For the second case of $trace(body(r), s, t)$, assume there is a $c \in body(r)$ and state u such that $s \overset{c}{\leadsto} u$ and $trace(body(r) - head(r), u, t)$. Since c is enabled in s it follows that $c = head(r)$. Because there is a trace from u to t, there is a command $c_1 \in body(r) - \{head(r)\}$ such that $enabled(c_1)(u)$. This command begins a sub-region $r' = region(label(c_1), body(r) - \{head(r)\})$ and, by definition, $r' \in rest(r)$. Since $head(r) \notin r'$, it follows that $loopfree?(r')$ and therefore $single?(r')$.

From the assumption $mtrace(body(r), s, t)$ and from Lemma (D.16), the only command of r which can be enabled in t is $head(r)$. Since $head(r) \notin r'$, there is a maximal trace through r' from u to t, $mtrace(body(r'), u, t)$. The head of r' is c_1 which is enabled in u and, since $r' \subset r$, the assumptions of the inductive hypothesis are satisfied. From the inductive hypothesis, it follows that $\mathcal{I}(T_1(r'))(u,t)$ is *true*.

Because only $head(r)$ can be enabled in t, there is no command $c' \in T_1(rest(r))$ such that $enabled(c')(t)$. From $\mathcal{I}(head(r), s, u)$, from $\mathcal{I}(T_1(r'))(u,t)$ and from Theorem (D.8) it follows that $\mathcal{I}(head(r); T_1(rest(r)))(s,t)$, completing the proof. □

For any region r and states s, t, if $\mathcal{I}(T_1(r))(s,t)$ then $s \overset{r}{\leadsto} t$. The proof uses the following property of the *leads-to* relation through a set.

Lemma D.17 *For any sets $a, b \in Set(\mathcal{L})$ and $s, t \in State$,*

$$\frac{s \overset{a}{\leadsto} t \quad (\forall (c : \mathcal{L}), (s_1, t_1 : State) : c \in a \wedge s_1 \overset{c}{\leadsto} t_1 \Rightarrow s_1 \overset{b}{\leadsto} t_1)}{s \overset{b}{\leadsto} t}$$

Proof. Straightforward by induction on $s \overset{a}{\leadsto} t$. □

Theorem D.15 *For $r \in \mathcal{R}$, $s, t \in State$,*

$$\frac{single?(r) \quad \mathcal{I}(T_1(r))(s,t)}{s \overset{r}{\leadsto} t}$$

Proof.

By strong region induction on r.

Case unit?(r): $\mathcal{I}(T_1(r))(s,t)$ iff $\mathcal{I}(head(r))(s,t)$, by definition, and $s \overset{r}{\leadsto} t$ is straightforward from $head(r) \in r$.

Case ¬unit?(r) and the property holds for every proper subregion of r: There is no command $c \in T_1(rest(r))$ such that $label(c) = label(head(r))$ and $rest(r)$ is a program, since for every $r' \in rest(r)$ there is a $c' \in r$ such that $r' = region(label(c'), body(r) - \{head(r)\})$. Therefore, for any regions $r_1, r_2 \in \mathit{rest}(r)$ such that $\mathit{label}(r_1) = \mathit{label}(r_2)$, the regions r_1 and r_2 are the same, $r_1 = r_2$. It follows that for any commands $c_1, c_2 \in T_1(\mathit{rest}(r))$ such that $\mathit{label}(c_1) = \mathit{label}(c_2)$, $c_1 = c_2$. Consequently $T_1(\mathit{rest}(r))$ is a program, $\mathit{program?}(T_1(\mathit{rest}(r)))$.

From Theorem (D.10), there is a trace $\mathit{trace}(T_1(\mathit{rest}(r)) \cup \{\mathit{head}(r)\}, s, t)$ and, by Lemma (D.3), $s \overset{T_1(\mathit{rest}(r)) \cup \{\mathit{head}(r)\}}{\leadsto} t$. Every region $r' \in \mathit{rest}(r)$ is loop-free, $\mathit{loopfree?}(r')$, and therefore a single loop, $\mathit{single?}(r')$. From the inductive hypothesis, for every $r' \in \mathit{rest}(r)$ and states s_1, t_1, $\mathcal{I}(T_1(r'))(s_1, t_1) \Rightarrow s_1 \overset{r'}{\leadsto} t_1$. Since $r' \subset r$, by Lemma (4.2), this leads to $s_1 \overset{r}{\leadsto} t_1$. $\mathcal{I}(\mathit{head}(r))(s_1, t_1) \Rightarrow s \overset{r}{\leadsto} t$ is immediate from the definitions. From Lemma (D.17) it follows that $s \overset{r}{\leadsto} t$. □

For any region r and states s, t, if $\mathcal{I}(T_1(r))(s, t)$ then there is a maximal trace from s to t through r. The proof of this property is by a number of steps to establish that t is final for $\mathit{body}(r) - \{\mathit{head}(r)\}$ and that there is a trace from s to t through r, $\mathit{trace}(\mathit{body}(r), s, t)$.

Lemma D.18 *For $r, r_1 \in \mathcal{R}$ and $s, t \in \mathit{State}$,*

$$\frac{\mathit{loopfree?}(r) \quad \mathcal{I}(T_1(r))(s, t)}{\mathit{final?}(\mathit{body}(r))(t)}$$

Proof.

By strong region induction on r to show that if a command $c \in r$ is enabled in t then there is a contradiction.

Case unit?(r): Any command $c \in r$ enabled in t is $\mathit{head}(r)$. Therefore $\mathit{head}(r) \longmapsto \mathit{head}(r)$ contradicting the assumption that r is loop-free.

Case $\neg$*unit?*(r) and the property holds for proper sub-regions of r: By Theorem (D.11) either $\mathcal{I}(\mathit{head}(r))(s, t)$ or there is a u such that $\mathcal{I}(\mathit{head}(r))(s, u)$ and $\mathit{trace}(T_1(\mathit{rest}(r)), u, t)$.

Assume $\mathcal{I}(\mathit{head}(r))(s, t)$ and that there is a command $c \in r$ such that $\mathit{enabled}(c)(t)$. It follows that there is a sub-region $r' = \mathit{region}(\mathit{label}(c), \mathit{body}(r) - \{Head(r)\})$ such that $r' \in \mathit{rest}(r)$ and $\mathit{enabled}(r')(t)$. Therefore there is a command $T_1(r') \in T_1(\mathit{rest}(r))$ such that $\mathit{enabled}(T_1(r'))(t)$. Assume that there is a state t_1 such that $\mathcal{I}(T_1(r'))(s, t_1)$. From $r' \subset r$ and $\mathit{loopfree?}(r)$ it follows that $\mathit{loopfree?}(r')$. From the inductive hypothesis it follows that $\mathit{final?}(\mathit{body}(r'))(t_1)$ and that no region $r_1 \in (\mathit{rest}(r))$ is enabled in T_1, otherwise $\mathit{head}(r_1) \in r'$ (from $\mathit{head}(r') \overset{\mathit{body}(r) - \{\mathit{head}(r)\}}{\longmapsto} \mathit{head}(r_1)$) and $\mathit{enabled}(\mathit{head}(r_1))(t_1)$ which is a contradiction. From Theorem (D.8), it follows that $\mathcal{I}(\mathit{head}(r); T_1(\mathit{rest}(r)))(s, t_1)$ is *true* and from the assumption $\mathcal{I}(T_1(r))(s, t)$ it follows that $t_1 = t$. Therefore $\mathit{head}(r') \overset{r}{\longmapsto} \mathit{head}(r')$, contradicting the assumption that r is loop-free, *loopfree?*. Assume that there is no state t_1 such that $\mathcal{I}(r')(t, t_1)$. From Theorem (D.12), there can be no state such that $\mathcal{I}(\mathit{head}(r); T_1(\mathit{rest}(r)))(s, t)$. This contradicts the assumption $\mathcal{I}(T_1(r))(s, t)$.

Assume there is a state u such that $\mathcal{I}(\mathit{head}(r))(s, u)$ and $\mathit{trace}(T_1(\mathit{rest}(r)), u, t)$. By definition of *trace*, there is a region $r' \in \mathit{rest}(r)$ and state t_1 such that $\mathcal{I}(T_1(r'))(u, t_1)$. If t_1 is not

t, then there is a trace through $T_1(\mathit{rest}(r))$ from t_1 to t and a region $r_1 \in \mathit{rest}(r)$ such that $\mathit{enabled}(T_1(r_1))(t_1)$. Therefore there is a command $c_1 \in r$ such that $\mathit{label}(c_1) = \mathit{label}(r_1)$ and, as before, $\mathit{head}(r') \overset{\mathit{body}(r)-\{\mathit{head}(r)\}}{\longmapsto} c_1$. From the definition of *region*, $c_1 \in r'$ contradicting the inductive hypothesis that $\mathcal{I}(T_1(r'))(u, t_1) \Rightarrow \mathit{final?}(r')(t_1)$. Therefore $t_1 = t$, t is final for r' and is therefore final for $\mathit{body}(r) - \{\mathit{head}(r)\}$. State t is also final for $\mathit{body}(r)$ otherwise $\mathit{head}(r) \overset{r}{\longmapsto} \mathit{head}(r)$ contradicting the assumption that r is loop-free. □

Lemma D.19 *For* $r, r_1 \in \mathcal{R}$ *and* $s, t \in \mathit{State}$*,*

$$\frac{\mathit{single?}(r) \quad r_1 \in \mathit{rest}(r) \quad \mathcal{I}(T_1(r_1))(s, t)}{\mathit{final?}(\mathit{body}(r) - \mathit{head}(r))(t)}$$

Proof.

Since $r_1 \in \mathit{rest}(r)$, $\mathit{head}(r) \notin r_1$ and r_1 is necessarily loop-free. From Lemma (D.18), t is final for r_1. State t is therefore final for $\mathit{body}(r) - \{\mathit{head}(r)\}$ since otherwise there is a command $c \in \mathit{body}(r) - \{\mathit{head}(r)\}$ such that $\mathit{enabled}(c)(t)$ which can be reached from $\mathit{head}(r_1)$ through $\mathit{body}(r) - \{\mathit{head}(r)\}$. Since $\mathit{head}(r_1) \overset{\mathit{body}(r)-\{\mathit{head}(r)\}}{\longmapsto} c$, it follows that $c \in r_1$, contradicting the assumption that $\mathit{final?}(r_1)(t)$. □

Theorem D.16 *For* $r \in \mathcal{R}$*,* $s, t \in \mathit{State}$*,*

$$\frac{\mathcal{I}(T_1(r))(s, t)}{\mathit{trace}(\mathit{body}(r), s, t)}$$

Proof.

By strong region induction.

Case $\mathit{unit?}(r)$: By definition, $T_1(r) = \mathit{head}(r)$ and $\mathit{trace}(\mathit{body}(r), s, t)$ follows immediately.

Case $\neg\mathit{unit?}(r)$ and the property holds for every proper subregion of r: By definition, $T_1 = \mathit{head}(r); T_1(\mathit{rest}(r))$. By Theorem (D.11), either $\mathcal{I}(\mathit{head}(r))(s, t)$ and the proof is complete or there is a u such that $\mathcal{I}(\mathit{head}(r), s, u)$ and $\mathit{trace}(T_1(\mathit{rest}(r)), u, t)$.

From the definition of *trace*, there is a region $r_1 \in \mathit{rest}(r)$ and state t_1 such that $\mathcal{I}(T_1(r_1))(u, t_1)$. From Lemma (D.19), state t_1 is final for r_1. State t_1 is therefore final for $T_1(\mathit{rest}(r))$, otherwise there is a command in r which is enabled in t_1, can be reached from $\mathit{head}(r_1)$ through $\mathit{body}(r) - \{\mathit{head}(r)\}$ and is therefore in r_1 leading to a contradiction. Since $\mathcal{I}(T_1(r_1))(u, t_1)$ and $r_1 \subset r$, the inductive hypothesis leads to $\mathit{trace}(\mathit{body}(r_1), u, t)$. From $\mathit{body}(r_1) \subseteq \mathit{body}(r) - \{\mathit{head}(r)\}$, it follows that $\mathit{trace}(\mathit{body}(r) - \{\mathit{head}(r)\}, u, t)$. From the definition of *trace* and the assumption $\mathcal{I}(\mathit{head}(r))(s, u)$ it follows that $\mathit{trace}(\mathit{body}(r), s, t)$, completing the proof. □

The result of applying the path transformation to region r is a command which is equivalent to a maximal trace through r. Theorem (D.14) established $\mathit{mtrace}(\mathit{body}(r), s, t) \Rightarrow \mathcal{I}(T_1(r))(s, t)$. Theorem (D.17) below establishes the reverse.

Theorem D.17 *For any region $r \in \mathcal{R}$ and states $s, t \in$ State,*

$$\frac{\mathit{single?}(r) \quad \mathcal{I}(T_1(r))(s,t)}{\mathit{mtrace}(\mathit{body}(r), s, t)}$$

Proof.

Note that from $\mathcal{I}(T_1(r))(s,t)$ it follows that *enabled*$(r)(s)$ and *head*$(r) \in$ *tset*(*body*$(r), s, t)$. The conclusion is straightforward from Theorem (D.16) and Lemma (D.19). □

D.10.1 Theorem (4.13)

Lemma D.20 *For $r \in \mathcal{R}$, $s, t \in$ State*

$$\frac{\mathit{single?}(r) \quad \mathit{enabled}(\mathit{head}(r))(s) \quad \mathit{mtrace}^{+}(\mathit{body}(r), s, t)}{\mathit{enabled}(\mathit{head}(r))(t) \vee \mathit{final?}(\mathit{body}(r))(t)}$$

Proof.

The proof is by right induction on the transitive closure of *mtrace*. There are two cases, the proof for the base case *mtrace*(*body*$(r), s, t)$ is straightforward from Lemma (D.16). The inductive case is similar: there is a state u such that *mtrace*$^{+}$(*body*$(r), s, u)$ and *mtrace*(*body*$(r), u, t)$. From the inductive hypothesis, *enabled*(*head*$(r))(u)$ and the proof is straightforward from Lemma (D.16).

□

Lemma D.21 *For $r \in \mathcal{R}$, $s, t \in$ State,*

$$\frac{\mathit{single?}(r) \quad s \overset{\mathit{unit}(T_1(r))}{\leadsto} t}{\mathit{mtrace}^{+}(\mathit{body}(r), s, t)}$$

Proof.

By induction on $s \overset{\mathit{unit}(T_1(r))}{\leadsto} t$. The inductive case is immediate from the inductive hypothesis.

Base case, $s \overset{T_1(r)}{\leadsto} t$: By Theorem (D.17), *mtrace*(*body*$(r), s, t)$ is *true* and *mtrace*$^{+}$(*body*$(r), s, t)$ is immediate by definition.

□

Lemma D.22 *For $p \in \mathcal{P}$, $s, t \in$ State,*

$$\frac{\mathit{mtrace}^{+}(p, s, t)}{s \overset{p}{\leadsto} t}$$

Proof.

Straightforward by induction on the transitive closure of *mtrace*, ($mtrace^{+}(p, s, t)$) and from Lemma (D.3). □

Proof. *Theorem 3.12*

There are three properties to prove.

1. $label(unit(T_1(r))) = label(r)$.
   From the definition of *unit* and T_1, $label(unit(T_1(r))) = label(head(r))$. By definition $label(r) = label(head(r))$, completing the proof.
2. $body(unit(T_1(r))) \sqsubseteq body(r)$
   The required property is $\forall s, t : s \overset{unit(T_1(r))}{\rightsquigarrow} t \Rightarrow s \overset{r}{\rightsquigarrow} t$.
   By definition, $body(unit(T_1(r))) = \{T_1(r)\}$. From Theorem (D.15), for any $s_1, t_1 \in State$, $s_1 \overset{T_1(r)}{\rightsquigarrow} t_1 \Rightarrow s_1 \overset{r}{\rightsquigarrow} t_1$. The property is therefore straightforward from Lemma (D.17).
3. $s \overset{unit(T_1)(r)}{\rightsquigarrow} t \wedge final?(unit(T_1(r)))(t) \Rightarrow s \overset{r}{\rightsquigarrow} t \wedge final?(r)(t)$.
   From item (2), $s \overset{unit(T_1)(r)}{\rightsquigarrow} t \Rightarrow s \overset{r}{\rightsquigarrow} t$. By Lemma (D.21), $mtrace^{+}(body(r), s, t)$ is *true*. The assumption $final?(unit(T_1(r)))(t)$ leads to $\neg enabled(head(r))(t)$, otherwise $enabled(T_1(r))(t)$ (since $label(T_1(r)) = label(head(r))$). Therefore, from Lemma (D.20), $final?(r)(t)$ is *true* and the proof is complete.

□

D.10.2 Theorem (4.14)

Lemma D.23 *For* $r \in \mathcal{R}$, $s, t \in State$,

$$\frac{single?(r) \quad enabled(head(r)) \quad mtrace^{+}(body(r), s, t)}{s \overset{unit(T_1(r))}{\rightsquigarrow} t}$$

Proof.

By induction on $mtrace^{+}(body(r), s, t)$. The base case, with $mtrace(body(r), s, t)$, is immediate from Theorem (D.14). The inductive case is immediate from the inductive hypothesis and Lemma (D.20). □

Proof. *Theorem (4.14)*

($\Rightarrow$): From Theorem (4.13), $label(r_1) = label(unit(T_1(r)))$, therefore $enabled(unit(T_1(r)))(s)$ iff $enabled(r_1)(s)$. Also from Theorem (4.13) and the definition of $\sqsubseteq$, if $s \overset{unit(T_1(r))}{\rightsquigarrow} t$ and $final?(unit(T_1)(r))$ then $s \overset{r}{\rightsquigarrow} t$ (from $body(unit(T_1(r))) \sqsubseteq body(r)$) and $final?(r)(t)$.

($\Leftarrow$): As before, $\textit{enabled}(r)(s) \Leftrightarrow \textit{enabled}(\textit{unit}(T_1)(r))$. From Corollary (D.1), if $s \overset{r}{\leadsto} t$ and $\textit{final?}(t)$ then $\textit{mtrace}^{+}(\textit{body}(r), s, t)$. From Lemma (D.23), this leads to $s \overset{\textit{unit}(T_1(r))}{\leadsto} t$. Assume $\textit{final?}(\textit{unit}(T_1(r)))(t)$ is *false*, then the only command in the region, $T_1(r)$ is enabled in t. $\textit{label}(T_1(r)) = \textit{label}(\textit{head}(r))$ and therefore $\textit{head}(r)$ is enabled in t contradicting the assumption that $\textit{final?}(r)(t)$, since $\textit{head}(r) \in r$. Therefore $\textit{final?}(\textit{unit}(T_1)(r))(t)$ and the proof is complete.

□

Theorem (4.15)

The equivalence of the failure of a region and a transformed region is established in two steps. The first shows that if the head of a region r halts then so does the transformed region $T_1(r)$. The second step shows that if r beginning in a state s leads to a state t in which r halts then $T_1(r)$ either halts in s or there is a state u produced by $T_1(r)$ beginning in s in which $T_1(r)$ halts.

Lemma D.24 *For $r \in \mathcal{R}$, $s, t \in State$,*

$$\frac{\textit{single?}(r) \quad \textit{halt?}(\textit{head}(r))(s)}{\textit{halt?}(T_1(r))(s)}$$

Proof.

By definition of *halt?*, there are two properties to be proved.

The first is that from $\textit{enabled}(T_1(r))(s) \Rightarrow \textit{enabled}(\textit{head}(r))(s)$. This is straightforward from $\textit{label}(T_1(r)) = \textit{label}(\textit{head}(r))$.

The second is that from $\forall t_1 : \neg\mathcal{I}(\textit{head}(r))(s, t_1)$, it can be inferred that $\forall t : \neg\mathcal{I}(T_1(r))(s, t)$. Assume $\mathcal{I}(T_1(r))(s, t)$ for some $t \in State$. From Theorem (D.16), there is a trace from s to t through r, $\textit{trace}(\textit{body}(r), s, t)$. Since $\textit{enabled}(\textit{head}(r))(s)$ (from $\textit{halt?}(\textit{head}(r))(s)$), $\textit{head}(r)$ is the first command of the trace. From the definition of *trace*, there is a state t' such that $s \overset{\textit{head}(r)}{\leadsto} t'$ contradicting the assumption that no such state exists. □

Lemma D.25 *For $r \in \mathcal{R}$, $s, t \in State$,*

$$\frac{\textit{single?}(r) \quad \textit{enabled}(r)(s) \quad \textit{trace}(\textit{body}(r), s, t) \quad \textit{halt?}(\textit{body}(r))(t)}{\textit{halt?}(T_1(r))(s) \lor \textit{enabled}(\textit{head}(r))(t)}$$

Proof.

By strong region induction on r.

Case unit?(r): The only command in r is $\textit{head}(r)$. From the definition of $\textit{halt?}(\textit{body}(r))(t)$, there is a command in r which is enabled in t. Since $\textit{unit?}(r)$, the only command in r is $\textit{head}(r)$.

Case $\neg$*unit?*(r) and the property is *true* for the proper sub-regions of r: Since *enabled*$(r)(s)$ it follows that *enabled*(*head*$(r))(s)$. Since *halt?*(*body*$(r))(t)$, there is a command $c \in r$ such that *enabled*$(c)(t)$ and $\forall t_1 : \neg\mathcal{I}(c)(t, t_1)$. If $c = $ *head*(r) then the proof is immediate, assume $c \neq$ *head*(r). From the definition of *trace*, there are two cases to consider.

Assume $s \overset{head(r)}{\leadsto} t$. From *enabled*$(c)(t)$ it follows that c begins a proper sub-region r' of r, $r' = $ *region*(*label*(c), *body*$(r) - \{$*head*$(r)\})$, such that $r' \in$ *rest*(r) and *enabled*$(r')(t)$. From *halt?*$(c)(t)$, $c = $ *head*(r') and by Lemma (D.24), *halt?*$(T_1(r'))(t)$. Since $r' \in$ *rest*(r), it can be inferred that $T_1(r') \in T_1($*rest*$(r))$. In addition, the truth of *head*$(r) \notin T_1($*rest*$(r))$, $\mathcal{I}($*head*$(r))(s, t)$ and *halt?*$(T_1(r'))(t)$ satisfies the assumptions of Theorem (D.12). From Theorem (D.12) it follows that $\forall t_2 : \neg\mathcal{I}($*head*$(r); T_1($*rest*$(r)))(s, t_1)$ and therefore *halt?*(*head*$(r); T_1($*rest*$(r)))(s)$. The proof for this case follows immediately from the definition of T_1.

Assume there is a state u such that *trace*(*body*$(r) - \{$*head*$(r)\}, u, t)$. Then there is a command $c_1 \in$ *body*$(r) - \{$*head*$(r)\}$ such that *enabled*$(c_1)(u)$. Command c_1 begins a proper sub-region r' of r, $r' = $ *region*(*label*$(c_1$, *body*$(r) - \{$*head*$(r)\})$, such that $r' \in$ *rest*(r), *enabled*$(r')(u)$ and *single?*(r') (from *loopfree?*(r')).

From the inductive hypothesis, there are two cases: the first that *enabled*(*head*$(r'))(t)$ is *true*, the second that *halt?*$(T_1(r'))(u)$ and *halt?*(*body*$(r'))(t)$. If *enabled*(*head*$(r')(t))$ is *true* then from *enabled*(*head*$(r'))(u)$ and $u \overset{body(r)-\{head(r)\}}{\leadsto} t$, *head*$(r') \overset{body(r)-\{head(r)\}}{\longmapsto}$ *head*(r'), contradicting the assumption *single?*(r). Assume *halt?*$(T_1(r')(u))$. Since *single?*(r), no other command in $T_1($*rest*$(r))$ can be enabled in t and *enabled*$(T_1(r'))(u)$. Theorem (D.21) applies and leads to *halt?*(*head*$(r); T_1($*rest*$(r)))(s)$. The conclusion *halt?*$(T_1(r))$ follows from the definition of T_1.

$\square$

Lemma D.26 *For $r \in \mathcal{R}$ and $s, t \in$ State,*

$$\frac{single?(r) \quad halt?(T_1(r))(s)}{halt?(head(r))(s) \vee \exists t : trace(body(r), s, t) \wedge halt?(body(r))(t)}$$

Proof.

By strong induction on r.

Case unit?(r): By definition $T_1(r) = $ *head*(r) and the proof is immediate.

Case $\neg$*unit?*(r) and assuming the property is *true* for all proper sub-regions of r:

Assume there is a state t such that $\mathcal{I}($*head*$(r))(s, t)$. Also assume no command $c \in T_1($*rest*$(r))$ is enabled in t. From Theorem (D.7), $\mathcal{I}($*head*$(r); T_1($*rest*$(r)))(s, t)$ follows immediately contradicting the assumption that *halt?*$(T_1(r))(s)$.

Assume there is a command $c \in T_1($*rest*$(r))$ such that *enabled*$(c)(t)$. It follows that there is a region $r' = $ *region*(*label*(c), *body*$(r) - \{$*head*$(r)\})$ such $c = T_1(r')$. Assume there is no t_1 such that $\mathcal{I}(T_1(r'))(t, t_1)$ then *halt?*$(r')(t)$ is immediate from definition. In addition, since

$\mathcal{I}(\mathit{head}(r))(s,t)$ there is a trace through r, $\mathit{trace}(\mathit{body}(r), s, t)$, and, since $\mathit{head}(r') \in \mathit{body}(r)$, $\mathit{halt?}(\mathit{body}(r))(t)$. Assume there is a t_1 such that $\mathcal{I}(T_1(r'))(t, t_1)$. From Theorem (D.8) and the fact that r' is loop-free (from $\mathit{single?}(r)$), $\mathcal{I}(\mathit{head}(r); T_1(\mathit{rest}(r)))(s, t_1)$ is *true*. This contradicts the assumption $\mathit{halt?}(T_1(r))(s)$. □

Lemma D.27 *For $r \in \mathcal{R}$ and $s, t \in State$,*

$$\frac{\mathit{single?}(r) \quad \mathit{enabled}(\mathit{unit}(T_1(r)))(s) \quad \mathit{halt?}(\mathit{unit}(T_1(r)))(s)}{\mathit{halt?}(\mathit{head}(r))(s) \vee \exists t : \mathit{trace}(\mathit{body}(r), s, t) \wedge \mathit{halt?}(\mathit{body}(r))(t)}$$

Proof.

From the assumption $\mathit{halt?}(\mathit{unit}(T_1(r)))$, there is a command $c \in \mathit{body}(\mathit{unit}(T_1(r)))$ such that $\mathit{halt?}(c)(s)$. From the definition of *unit*, $c = T_1(r)$ and by Lemma (D.26), the proof is immediate.

□

Lemma D.28 *For $r \in \mathcal{R}$ and $s, t, u \in State$,*

$$\frac{\mathit{single?}(r) \quad \mathit{enabled}(\mathit{head}(r))(s) \quad s \overset{r}{\leadsto} t \quad \mathit{halt?}(\mathit{body}(r))(t)}{\mathit{halt?}(T_1(r))(s) \vee \exists u : s \overset{\mathit{unit}(T_1(r))}{\leadsto} u \wedge \mathit{halt?}(\mathit{unit}(T_1(r)))(u)}$$

Proof.

From Theorem (D.2) and $s \overset{r}{\leadsto} t$, either $\mathit{trace}(\mathit{body}(r), s, t)$ or there is a $u \in State$ such that $\mathit{mtrace}^+(\mathit{body}(r), s, u) \wedge \mathit{trace}(\mathit{body}(r), u, t)$. The proof for both cases are similar and only the second is given.

From Lemma (D.20), $\mathit{enabled}(\mathit{head}(r))(s)$ and $\mathit{mtrace}^+(\mathit{body}(r), s, u)$, either $\mathit{final?}(r)(u)$ or $\mathit{enabled}(\mathit{head}(r))(u)$. Since there is a trace from u to t through $\mathit{body}(r)$, $\mathit{final?}(\mathit{body}(r))(u)$ leads to a contradiction. Therefore $\mathit{enabled}(\mathit{head}(r))(u)$ is *true* and there is a trace, $\mathit{trace}(\mathit{body}(r), u, t)$ and $\mathit{halt?}(\mathit{body}(r))(t)$.

From Lemma (D.25), either $\mathit{halt?}(T_1(r))(t)$ and the case is proved or $\mathit{enabled}(\mathit{head}(r))(t)$. Assume there is a state t_1 such that $\mathcal{I}(\mathit{head}(r))(t, t_1)$. Since $\mathit{head}(r) \in \mathit{body}(r)$ this contradicts the assumption $\mathit{halt?}(\mathit{body}(r))(t)$ and the proof is complete. □

Proof. *Theorem (4.15)*

($\Rightarrow$): Assume $\mathit{enabled}(r)(s)$ and $\mathit{halt?}(\mathit{unit}(T_1(r)))(s)$, then either $\mathit{halt?}(\mathit{body}(\mathit{unit}(T_1(r))))(s)$ or there is a state t such that $s \overset{\mathit{unit}(T_1(r))}{\leadsto} t$ and $\mathit{halt?}(\mathit{body}(\mathit{unit}(T_1)(r)))(t)$. For the first case, the proof is straightforward from Lemma (D.27) and Lemma (D.3).

For the second case, note that from Theorem (D.15) and Lemma (D.17), $s \overset{r}{\leadsto} t$ is straightforward. In addition, since any command of $\mathit{body}(\mathit{unit}(T_1(r)))$ enabled in t is $T_1(r)$ it follows that *enabled*$(\mathit{unit}(T_1)(r))$ is *true*. The proof is straightforward by Lemma (D.27). Note that if there is a state u, such that *trace*$(\mathit{body}(r), t, u)$ then $t \overset{r}{\leadsto} u$ is also *true* (Lemma D.3).

($\Leftarrow$): Assume *enabled*$(r)(s)$ and *halt?*$(r)(s)$. There are two cases. Assume *halt?*$(\mathit{body}(r))(s)$. From the assumption *enabled*$(r)(s)$, the only command of r enabled in s is *head*(r) and therefore *halt?*$(\mathit{head}(r))(s)$. The proof is straightforward from Lemma (D.24). For the second case, assume a state t such that $s \overset{r}{\leadsto} t$ and *halt?*$(\mathit{body}(r))(t)$. The proof is immediate from Lemma (D.28). □

D.11 General Transformation

The proofs for the theorems concerning the general transformation are based on a number of basic properties of traces and regions.

If command c of program p is enabled in s and there is trace through p from s to t then there is a trace through the region of p which begins with c.

Lemma D.29 *For* $p \in \mathcal{P}$, $c, c_1 \in \mathcal{L}$ *and* $s, t \in \mathit{State}$,

$$\frac{\mathit{trace}(p, s, t) \quad c \in p \quad \mathit{enabled}(c)(s)}{\mathit{trace}(\mathit{body}(\mathit{region}(\mathit{label}(c), p), s, t))}$$

Proof.

By induction on *trace*(p, s, t).

Case $s \overset{c}{\leadsto} t$. By definition, $c \in \mathit{region}(\mathit{label}(c), p)$ and the conclusion is immediate.

Case $s \overset{c}{\leadsto} u$ and *trace*$(p - \{c\}, u, t)$ and the property is *true* for *trace*$(p - \{c\}, u, t)$: Since there is a trace from u to t through $p - \{c\}$, there is a command $c_1 \in p - \{c\}$ which is enabled in u. From Theorem (4.8), $c \longmapsto c_1$ and therefore, for every $c_2 \in \mathit{body}(\mathit{region}(\mathit{label}(c_1), p - \{c\}))$, $c \overset{p}{\longmapsto} c_2$ and $c_2 \in \mathit{body}(\mathit{region}(\mathit{label}(c), p))$. Since $\mathit{body}(\mathit{region}(\mathit{label}(c_1), p - \{c\}))$ does not contain c, $\mathit{body}(\mathit{region}(\mathit{label}(c_1), p - \{c\})) \subseteq \mathit{body}(\mathit{region}(\mathit{label}(c), p)) - \{c\}$. From the inductive hypothesis, *trace*$(\mathit{body}(\mathit{region}(c_1, p - \{c\})), u, t)$ is *true*. From Lemma (D.6), it follows that *trace*$(\mathit{body}(\mathit{region}(\mathit{label}(c), p)) - \{c\}, u, t)$ is also *true*. From the assumption $s \overset{c}{\leadsto} u$, the conclusion *trace*$(\mathit{body}(\mathit{region}(\mathit{label}(c), p)), s, t)$ is immediate from the definitions. □

If a command c is a member of a trace set between states s to t, then there is a trace from s to a state in which c is enabled.

Lemma D.30 *For* $p \in \mathcal{P}$, $c \in \mathcal{L}$ *and* $s, t, u \in \mathit{State}$,

$$\frac{c \in \mathit{tset}(p, s, t) \quad \neg\mathit{enabled}(c)(s)}{\exists u : \mathit{trace}(p, s, u) \wedge \mathit{enabled}(c)(u)}$$

Proof.

Straightforward, by strong induction on p and cases of *tset*. □

If program p_1 is a subset of program p_2 then trace set through p_1 is a subset of a trace set through p_2.

Lemma D.31 *For $p_1, p_1 \in \mathcal{P}$ and $s, t \in State$,*

$$\frac{p_1 \subseteq p_2}{\mathit{tset}(p_1, s, t) \subseteq \mathit{tset}(p_2, s, t)}$$

Proof.

By strong induction on p_1 and cases of *tset*. The property to proved is for any $c' \in \mathcal{L}$, $c' \in \mathit{tset}(p_1, s, t) \Rightarrow c' \in \mathit{tset}(p_2, s, t)$.

Case $c \in p_1$, $s \overset{c}{\leadsto} t$ and $c' = c$. Since $c \in p_2$, the proof is immediate by definition of *tset*.

Case $u \in \mathit{State}$ and $c \in p_1$, $s \overset{c}{\leadsto} u$, $\mathit{trace}(p_1 - \{c\}, u, t)$ and $c' = c$. As before, the proof is straightforward. Note that from Lemma (D.6) and the assumptions it follows that $\mathit{trace}(p_2 - \{c\}, u, t)$ is *true*.

Case $u \in \mathit{State}$ and $c \in p_1$, $s \overset{c}{\leadsto} u$ and $c' \in \mathit{tset}(p_1 - \{c\}, u, t)$ with the property true for all proper subsets of p_1. From the inductive hypothesis and the assumptions it follows that $c' \in \mathit{tset}(p_2 - \{c\}, u, t)$. The extension to $c' \in \mathit{tset}(p_2 - \{c\}, u, t)$ is straightforward from the definition of *tset*. □

If a command c_1 is enabled in a state t after a trace through a program p beginning with a command c then c_1 is a member of the trace set of the region beginning with c.

Lemma D.32 *For $p \in \mathcal{P}$, $c, c_1 \in \mathcal{L}$ and $s, t \in State$,*

$$\frac{\mathit{trace}(p, s, t) \quad c \in p \quad \mathit{enabled}(c)(s) \quad c_1 \in \mathit{tset}(p, s, t)}{c_1 \in \mathit{tset}(\mathit{body}(\mathit{region}(\mathit{label}(c), p)), s, t)}$$

Proof.

By strong program induction on p and by cases of $\mathit{tset}(p, s, t)$.

Case $s \overset{c}{\leadsto} t$ and $c_1 = c$: The proof is immediate from $c \in \mathit{body}(\mathit{region}(\mathit{label}(c), p), s, t)$ and therefore $c \in \mathit{tset}(\mathit{body}(\mathit{region}(\mathit{label}(c), p)), s, t)$.

Case $s \overset{c}{\leadsto} u$, $c_1 = c$, $\mathit{trace}(p - \{c\}, u, t)$ and the property is *true* for $p - \{c\}$: From the definitions of *region* and *trace*, $c \in \mathit{body}(\mathit{region}(c), p))$ and there is a $c' \in p - \{c\}$ such that c' is enabled in u. Since $s \overset{c}{\leadsto} u$, it follows that $c \longmapsto c'$ and therefore $\mathit{body}(\mathit{region}(\mathit{label}(c'), p - \{c\}))$ is a subset of $\mathit{body}(\mathit{region}(\mathit{label}(c), p))$. From the assumption $\mathit{trace}(p - \{c\}, u, t)$ and from Lemma (D.29) it follows that $\mathit{trace}(\mathit{body}(\mathit{region}(\mathit{label}(c'), p - \{c\})), u, t)$ is *true*. From

Lemma (D.31), this establishes $\mathit{trace}(\mathit{body}(\mathit{region}(\mathit{label}(c), p)), u, t)$. The conclusion that c_1 is a member of $\mathit{tset}(\mathit{body}(\mathit{region}(\mathit{label}(c), p)), s, t)$ follows from the definition of *tset*.

Case $s \overset{c}{\leadsto} u, c_1 \in \mathit{tset}(p - \{c\}, u, t)$ and the property is *true* for $p - \{c\}$: From $\mathit{trace}(p, s, t)$, there is a trace $\mathit{trace}(p - \{c\}, u, t)$ and therefore a command $c_2 \in p - \{c\}$ such that $\mathit{enabled}(c_2)(u)$. This command begins a region r of $p - \{c\}$, $r = \mathit{region}(\mathit{label}(c_2), p - \{c\})$, and, from the inductive hypothesis, $c_1 \in \mathit{tset}(\mathit{body}(r), u, t)$. From Theorem (4.8), $c \longmapsto c_2$ and therefore $c \overset{p}{\longmapsto} c_3$, for all $c_3 \in \mathit{body}(r)$. Since, by definition, $c \notin \mathit{body}(r)$ it follows that $\mathit{body}(r)$ is a subset of $\mathit{body}(\mathit{region}(\mathit{label}(c), p)) - \{c\}$.

Let $p' = \mathit{body}(\mathit{region}(\mathit{label}(c, p)))$. By definition, $\mathit{body}(r) \subseteq p' - \{c\}$ and, from Lemma (D.31), $\mathit{tset}(\mathit{body}(r), u, t) \subseteq \mathit{tset}(p' - \{c\}, u, t)$. Since $c_1 \in \mathit{tset}(\mathit{body}(r), u, t)$ and $\mathit{tset}(p' - \{c\}, u, t)$ is a subset of $\mathit{tset}(p', s, t)$, the proof is complete. □

D.11.1 Loops

The properties of loops in a region are syntactic and based on the commands beginning a loop. The proof of Theorem (4.16) is based on the property that, for region r, a loop in a sub-region r_1 of r, $r_1 \in \mathit{loops}(r)$, is also a loop in r. To show that this is true, strong induction is used with the inductive hypothesis applied to a sub-region $r_2 \in \mathit{rest}(r)$.

Any command beginning a loop in a sub-region of r (in $\mathit{rest}(r)$) also begins a loop in r.

Lemma D.33 *For* $r, r_1 \in \mathcal{R}$*, and* $c \in \mathcal{L}$*,*

$$\frac{r_1 \in \mathit{rest}(r) \quad \mathit{lphead?}(c, r_1)}{\mathit{lphead?}(c, r)}$$

Proof.

From $\mathit{lphead?}(c, r_1)$ it follows that there is a set $p \subseteq \mathit{body}(r_1)$ such that $\mathit{head}(r_1) \overset{p}{\longmapsto} c$ and $c \overset{(\mathit{body}(r_1)-p)\cup\{c\}}{\longmapsto} c$. By definition, $\mathit{head}(r) \notin \mathit{body}(r_1)$ and therefore $\mathit{head}(r) \notin p$. It follows that there is a set $p' = p \cup \{\mathit{head}(r)\}$ such that $\mathit{head}(r) \overset{p'}{\longmapsto} c$ and $c \overset{(\mathit{body}(r)-p')\cup\{c\}}{\longmapsto} c$. The conclusion $\mathit{lphead?}(c, r)$ is immediate from the definition. □

Any command beginning a loop in a sub-region of r (in $\mathit{loops}(r)$) also begins a loop in r.

Lemma D.34 *For* $r, r_1 \in \mathcal{R}$*, and* $c \in \mathcal{L}$*,*

$$\frac{r_1 \in \mathit{loops}(r) \quad \mathit{lphead?}(c, r_1)}{\mathit{lphead?}(c, r)}$$

Proof.

There are two cases to consider, for both the proof is similar to that of Lemma (D.33). Since $r_1 \in \mathit{loops}(r)$, there is a command $c_1 \in \mathit{cuts}(r)$ such that $r_1 = \mathit{region}(\mathit{label}(c_1), \mathit{lpbody}(c_1, r))$.

Case $c_1 = \mathit{head}(r)$. The proof is straightforward from $\mathit{lpbody}(c_1, r) \subseteq \mathit{body}(r)$, from the definition of *lphead?* and by the properties of the *reaches* relation.

Case $c_1 \neq \mathit{head}(r)$. By definition, $\mathit{lphead?}(c_1, r)$ is *true* and there is a path $p \subseteq \mathit{body}(r)$ such that $\mathit{head}(r) \xmapsto{p} c_1$ and, for $p' = \mathit{body}(r) - p \cup \{c_1\}$, $c_1 \xmapsto{p'} c_1$. From $\mathit{lphead?}(c, r_1)$, there is also a path $p_1 \subseteq \mathit{body}(r_1)$ such that $c_1 \xmapsto{p_1} c$ and, for $p'_1 = \mathit{body}(r') - p'_1 \cup \{c_1\}$, $c \xmapsto{p'_1} c$. By definition, $\mathit{lpbody}(c_1, r) \subseteq \mathit{body}(r)$, therefore there is a path $p_2 = p \cup p_1$ such that $\mathit{head}(r) \xmapsto{p_2} c$. Since $p'_1 \cap p_2 = \{\}$, there is also a path $p'_2 = (\mathit{body}(r) - p_2) \cup \{c\}$ such that $p' \subseteq \mathit{body}(r)$ and $c \xmapsto{p'} c$. The conclusion $\mathit{lphead?}(c, r)$ follows by definition. □

A single loop r is a region in which no command can reach itself without passing through $\mathit{head}(r)$. Conversely, if a command c can reach itself through $\mathit{body}(r) - \{\mathit{head}(r)\}$ then there is a command $c \in \mathit{body}(r) - \{\mathit{head}(r)\}$ beginning a loop in r.

Lemma D.35 *For* $r \in \mathcal{R}$, $p \in \mathcal{P}$ *and* $c, c_1 \in \mathcal{L}$,

$$\frac{p = \mathit{body}(r) - \{\mathit{head}(r)\} \quad c \xmapsto{p} c}{\exists c_1 : c_1 \neq \mathit{head}(r) \wedge \mathit{lphead?}(c_1, r) \wedge c_1 \xmapsto{p} c \wedge c \xmapsto{p} c_1}$$

Proof.

By strong region induction on r. By definition and $c \xmapsto{p} c$, $c \in p$ and therefore $c \in \mathit{body}(r) - \{\mathit{head}(r)\}$. It follows that $\neg\mathit{unit?}(r)$ is *true*.

Since $c \in r$ it follows that $\mathit{head}(r) \xmapsto{r} c$. By Theorem (D.3), there is a $c_2 \in \mathit{body}(r) - \mathit{head}(r)$ which begins a region $r' = \mathit{region}(\mathit{label}(c_2), \mathit{body}(r) - \{\mathit{head}(r)\})$ such that $c \in r'$ and $r' \in \mathit{rest}(r)$. Let $p' = \mathit{body}(r') - \{\mathit{head}(r')\}$.

Assume $c \xmapsto{p'} c$. By the inductive hypothesis, there is a $c_1 \in p'$ such that $\mathit{lphead?}(c_1, r')$. From Lemma (D.33), $\mathit{lphead?}(c_1, r)$ is *true*. From Theorem (4.7), $p' \subset p$ and $c_1 \not\mathit{head}(r)$, it follows that $c_1 \xmapsto{p} c$ and $c \xmapsto{p} c_1$, completing the proof for this case.

Assume $\neg(c \xmapsto{p'} c)$. Since $c \in r'$, $\mathit{head}(r') \xmapsto{r'} c$ and, since $c_2 = \mathit{head}(r')$, $c_2 \xmapsto{r'} c$. From the assumption and the definition of *region*, $c \xmapsto{r'} \mathit{head}(r')$ and therefore $c \xmapsto{r'} c_2$. By transitivity of *reaches*, $c_2 \xmapsto{r'} c_2$ and, by definition, $\mathit{lphead?}(c_2, r')$. Therefore there is a $c_1 = c_2$ such that $c_1 \xmapsto{p} c$ and $c \xmapsto{p} c_1$. Since $c_1 = c_2$, by Lemma (D.33) it follows that $\mathit{lphead?}(c_1, r)$ and the proof is complete. □

D.11.2 Theorem (4.16)

Proof.

There are two cases: when $r' = \textit{region}(\textit{label}(\textit{head}(r)), \textit{lpbody}(\textit{head}(r), r))$ and when $r' = \textit{region}(\textit{label}(c), \textit{lpbody}(c, r))$ for $c \in \textit{body}(r) - \{\textit{head}(r)\}$ and $\textit{lphead?}(c, r)$. The proofs are similar and only the second case will be considered.

Assume $c \in \textit{body}(r) - \{\textit{head}(r)\}$ such that $\textit{lphead?}(c, r)$ and $r' = \textit{region}(\textit{label}(c), \textit{lpbody}(c, r))$. Let $p = \textit{body}(r') - \{\textit{head}(r')\}$. By definition $c \in \textit{cuts}(r)$, $\textit{head}(r') = c$ and $r' \in \textit{loops}(r)$. Assume $\neg\textit{single?}(r')$, it follows that there is a command $c_1 \in \textit{body}(r')$ such that $c_1 \xmapsto{p} c_1$. From Lemma (D.35), there is a command $c_2 \in \textit{body}(r')$ such that $\textit{lphead?}(c, r')$ and $c_2 \neq \textit{head}(r')$. From Lemma (D.34), c_2 is also a loop head of r, $\textit{lphead?}(c_2, r)$ and $c_2 \in \textit{cuts}(r)$. By definition, $\textit{lpbody}(c, r) = (\textit{body}(r) - \textit{lpheads}(r)) + c$. Since $c_2 \in \textit{cuts}(r)$ and $c_2 \neq \textit{head}(r')$, $c_2 \in \textit{lpbody}(c, r)$ leads to a contradiction. Therefore there can be no c_1 such that $c_1 \xmapsto{p} c_1$ leading to the conclusion that $\textit{single?}(r')$.

Note that for the case when $r' = \textit{region}(\textit{label}(\textit{head}(r)), \textit{lpbody}(\textit{head}(r), r))$, Lemma (D.35) requires that $c_1 \neq \textit{head}(r)$. This is straightforward from $c_1 \xmapsto{\textit{body}(r) - \{\textit{head}(r)\}} c_1$. □

D.11.3 Theorem (4.17)

The proof of Theorem (4.17) is based on showing that the commands of a transformed region $T_2(r)$ correspond to a maximal trace through r, restricted in the cut-points of r. For command $c \in T_2(r)$ and states s, t, if $\mathcal{I}(c)(s, t)$ then $\textit{rmtrace}(\textit{lpheads}(r))(\textit{body}(r), s, t)$. This is established in two steps, the first to show that from $\mathcal{I}(c)(s, t)$, it follows that there is a command $c_1 \in r$ such that $\textit{mtrace}(\textit{lpbody}(c, r), s, t)$. The second step shows that this establishes $\textit{rmtrace}(\textit{lpheads}(r))(\textit{body}(r), s, t)$. The proofs are based on a number of basic properties of maximal traces.

The transitive closure of a restrict maximal trace through a region establishes the *leads-to* relation through r.

Lemma D.36 *For $p \in \mathcal{P}$, $a \in \textit{Set}(\mathcal{L})$, $s, t \in \textit{State}$,*

$$\frac{\textit{rmtrace}^{+}(a)(p, s, t)}{s \overset{p}{\rightsquigarrow} t}$$

Proof.

Straightforward, by induction on the transitive closure of *rmtrace*. The base case follows from Lemma (D.3). The inductive case is immediate from the inductive hypothesis. □

Lemma D.37 *For $r \in \mathcal{R}$,*

$$\textit{label}(T_2(r)) = \textit{label}(r)$$

Proof.

Immediate from definitions. □

A maximal trace through a region r constructed from a cut-point c and the loop body of c, *lpbody*(c, r), establishes a maximal trace through *lpbody*(c, r).

Lemma D.38 *For* $r \in \mathcal{R}$, $c \in \mathcal{L}$ *and* $s, t \in$ *State,*

$$\frac{\mathit{enabled}(c)(s) \quad c \in \mathit{cuts}(r) \quad \mathit{mtrace}(\mathit{body}(\mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r))), s, t)}{\mathit{mtrace}(\mathit{lpbody}(c, r), s, t)}$$

Proof.

There are two properties to prove: the first that *trace*$(\mathit{lpbody}(c, r), s, t)$ and the second that *final?*$(\mathit{lpbody}(c, r) - \mathit{tset}(\mathit{lpbody}(c, r), s, t))(t)$ both follow from assumptions.

Let $p = \mathit{body}(\mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r)))$.

trace$(\mathit{lpbody}(c, r), s, t)$: From the assumption *mtrace*(p, s, t) and from the definition of *mtrace*, it follows that *trace*(p, s, t) is *true*. From Corollary (4.3), $p \subseteq \mathit{lpbody}(c, r)$ and, from Lemma (D.6), *trace*$(\mathit{lpbody}(c, r), s, t)$ is straightforward.

final?$(\mathit{lpbody}(c, r) - \mathit{tset}(\mathit{lpbody}(c, r), s, t))(t)$: From the assumption, *mtrace*(p, s, t), *final?*$(p - \mathit{tset}(p, s, t))$ is *true* and there is no $c_1 \in p - \mathit{tset}(p, s, t)$ such that *enabled*$(c_1)(t)$.

Assume there is a command $c_2 \in \mathit{lpbody}(c, r) - \mathit{tset}(\mathit{lpbody}(c, r))$ such that *enabled*$(c_2)(s)$. From $c_2 \in \mathit{lpbody}(c, r)$, *enabled*$(c)(s)$, *enabled*$(c_2)(t)$ and $s \overset{\mathit{lpbody}(c,r)}{\leadsto} t$ it follows, from Theorem (4.8), that $c \overset{\mathit{lpbody}(c,r)}{\longmapsto} t$. By definition of *region*, $c_2 \in \mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r))$ and therefore $c_2 \in p$.

Assume that $c_2 \in \mathit{tset}(p, s, t)$. From Lemma (D.31), and $p \subseteq \mathit{lpbody}(c, r)$ it follows that $c_2 \in \mathit{tset}(\mathit{lpbody}(c, r))$ leading to a contradiction with the assumption $c_2 \notin \mathit{tset}(\mathit{lpbody}(c, r), s, t)$ Therefore $c_2 \in p - \mathit{tset}(p, s, t)$ contradicting the assumption that there is no command in $p - \mathit{tset}(p, s, t)$ which is enabled in t. □

For a cut-point c of a region r, a maximal trace through *lpbody*(c, r) establishes a maximal trace through region beginning with c and constructed from *lpbody*(c, r).

Lemma D.39 *For* $r \in \mathcal{R}$, $c \in \mathcal{L}$ *and* $s, t \in$ *State,*

$$\frac{\mathit{enabled}(c)(s) \quad c \in \mathit{cuts}(r) \quad \mathit{mtrace}(\mathit{lpbody}(c, r), s, t)}{\mathit{mtrace}(\mathit{body}(\mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r)), s, t)}$$

Proof.

Let $p = \mathit{body}(\mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r)))$. There are two properties to prove: the first that *trace*(p, s, t) and the second that *final?*$(p - \mathit{tset}(p, s, t))(t)$.

trace(p, s, t): From the assumption *mtrace*$(\mathit{lpbody}(c, r), s, t)$ and from the definition of *mtrace*, *trace*$(\mathit{lpbody}(c, r), s, t)$ is *true*. From Lemma (D.29) it follows that *trace*(p, s, t) is also *true*.

final?$(p - \mathit{tset}(p,s,t))(t)$: From the assumption, *mtrace*$(\mathit{lpbody}(c,r),s,t)$, *final?*$(\mathit{lpbody}(c,r) - \mathit{tset}(\mathit{lpbody}(c,r),s,t))$ is *true* and there is no $c_1 \in \mathit{lpbody}(c,r) - \mathit{tset}(\mathit{lpbody}(c,r),s,t)$ such that *enabled*$(c_1)(t)$.

Assume there is a command $c_2 \in p - \mathit{tset}(p,s,t)$ such that *enabled*$(c_2)(s)$ and $c_2 \in p$. From $c_2 \in p$ and from Corollary (4.3), $c_2 \in \mathit{lpbody}(c,r)$). From the assumption *final?*$(\mathit{lpbody}(c,r) - \mathit{tset}(\mathit{lpbody}(c,r),s,t))(t)$ and *enabled*$(c_2)(t)$ it follows that $c_2 \in \mathit{tset}(\mathit{lpbody}(c,r))),s,t)$. From Lemma (D.32), it also follows that $c_2 \in \mathit{tset}(p,s,t)$ contradicting the assumption $c_2 \in p - \mathit{tset}(p,s,t)$. Therefore there can be no command in $p - \mathit{tset}(p,s,t)$ which is enabled in t and the proof of *final?*$(p - \mathit{tset}(p,s,t))(t)$ is complete. □

For cut-point c of region r, a maximal trace through *lpbody*(c,r) is equivalent to a maximal trace through r restricted in the loop heads of r, *rmtrace*$(\mathit{lpheads}(r))$.

Lemma D.40 *For* $r \in \mathcal{R}$, $c \in \mathcal{L}$ *and* $s,t \in \mathit{State}$,

$$\frac{\mathit{mtrace}(\mathit{lpbody}(c,r),s,t) \quad \mathit{enabled}(c)(s) \quad c \in \mathit{cuts}(r)}{\mathit{rmtrace}(\mathit{lpheads}(r))(\mathit{body}(r),s,t)}$$

Proof.

Straightforward from definitions. Note that $\mathit{lpbody}(c,r) = (\mathit{body}(r) - \mathit{lpheads}(r)) \cup \{c\}$. □

Lemma D.41 *For* $r \in \mathcal{R}$, $c \in \mathcal{L}$ *and* $s,t \in \mathit{State}$,

$$\frac{\mathit{rmtrace}(\mathit{lpheads}(r))(\mathit{body}(r),s,t) \quad \mathit{enabled}(c)(s) \quad c \in \mathit{cuts}(r)}{\mathit{mtrace}(\mathit{lpbody}(c,r),s,t)}$$

Proof. Straightforward from definitions. □

If a transformed region $T_2(r)$ produces a state s from state t then the transitive closure of *rmtrace*$(\mathit{lpheads}(r))$ is *true* between s and t in *body*(r).

Lemma D.42 *For* $r \in \mathcal{R}$ *and* $s,t \in \mathit{State}$,

$$\frac{s \overset{T_2(r)}{\leadsto} t}{\mathit{rmtrace}^{+}(\mathit{lpheads}(r))(\mathit{body}(r),s,t)}$$

Proof.

By right induction on $s \overset{T_2(r)}{\leadsto} t$.

Case $c \in \mathit{body}(T_2(r))$ and $s \overset{c}{\leadsto} t$: By definition of T_2, there is a region $r' \in \mathit{loops}(r)$ such $c = T_1(r')$. From Theorem (4.16), *single?*(r') and from Theorem (D.17), *mtrace*$(\mathit{body}(r'),s,t)$

is *true*. From the definitions of *region* and *lpbody*, there is a $c' \in \mathit{cuts}(r)$ such that $r' = \mathit{region}(\mathit{label}(c'), \mathit{lpbody}(c', r))$ and $\mathit{label}(\mathit{head}(r')) = \mathit{label}(c')$. Since $T_1(r')$ is enabled in s and $\mathit{label}(T_1(r')) = \mathit{label}(\mathit{head}(r'))$, c' is also enabled in s. By Lemma (D.38) it follows that $\mathit{mtrace}(\mathit{lpbody}(c', r), s, t)$. From this and Lemma (D.40) the conclusion is immediate.

Case $c \in \mathit{body}(r)$ and $u \in \mathit{State}$, $s \overset{T_2(r)}{\leadsto} u$, $u \overset{c}{\leadsto} t$ and the property holds for $s \overset{T_2(r)}{\leadsto} u$: From the inductive hypothesis, $\mathit{rmtrace}^+(\mathit{lpheads}(r))(\mathit{body}(r), s, u)$ is *true* and all that is required is to show $\mathit{rmtrace}(\mathit{lpheads}(r))(\mathit{body}(r), u, t)$. The proof for this is the same as for the previous case. □

Lemma D.43 *For* $r \in \mathcal{R}$ *and* $s, t \in \mathit{State}$,

$$\frac{s \overset{T_2(r)}{\leadsto} t}{s \overset{r}{\leadsto} t}$$

Proof.

By induction on $s \overset{T_2(r)}{\leadsto} t$. The inductive case is immediate from the inductive hypothesis and the definition of transitive closure.

Base case: $c \in T_2(r)$ and $s \overset{c}{\leadsto} t$. From $c \in T_2(r)$ and from the definitions, there is a sub-region $r' \in \mathit{loops}(r)$ such that $c = T_1(r)$. From Theorem (4.16), $\mathit{single?}(r')$ and the proof is immediate from $\mathcal{I}(T_1(r'))(s, t)$ and Theorem (D.15). □

If there is a restricted maximal trace through a region to a state t then any command enabled in t is the head of a loop in r.

Lemma D.44 *For* $r \in \mathcal{R}$, $c_1 \in \mathcal{L}$, $s, t \in \mathit{State}$,

$$\frac{\mathit{rmtrace}(\mathit{lpheads}(r))(\mathit{body}(r), s, t) \quad \mathit{enabled}(c)(s) \quad c \in \mathit{cuts}(r)}{\mathit{final?}(\mathit{body}(r))(t) \lor (\exists c_1 : c_1 \in \mathit{lpheads}(r) \land \mathit{enabled}(c_1)(t))}$$

Proof.

Assume there is a command $c_2 \in \mathit{body}(r)$ such that $\mathit{enabled}(c_2)(t)$. Either $c_2 \in \mathit{lpheads}(r)$ and the proof is complete or $c_2 \notin \mathit{lpheads}(r)$ and $c_2 \in \mathit{tset}(\mathit{body}(r), s, t)$ (definition of *rmtrace*). Assume $c_2 \notin \mathit{lpheads}(r)$ and $c_2 \in \mathit{tset}(\mathit{body}(r), s, t)$. By definition, $\mathit{mtrace}(\mathit{body}(r), s, t)$ is *true*. If $c = c_2$ then the conclusion is immediate, assume $c \neq c_2$.

From Theorem (D.4) there is a $u \in \mathit{State}$ and a set $p \subseteq \mathit{body}(r)$ such that $\mathit{enabled}(c_2)(u)$, $\mathit{trace}(\mathit{body}(r) - p \cup \{c_2\}, u, t)$ and $\mathit{trace}(p, s, u)$. Since $\mathit{enabled}(c)(s)$ and $\mathit{enabled}(c_2)(t)$ and from Lemma (D.3) and Theorem (4.8), $c \overset{p}{\longmapsto} c_2$ and $c_2 \overset{(\mathit{body}(r)-p)\cup\{c_2\}}{\longmapsto} c_2$. By definition, $\mathit{lpbody}(c, r) = \mathit{body}(r) - \mathit{lpheads}(r) \cup \{c\}$ and $\mathit{lphead?}(c_2, \mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r)))$ is *true*. Assume $c = \mathit{head}(r)$, then $\mathit{lphead?}(c_2, r)$ is immediate and $c_2 \in \mathit{lpheads}(r)$ completing the proof for this case.

Assume $c \neq head(r)$ and $c \in lpheads(r)$. By definition, $region(label(c), lpbody(c,r)) \in loops(r)$ and, from Lemma (D.34), $lphead?(c_2, r)$ is *true*. Therefore $c_2 \in lpheads(r)$ which is a contradiction. □

Lemma (D.44) can be extended to the transitive closure of a restricted maximal trace.

Lemma D.45 *For $r \in \mathcal{R}$, $c_1 \in \mathcal{L}$, $s, t \in State$,*

$$\frac{rmtrace^{+}(lpheads(r))(body(r), s, t) \quad enabled(c)(s) \quad c \in cuts(r)}{final?(body(r))(t) \vee (\exists c_1 : c_1 \in lpheads(r) \wedge enabled(c_1)(t))}$$

Proof.

Straightforward, by induction on the transitive closure of *rmtrace* and from Lemma (D.44). □

A consequence of Lemma (D.45) is that a final state produced by a transformed region $T_2(r)$ is also final for the region r.

Lemma D.46 *For $r \in \mathcal{R}$ and $s, t \in State$,*

$$\frac{enabled(head(T_2(r)))(s) \quad s \overset{T_2(r)}{\leadsto} t \quad final?(T_2(r))(t)}{final?(r)(t)}$$

Proof.

From the definitions, $label(T_2(r)) = label(r)$ and $label(r) = label(head(r))$. It follows that $enabled(T_2(r))(s)$ iff $enabled(head(r))(s)$. From the assumption $s \overset{T_2(r)}{\leadsto} t$ and Lemma (D.42), $rmtrace^{+}(lpheads(r))(body(r), s, t)$ is *true*. From Lemma (D.45), there are two cases: either $final?(body(r))(t)$ or there is a command $c \in lpheads(r)$ such that $enabled(c)(t)$. If it is the case $final?(body(r))(t)$ then $final?(r)(t)$ is immediate by definition.

Assume $c \in lpheads(r)$ and $enabled(c)(t)$. Then there is a $r' \in loops(r)$ such that $r' = region(label(c), lpbody(c,r))$ and $enabled(head(r'))(t)$. Region r' is in $gtbody(r)$ by definition. By Corollary (4.3), $body(T_2(r)) \subseteq gtbody(r)$ and, from $s \overset{T_2(r)}{\leadsto} t$ and Lemma (4.2), $s \overset{gtbody(r)}{\leadsto} t$. Since $enabled(head(T_2(r))(s)$, $enabled(T_1(r))(t)$, $s \overset{gtbody(r)}{\leadsto} t$ and, by Lemma (4.8), it follows that $head(T_2(r)) \overset{gtbody(r)}{\longmapsto} T_1(r')$. Therefore, by definition of *region*, $T_1(r') \in body(T_2(r))$ and, since $enabled(T_1(r'))(t)$, the assumption that t is final for $T_2(r)$ is contradicted. □

Proof. *Theorem (4.17)*

From the definition of $\sqsubseteq$, there are three properties to be proved

1. $label(T_2(r)) = label(r)$. Immediate from definition of T_2 and *region*.

2. $\mathit{body}(T_2(r)) \sqsubseteq \mathit{body}(r)$. The proof required is that for any $s, t \in \mathit{State}$, $s \overset{T_2(r)}{\leadsto} t \Rightarrow s \overset{r}{\leadsto} t$. This is immediate from Lemma (D.43).

3. For $s, t \in \mathit{State}$, if $\mathit{enabled}(T_2(r))(s)$, $s \overset{T_2(r)}{\leadsto} t$ and $\mathit{final?}(T_2(r))(t)$ are *true*, then so is $\mathit{final?}(r)(t)$. This is immediate from $\mathit{label}(r) = \mathit{label}(\mathit{head}(r))$, $\mathit{label}(T_2(r)) = \mathit{label}(r)$ and Lemma (D.46).

□

D.11.4 Theorem (4.18)

The proof of Theorem (4.18) is based on showing that the transitive closure of a restricted maximal trace through a region r is equivalent to the *leads-to* relation through the transformed region $T_2(r)$. The proof is in a number of steps which establish a relationship between the *leads-to* relation through r and the transitive closure of *rmtrace* through r.

Lemma D.47 *For* $r \in \mathcal{R}$, $c, c_1 \in \mathcal{L}$, $s, t, u \in \mathit{State}$,

$$\frac{s \overset{r}{\leadsto} t \quad c \in \mathit{cuts}(r) \quad \mathit{enabled}(c)(s)}{\begin{array}{l}\mathit{trace}(\mathit{lpbody}(c, r)) \\ \vee(\exists c_1, u : \mathit{rmtrace}^+(\mathit{lpheads}(r))(\mathit{body}(r), s, u) \\ \qquad\qquad \wedge\, \mathit{trace}(\mathit{lpbody}(c, r), u, t) \wedge \mathit{enabled}(c_1)(u) \wedge c_1 \in \mathit{lpheads}(r)\end{array}}$$

Proof.

By right induction on $s \overset{r}{\leadsto} t$. The proof for two cases are similar and only the inductive case is considered.

Assume $t_1 \in \mathit{State}$ and $c_2 \in r$ such that $s \overset{r}{\leadsto} t_1$ and $t_1 \overset{c_2}{\leadsto} t$. The inductive hypothesis is that either $\mathit{trace}(\mathit{lpbody}(c, r), s, t_1)$ or there is a state u and command $c_3 \in \mathit{lpheads}(r)$ such that $\mathit{rmtrace}^+(\mathit{lpheads}(r))(\mathit{body}(r), s, u)$, $\mathit{trace}(\mathit{lpbody}(c, r), u, t_1))$ and $\mathit{enabled}(c_3)(u)$. Assume the second case, the proof is similar for the first.

Assume $c_2 \notin \mathit{tset}(\mathit{lpbody}(c, r), u, t_1)$. By Lemma (D.9) and the assumption $t_1 \overset{c_2}{\leadsto} t$ it follows that $\mathit{trace}(\mathit{lpbody}(c, r), u, t)$ is *true*. This completes the proof for this case.

Assume $c_2 \in \mathit{tset}(\mathit{lpbody}(c, r), u, t_1)$. By definition, $\mathit{mtrace}(\mathit{lpbody}(c, r), s, t)$ is *true* and, by Lemma (D.40) and $c_2 \in \mathit{lpheads}(r)$, it follows that $\mathit{rmtrace}(\mathit{lpheads}(r))(\mathit{lpbody}(c, r), s, t)$ is also *true*. This leads to the conclusion that there is a state, t_1, and command, c_2, such that $\mathit{rmtrace}^+(\mathit{lpheads}(r))(\mathit{body}(r), s, t_1)$ and $\mathit{enabled}(c_2)(t_1)$. By Lemma (D.45) it follows that $c_2 \in \mathit{lpheads}(r)$ and $\mathit{trace}(\mathit{lpbody}(c_2, r), t_1, t)$ follows immediately from $t_1 \overset{c_2}{\leadsto} t$ and the definitions, completing the proof. □

When the state produced by a region r is final for r, Lemma (D.47) can be described in terms of maximal traces only.

Lemma D.48 *For $r \in \mathcal{R}$, $c, c_1 \in \mathcal{L}$, $s, t, u \in State$,*

$$\frac{s \overset{r}{\leadsto} t \quad c \in \mathit{cuts}(r) \quad \mathit{enabled}(c)(s) \quad \mathit{final?}(r)(t)}{\begin{array}{l}\mathit{mtrace}(\mathit{lpbody}(c, r)) \\ \vee(\exists c_1, u : \mathit{rmtrace}^+(\mathit{lpheads}(r))(\mathit{body}(r), s, u) \\ \qquad \wedge\ \mathit{mtrace}(\mathit{lpbody}(c, r), u, t) \wedge \mathit{enabled}(c_1)(u) \wedge c_1 \in \mathit{lpheads}(r)\end{array}}$$

Proof.

Straightforward from Lemma (D.47) since for any program p and states s, t, $\mathit{trace}(p, s, t) \wedge \mathit{final?}(p)(t)$ immediately lead to $\mathit{mtrace}(p, s, t)$. □

For region r, the *leads-to* relation through $T_2(r)$ follows from the transitive closure of the restricted maximal trace, $\mathit{rmtrace}(\mathit{lpheads}(r))$, through r.

Lemma D.49 *For $r \in \mathcal{R}$, $s, t \in State$,*

$$\frac{\mathit{enabled}(\mathit{head}(r))(s) \quad \mathit{rmtrace}^+(\mathit{lpheads}(r))(\mathit{body}(r), s, t)}{s \overset{T_2(r)}{\leadsto} t}$$

Proof.

By right induction on the transitive closure of *rmtrace*. The proof for the base and inductive cases are similar and only the inductive case is considered.

For the inductive case, assume a state u such that both $\mathit{rmtrace}^+(\mathit{lpheads}(r))(\mathit{body}(r), s, u)$ and $\mathit{rmtrace}(\mathit{lpheads}(r))(\mathit{body}(r), u, t)$ are *true*. Also assume that the property holds between s and u, $s \overset{T_2(r)}{\leadsto} u$. From the definition of *leads-to*, all that is required is to show that there is a $c \in T_2(r)$ such that $\mathcal{I}(c)(u, t)$.

From Lemma (D.45), either $\mathit{final?}(r)(u)$ or there is a $c_1 \in \mathit{lpheads}(r)$ such that $\mathit{enabled}(c_1)(u)$. $\mathit{final?}(\mathit{body}(r))(u)$ contradicts the assumption that there is a trace beginning at u, therefore there is a command $c_1 \in \mathit{lpheads}(r)$ enabled in u. By Lemma (D.41) and the assumption $\mathit{rmtrace}(\mathit{lpheads}(r))(\mathit{body}(r), u, t)$ it follows that $\mathit{mtrace}(\mathit{lpbody}(c_1, r), u, t)$. Let r' be the region constructed from $\mathit{lpbody}(c_1, r)$ beginning with c_1, $r' = \mathit{region}(\mathit{label}(c_1), \mathit{lpbody}(c_1, r))$. Since $c_1 \in \mathit{lpheads}(r)$, by definition $r' \in \mathit{loops}(r)$ and, by Lemma (D.39), $\mathit{mtrace}(\mathit{body}(r'), u, t)$ is *true*. From the inductive hypothesis, $s \overset{T_2(r)}{\leadsto} u$, and from the assumption, $\mathit{enabled}(c_1)(u)$ and therefore $\mathit{enabled}(r')(u)$. Also from the assumptions, $\mathit{enabled}(\mathit{head}(r))(s)$ and therefore $\mathit{head}(T_2(r))$ reaches $T_1(r')$ in $\mathit{gtbody}(r)$ and from the definition of T_2 and *region*, $T_1(r') \in T_2(r)$.

Since $r' \in \mathit{loops}(r)$, $\mathit{single?}(r)$ follows from Theorem (4.16). From $\mathit{enabled}(c_1)(u)$, and since $\mathit{head}(r') = c_1$, $\mathit{enabled}(\mathit{head}(r'))(u)$ is *true*. Therefore, from $\mathit{mtrace}(\mathit{body}(r'), u, t)$ and Theorem (D.14), it follows that $\mathcal{I}(T_1(r'))(u, t)$ is *true*, completing the proof. □

The property of Lemma (D.49) can be re-stated in terms of the *leads-to* relation through region r when r produces a final state.

Lemma D.50 *For $r \in \mathcal{R}$, $s, t \in State$,*

$$\frac{\textit{enabled}(\textit{head}(r))(s) \quad s \overset{r}{\leadsto} t \quad \textit{final?}(r)(t)}{s \overset{T_2(r)}{\leadsto} t}$$

Proof.

From Lemma (D.48), and from $\textit{head}(r) \in \textit{cuts}(r)$, there are two cases. The proofs for both are similar and only the second case is considered.

Assume command $c \in \textit{lpheads}(r)$ and $u \in \textit{State}$ such that c is enabled in u, $\textit{enabled}(c)(u)$, and $\textit{rmtrace}^{+}(\textit{lpheads}(r))(\textit{body}(r), s, u)$ and $\textit{mtrace}(\textit{lpbody}(c, r), u, t)$ are *true*. By definition, $\textit{mtrace}(\textit{lpbody}(c, r), u, t)$ is equivalent to $\textit{rmtrace}(\textit{lpheads}(r))(\textit{body}(r), u, t)$. Together with the assumption $\textit{rmtrace}^{+}(\textit{lpheads}(r))(\textit{body}(r), s, u)$, this is enough to establish the transitive closure of the restricted maximal trace between s and t, $\textit{rmtrace}^{+}(\textit{lpheads}(r))(\textit{body}(r), s, t)$, and the conclusion is immediate from Lemma (D.49). □

Proof. *Theorem (4.18)*

(⇒): $\mathcal{I}(T_2(r))(s, t) \Rightarrow \mathcal{I}(r)(s, t)$ is straightforward from $T_2(r) \sqsubseteq r$ (Theorem 4.17).

(⇐): By definition and the assumption $\mathcal{I}(r)(s, t)$, $\textit{enabled}(r)(s)$, $s \overset{r}{\leadsto} t$ and $\textit{final?}(r)(t)$ are *true*. By definition, $\textit{label}(T_2(r)) = \textit{label}(r)$ and $\textit{enabled}(r)(s)$ iff $\textit{enabled}(T_2(r))(s)$.

From the assumption $\textit{enabled}(r)(s)$, it follows that $\textit{enabled}(\textit{head}((r))(s)$ is *true*. From $s \overset{r}{\leadsto} t$, $\textit{final?}(r)(t)$ and Lemma (D.50), the conclusion $s \overset{T_2(r)}{\leadsto} t$ is immediate.

From $\textit{final?}(r)(t)$, $\textit{final?}(T_2(r))(t)$ is straightforward since for every command $c \in T_2(r)$, there is a command $c' \in r$ such that $\textit{label}(c') = \textit{label}(c)$ (definition of $\textit{gtbody}(r)$). If $\textit{final?}(T_2(r))(t)$ is *false* then there is a command $c' \in T_2(r)$ such that $\textit{enabled}(c')(t)$. Therefore there is a command $c \in r$ such that $\textit{enabled}(c)(t)$ contradicting the assumption that $\textit{final?}(r)(t)$. □

D.11.5 Theorem (4.19)

The proof of Theorem (4.19) is based on the behaviour of a region r and the transformed region $T_2(r)$ when commands of either halt in a state.

If a command of a transformed region $T_2(r)$ halts in a state then so does the region r.

Lemma D.51 *For $r \in \mathcal{R}$, $c \in \mathcal{L}$ and $s \in State$,*

$$\frac{c \in T_2(r) \quad \textit{halt?}(c)(s)}{\textit{halt?}(r)(s)}$$

Proof.

From the definition of T_2, there is a command $c_1 \in r$ and region $r' \in$ *loops*(r) such that $c = T_1(r')$ and $r' =$ *region*(*label*(c_1), *lpbody*(c_1, r)). From Theorem (4.16), *single?*(r') and by Lemma (D.26), there are two cases to consider.

Case halt?(*head*(r'))(s). Since *head*$(r') = c_1$, by definition of *region*, and since $c_1 \in r$, it follows, from the definitions, that *halt?*(*body*(r))(s) and *halt?*$(r)(s)$ are *true*.

Case there is a state t such that *trace*(*body*(r'), s, t) and *halt?*(*body*$(r')(t)$). From Lemma (D.3), $s \overset{r'}{\leadsto} t$. By definition of *lpbody*(c_1, r) and *region*, *body*$(r') \subseteq$ *body*(r). From this together with Lemma (4.2) it follows that $s \overset{r}{\leadsto} t$. Since *halt?*(*body*$(r')$)$(t)$, there is a $c_3 \in$ *body*(r') such that for any state t_1, $\neg\mathcal{I}(c_3)(t, t_1)$. Since *body*$(r') \subseteq$ *body*(r), $c_3 \in$ *body*(r) and *halt?*(*body*(r))(t) is proved. *halt?*$(r)(s)$ follows from the definition of $s \overset{r}{\leadsto} t$ and *halt?*. □

If a transformed region $T_2(r)$ fails in a state then so does the region r.

Lemma D.52 *For $r \in \mathcal{R}$ and $s \in$ State,*

$$\frac{\textit{halt?}(T_2(r))(s)}{\textit{halt?}(r)(s)}$$

Proof.

From the definition of *halt?*$(T_2(r))(s)$, there are two cases.

Case halt?(*body*$(T_2(r))$)(s). By definition there is a command $c \in T_2(r)$ and *halt?*$(c)(s)$. The conclusion follows immediately from Lemma (D.51).

Case there is a state t such that $s \overset{T_2(r)}{\leadsto} t$ and *halt?*(*body*$(T_2(r))$)(t). From Lemma (D.43), $s \overset{r}{\leadsto} t$. From Lemma (D.51), *halt?*$(r)(t)$ and, by definition, there are two cases. Assume *halt?*(*body*(r))(t) then the conclusion is immediate from $s \overset{r}{\leadsto} t$ and the definition of *halt?*$(r)(s)$. Assume there is a state u such that $t \overset{r}{\leadsto} u$ and *halt?*(*body*(r))(u). From $s \overset{r}{\leadsto} t$ and the definition of *leads-to*, it follows that $s \overset{r}{\leadsto} u$. Since $s \overset{r}{\leadsto} u$ and *halt?*(*body*(r))(u), the conclusion is immediate from the definition of *halt?*$(r)(s)$. □

Lemma (D.52) establishes that *halt?*$(T_2(r))(s) \Rightarrow$ *halt?*$(r)(s)$. The reverse, *halt?*$(r)(s) \Rightarrow$ *halt?*$(T_2(r))$, is established by considering the commands of r which halts in s.

If the head of region r fails in a state s then so does the transformed region $T_2(r)$.

Lemma D.53 *For $r \in \mathcal{R}$ and $s \in$ State,*

$$\frac{\textit{halt?}(\textit{head}(r))(s)}{\textit{halt?}(T_2(r))(s)}$$

Proof.

By definition, there is a region $r' \in \mathit{loops}(r)$ beginning with the head of r and constructed from $\mathit{lpbody}(\mathit{head}(r), r)$, $r' = \mathit{region}(\mathit{label}(\mathit{head}(r)), \mathit{lpbody}(\mathit{head}(r), r))$. The head of r' is $\mathit{head}(r)$ and, by definition of *region* and $\mathit{label}(r') = \mathit{label}(r)$, $T_1(r') \in T_2(r)$. By Theorem (4.16), $\mathit{single?}(r')$ and, by Lemma (D.24), $\mathit{halt?}(T_1(r'))(s)$ is immediate. By definition it follows that $\mathit{halt?}(\mathit{body}(T_2(r)))(s)$ is *true* and the conclusion $\mathit{halt?}(T_2(r))(s)$ is immediate from the definitions. □

If a region r is enabled and begins in a state s to produce a state t in which it fails then the transformed region fails in state s.

Lemma D.54 *For* $r \in \mathcal{R}$ *and* $s \in \mathit{State}$,

$$\frac{\mathit{enabled}(\mathit{head}(r))(s) \quad s \overset{r}{\leadsto} t \quad \mathit{halt?}(\mathit{body}(r))(t)}{\mathit{halt?}(T_2(r))(s)}$$

Proof.

From $\mathit{enabled}(\mathit{head}(r))(s)$, $s \overset{r}{\leadsto} t$ and Lemma (D.47), there are two cases. The proofs for both cases are similar and the more general is considered.

Assume $c_1 \in \mathcal{L}$ and $u \in \mathit{State}$ such that

$$\begin{array}{l}\mathit{enabled}(c_1)(u) \wedge c_1 \in \mathit{lpheads}(r) \\ \wedge \mathit{rmtrace}^{+}(\mathit{lpheads}(r))(\mathit{body}(r), s, u) \wedge \mathit{trace}(\mathit{lpbody}(c_1, r), u, t)\end{array}$$

From the assumptions $\mathit{enabled}(\mathit{head}(r))(s)$ and $\mathit{rmtrace}^{+}(\mathit{lpheads}(r))(\mathit{body}(r), s, u)$, it follows, from Lemma (D.49), that $s \overset{T_2(r)}{\leadsto} u$. From the assumption $\mathit{halt?}(\mathit{body}(r))(t)$, there is a command $c' \in r$ such that $\mathit{enabled}(c')(t)$ and $\forall t_1 : \neg\mathcal{I}(c')(t, t_1)$.

Case $c' \in \mathit{lpbody}(c_1, r)$. Let $r' = \mathit{region}(\mathit{label}(c_1), \mathit{lpbody}(c_1, r))$. Since $c_1 \in \mathit{lpheads}(r)$, it follows that $r' \in \mathit{loops}(r)$. From Lemma (D.3) and $\mathit{trace}(\mathit{lpbody}(c_1, r), u, t)$ it follows that $u \overset{\mathit{lpbody}(c_1,r)}{\leadsto} t$. Since $\mathit{enabled}(c_1)(s)$ and $\mathit{enabled}(c')(u)$, and from Theorem (4.8), c_1 reaches c' in $\mathit{lpbody}(c_1, r)$, $c_1 \overset{\mathit{lpbody}(c_1,r)}{\longmapsto} c'$, and $c' \in r'$, by definition of *region*.

From $\mathit{halt?}(c')(t)$ and $c' \in r'$, it follows that $\mathit{halt?}(\mathit{body}(r'))(t)$. From Lemma (D.29), the assumption $\mathit{trace}(\mathit{lpbody}(c_1, r), u, t)$ leads to $\mathit{trace}(\mathit{body}(r'), u, t)$. From this and Lemma (D.25), either $\mathit{halt?}(T_1(r'))(u)$ or $\mathit{enabled}(\mathit{head}(r'))(t)$ are *true*.

Assume $\mathit{enabled}(\mathit{head}(r'))(t)$. Since $c' \in r'$ and $\mathit{enabled}(c')(t)$, it follows that $\mathit{head}(r') = c'$. Since $\mathit{head}(r') = c_1$ and there is a trace from state u through $\mathit{lpbody}(c_1, r)$, it follows that $c_1 \in \mathit{tset}(\mathit{lpbody}(c, r), u, t)$. By definition, this leads to $\mathit{mtrace}(\mathit{lpbody}(c, r), u, t)$ and, by Lemma (D.39), it follows that $\mathit{mtrace}(\mathit{body}(r'), u, t)$. Since $r' \in \mathit{loops}(r)$, by Theorem (4.16), $\mathit{single?}(r')$ and by Theorem (D.14), $\mathcal{I}(r')(u, t)$. By Lemma (D.49) and from the assumptions, $s \overset{T_2(r)}{\leadsto} u$ is *true*. From the definitions, $\mathit{enabled}(\mathit{head}(T_2(r)))(s)$ and from $\mathit{enabled}(c_1)(u)$ and

Theorem (4.8) and the definitions, it follows that $head(T_2(r)) \overset{gtbody(r)}{\longmapsto} T_1(r')$. Therefore $T_1(r') \in T_2(r)$ and $s \overset{T_2(r)}{\leadsto} t$.

Since $c' = c_1$ and $enabled(c')(t)$ (from $halt?(c')(t)$), it follows that $enabled(T_1(r'))(t)$ (since $c_1 = head(r')$). Assume that there is a state t_1 such that $\mathcal{I}(T_1(r'))(t, t_1)$. By Theorem (D.15), it follows that $t \overset{r'}{\leadsto} t_1$ and, since $r' \subseteq r$, this leads to $t \overset{r}{\leadsto} t_1$. By definition of *leads-to* and since $enabled(c')(t)$, there is a state t_2 such that $\mathcal{I}(c')(t, t_2)$ contradicting the assumption $halt?(r)(t)$. Therefore $T_1(r')$ fails in t, $halt?(T_1(r'))(t)$ and $halt?(T_2(r))(s)$ follows from the definition, from $s \overset{T_2(r)}{\leadsto} t$ and from $T_1(r') \in T_2(r)$.

Assume $halt?(T_1(r'))(u)$. As before, $s \overset{T_2(r)}{\leadsto} u$ and $head(T_2(r)) \overset{gtbody(r)}{\longmapsto} head(r')$ and it follows that $T_1(r') \in T_2(r)$. From $halt?(T_1(r'))(u)$, it follows that $halt?(body(T_2(r)))(u)$ and $halt?(T_2(r))(s)$ is straightforward from the definitions.

Case $c' \notin lpbody(c_1, r)$. It follows that $final?(lpbody(c_1, r) - tset(lpbody(c_1, r), u, t))(t)$ is *true* and therefore so is $mtrace(lpbody(c_1, r), u, t)$. Let $r_1 = region(lpbody(c_1, r), u,)$. By definition, $r_1 \in loops(r)$ and, by Theorem (4.16), $single?(r)$. By Lemma (D.39), $mtrace(body(r_1), u, t)$ and, since $c_1 = head(r_1)$, by Lemma (D.14), $\mathcal{I}(T_1(r_1))(u, t)$. As before, $head(T_2(r)) \overset{gtbody(r)}{\longmapsto} T_1(r_1)$ and $T_1(r_1) \in T_2(r)$. It follows that $s \overset{T_2(r)}{\leadsto} u$.

From the assumptions, $c' \in lpheads(r)$ and $enabled(c')(t)$ and begins a sub-region r' of r. Let $r' = region(label(c'), lpbody(c', r))$. By definition, $r' \in loops(r)$ and $T_1(r') \in gtbody(r)$. Since $s \overset{T_2(r)}{\leadsto} t$ and $body(T_2(r)) \subseteq gtbody(r)$, it follows that $head(T_2(r)) \overset{gtbody(r)}{\longmapsto} T_1(r')$ and $T_1(r') \in T_2(r)$. Furthermore, $head(r') = c'$ and, since $halt?(c')(t)$, it follows, by Lemma (D.24), that $halt?(T_1(r'))(t)$. Therefore, $halt?(body(T_2))(t)$ is *true* and $halt?(T_2(r))(t)$ is straightforward from the definitions. □

Lemma (D.53) and Lemma (D.54) complete the proof of the equivalence of failures of a region r and the transformed region $T_2(r)$. As a consequence, the proof of Theorem (4.19) is straightforward from the definitions and the assumption that the regions are enabled in the state in which they fail.

Proof. *Theorem (4.19)*

Since $label(r) = label(T_2(r))$, by definition $enabled(r)(s) \iff enabled(T_2(r))(s)$.

($\Rightarrow$): Immediate from Lemma (D.52).

($\Leftarrow$): There are two cases: Assume $halt?(body(r))(s)$. Then there is a $c \in r$, such that $halt?(c)(s)$ and, therefore, $enabled(c)(s)$. Since $enabled(r)(s)$ and $label(r) = label(head(r))$ it follows that $c = head(r)$. From Lemma (D.53), $halt?(head(r))(s) \Rightarrow halt?(T_2(r))(s)$ and the proof is complete for this case.

Assume $t \in State$ such that $s \overset{r}{\leadsto} t$ and $halt?(body(r))(t)$. The proof for this case is immediate from Lemma (D.54) and completes the proof for the Theorem. □

D.11.6 Theorem (4.20)

Theorem (4.20) is a syntactic property of transformation T_2. The proof is based on the fact that the only cut-point of a single loop r is the head of r.

Lemma D.55 *For $r \in \mathcal{R}$,*

$$\frac{\mathit{single?}(r)}{\mathit{cuts}(r) \subseteq \{\mathit{head}(r)\}}$$

Proof.

Assume there is a cut-point $c \in \mathit{cuts}(r)$ such that $c \neq \mathit{head}(r)$. By definition c must be the head of a loop in r, $\mathit{lphead?}(c, r)$. Since $c \neq \mathit{head}(r)$, there are sets a and b such that $a \subseteq \mathit{body}(r)$ and $b = (\mathit{body}(r) - a) \cup \{c\}$. The head of r reaches c through a, $\mathit{head}(r) \overset{a}{\longmapsto} c$ and therefore $\mathit{head}(r) \in a$ by definition of *reaches*. Since $\mathit{head}(r) \neq c$ and $\mathit{head}(r) \in a$, $\mathit{head}(r) \notin (\mathit{body}(r) - a) \cup \{c\}$. From $\mathit{head}(r) \notin b$, $b \subseteq \mathit{body}(r) - \{\mathit{head}(r)\}$. Since the command c forms a loop in b, $c \overset{b}{\longmapsto} c$, it follows, by Lemma (4.7), that $c \overset{\mathit{body}(r) - \{\mathit{head}(r)\}}{\longmapsto} c$. This contradicts the assumption $\mathit{single?}(r)$.
□

Since the only cut-point of a single loop r is the head of r, the result of $\mathit{lpbody}(c, r)$, for $c \in \mathit{cuts}(r)$, is the body of r. Consequently, every region formed from $\mathit{lpbody}(c, r)$ is r.

Lemma D.56 *For $r \in \mathcal{R}$ and $c \in \mathcal{L}$,*

$$\frac{\mathit{single?}(r) \quad c \in \mathit{cuts}(r)}{\mathit{region}(\mathit{label}(c), \mathit{lpbody}(c, r)) = r}$$

Proof.

From Lemma (D.55), if there is a command $c \in \mathit{cuts}(r)$ then $c = \mathit{head}(r)$. From the definition of $\mathit{cuts}(r)$, the result of $\mathit{lpheads}(r) \cup \{\mathit{head}(r)\}$ is $\{\mathit{head}(r)\}$. Therefore the result of $(\mathit{body}(r) - \mathit{lpheads}(r)) + \mathit{head}(r)$ is $\mathit{body}(r)$. By definition, $\mathit{label}(\mathit{head}(r)) = \mathit{label}(r)$ and, by definition of *region*, $\mathit{region}(\mathit{label}(r), \mathit{body}(r)) = r$. □

Proof. *Theorem (4.20)*

From the assumption $\mathit{single?}(r)$, by Lemma (D.56) and by definition, $\mathit{loops}(r) = \{r\}$ and $\mathit{gtbody}(r) = \{T_1(r)\}$. Since $\mathit{label}(T_1(r)) = \mathit{label}(r)$, the result of $\mathit{region}(\mathit{label}(r), \mathit{gtbody}(r))$ is $(\mathit{label}(r), \{T_1(r)\})$. This is also the result of $\mathit{unit}(T_1(r))$ and the proof is complete. □

Appendix E

Proofs: Proof Methods for Object Code

This appendix contains proofs of the theorems and lemmas of Chapter 6. The order in which the proofs and definitions are presented follows that of Chapter 6.

E.1 Theorem (6.3)

Lemma E.1 *For any $p \in \mathcal{P}$, $I, A, B, P, Q \in \mathcal{A}$, $c \in \mathcal{C}$ and $s, t \in$ State,*

$$\frac{s \overset{p}{\leadsto} t \quad (\forall c \in p : (\vdash A \Rightarrow \mathbf{wp}(c, B)) \Rightarrow (\vdash I \land A \Rightarrow \mathbf{wp}(c, I \land B))) \quad (I \land P)(s) \quad Q(t)}{I(t)}$$

Proof. By right induction on $s \overset{p}{\leadsto} t$.

Case $c_1 \in p$ and $s \overset{c_1}{\leadsto} t$. By definition of $s \overset{c_1}{\leadsto} t$, $\mathcal{I}_c(c_1)(s, t)$ is *true*. From $Q(t)$, it follows that $\mathbf{wp}(c, Q)(s)$ is *true*. From the assumptions, this establishes $\mathbf{wp}(c, I \land Q)(s)$ and, as a property of **wp**, this establishes a state t' such that $\mathcal{I}_c(c_1)(s, t')$ and $(Q \land I)(t')$. Since the commands are deterministic, it follows that $t' = t$ and $I(t)$ follows immediately.

Case $c_1 \in p$, $u \in$ *State*, $s \overset{p}{\leadsto} u$ and $u \overset{c_1}{\leadsto} t$. As before, since $\mathcal{I}_c(c_1)(u, t)$ and $Q(t)$, it follows that $\mathbf{wp}(c, Q)(t)$. The assumptions establish $\mathbf{wp}(c, I \land Q)(t)$ from which $I(t)$ is straightforward. □

Lemma E.2 *For any $p_1, p_2 \in \mathcal{P}$, $I, A, B, P, Q \in \mathcal{A}$, $al \in$ Alist, $c \in \mathcal{C}$ and $s, t \in$ State,*

$$\frac{\begin{array}{l} s \overset{p_1}{\leadsto} t \\ \land \forall c \in p_1 : \forall A, B : (\vdash A \Rightarrow \mathbf{wp}(c, B)) \Rightarrow (\vdash I \land A \Rightarrow \mathbf{wp}(c, I \land B)) \\ \land \forall c \in p_1 : \forall A, B :\vdash ((I \land A) \Rightarrow \mathbf{wp}(c, I \land B) \Rightarrow [I \land (A \triangleleft al)]p_2[I \land (B \triangleleft al)]) \\ \land (I \land P)(s) \land (I \land Q)(t) \end{array}}{([I \land (P \triangleleft al)]p_2[I \land (Q \triangleleft al)])(s)}$$

Proof. By right induction on $s \overset{p_1}{\leadsto} t$.

Case $c_1 \in p_1$ and $s \overset{c_1}{\leadsto} t$. From $(I \wedge Q)(t)$ and $\mathcal{I}_c(c_1)(s,t)$ (definition of $s \overset{c_1}{\leadsto} t$) , it follows that $\mathbf{wp}(c, I \wedge Q)(s)$ is *true*. The proof is then straightforward from the assumptions.

Case $c_1 \in p_1$, $u \in$ *State*, $s \overset{p_1}{\leadsto} u$ and $u \overset{c_1}{\leadsto} t$. The inductive hypothesis states that the property holds for $s \overset{p_1}{\leadsto} u$. By definition of the specification operator for programs and from $([I \wedge (P \lhd al)]p_2[I \wedge (Q \lhd al)])(s)$, the assertion $(I \wedge (P \lhd al))(s)$ is *true*. The property to prove is that there is a state t' such that $s \overset{p_2}{\leadsto} t'$ and $(I \wedge (Q \lhd al))(t')$ is *true*.

As before, $\mathbf{wp}(c, I \wedge Q)(u)$ follows from $(I \wedge Q)(t)$ and $\mathcal{I}_c(c_1)(u,t)$. From Lemma (E.1) and the assumptions, it can be concluded that $I(u)$ is *true*. From the inductive hypothesis, it follows that $([I \wedge (P \lhd al)]p_2[I \wedge (\mathbf{wp}(c, (I \wedge Q)) \lhd al)])(s)$. By definition, this establishes a state u' such that $s \overset{p_2}{\leadsto} u'$ and $(I \wedge (\mathbf{wp}(c, I \wedge Q) \lhd al))(u')$. From the assumptions, $([(I \wedge \mathbf{wp}(c, I \wedge (Q)) \lhd al)]p_2[I \wedge (Q \lhd al)])(u')$ can be established from $((I \wedge \mathbf{wp}(c, I \wedge Q)) \Rightarrow \mathbf{wp}(c, I \wedge Q))(u')$, which is trivially *true*. This establishes $([I \wedge (\mathbf{wp}(c, I \wedge Q) \lhd al)]p_2[I \wedge (Q \lhd al)])(u')$. By definition, it follows that there is a state t' such that $s \overset{p_2}{\leadsto} t'$ and $(I \wedge (Q \lhd al)])(t')$ and therefore $[I \wedge (P \lhd al)]p_2[I \wedge (Q \lhd al)](s)$ is *true*, completing the proof. □

Proof. *Theorem (6.3)*

Straightforward from the assumptions, the definition of the specification operator and from Lemma (E.2). □

E.2 Theorem (6.4)

Lemma E.3 *Assume* $r \in \mathcal{R}$, $s, t \in$ *State and* $P, Q \in \mathcal{A}$.

$$\mathbf{wp}_0(r, Q)(s) \Leftrightarrow (\mathit{enabled}(r)(s) \wedge (\exists t : \mathit{mtrace}(\mathit{body}(r), s, t) \wedge Q(t)))$$

Proof. $(\Rightarrow)$

Note that $\mathbf{wp}_0(r, Q)(s) \Rightarrow \mathit{enabled}(r)(s)$ is straightforward by induction on r and by definition of **wp** (for commands).

The proof of $\mathbf{wp}_0(r, Q)(s) \Rightarrow \exists t : \mathit{mtrace}(\mathit{body}(r), s, t) \wedge Q(t)$ is by induction on r.

Case unit?(r). By definition $\mathbf{wp}_0(r, Q)(s) = \mathbf{wp}(\mathit{head}(r), Q)(s)$. The properties of the function **wp** establish $\exists t : \mathcal{I}_c(\mathit{head}(r))(s,t) \wedge Q(t)$ from $\mathbf{wp}(\mathit{head}(r), Q)$ and the conclusion is immediate from the definition of *mtrace* and $r = \mathit{unit}(\mathit{head}(r))$.

Case $\neg$*unit?*(r) and the property holds for any $r' \subset r$. By definition of $\mathbf{wp}_0$, $\mathbf{wp}_0(r, Q)(s) = \mathbf{wp}(\mathit{head}(r), Q')(s)$ where $Q' \in \mathcal{A}$. From the properties of **wp**, $\exists t' : \mathcal{I}_c(\mathit{head}(r))(s,t) \wedge Q'(t')$. There are two cases: either $Q'(t) \Rightarrow \mathit{final?}(\mathit{body}(r) - \{\mathit{head}(r)\})(t') \wedge Q(t')$ or there is a $r' \in \mathit{rest}(r)$ such that $Q'(t') \Rightarrow \mathbf{wp}_0(r', Q)(t')$. In the first case, the proof is as for the base case of the induction. For the second case, the inductive hypothesis establishes $\exists t : \mathit{mtrace}(\mathit{body}(r'), t', t) \wedge$

$Q(t)$. Since *body*$(r') \subseteq$ *body*$(r) - \{$*head*$(r)\}$, it follows that the trace *mtrace*$($*body*$(r'), t', t)$ can be established without *head*(r). From $\mathcal{I}_c($*head*$(r))(s, t')$ it follows that *mtrace*$($*body*$(r), s, t)$ is also *true*. The maximal trace is established since *head*(r) cannot be executed in state t if it is executed in state s. This establishes $\exists t :$ *mtrace*$($*body*$(r), s, t) \wedge Q(t)$ completing the proof. □

Proof. ($\Leftarrow$) By induction on r.

Case unit?(r). By definition, *mtrace*$($*body*$(r), s, t)$ is $\mathcal{I}_c($*head*$(r), s, t)$. From $Q(t)$ and the properties of **wp**, the conclusion $\mathbf{wp}_0(r, Q)(s)$ is straightforward.

Case $\neg$*unit?*(r) and the property holds for any $r' \subset r$. By definition and the assumption *enabled*$(r)(s)$, *mtrace*$($*body*$(r), s, t)$ establishes $\mathcal{I}_c($*head*$(r))(s, t')$ for $t' \in$ *State*. There are two cases: either t' is final for *body*$(r) - \{$*head*$(r)\}$ (and therefore $t' = t$) or there is a $c \in$ *body*$(r) - \{$*head*$(r)\}$ and *enabled*$(c')(t') \wedge$ *mtrace*$($*body*$(r) - \{$*head*$(r)\}, t', t)$. In the first case, the proof is immediate from $Q(t)$, *final?*$($*body*$(r) - \{$*head*$(r)\})(t)$ and the properties of **wp**. In the second case, c' begins a region $r' \in$ *rest*(r) such that *enabled*$(r')(t')$. From *mtrace*$($*body*$(r) - \{$*head*$(r)\}, t', t)$ and $r' =$ *region*$($*body*$(r) - \{$*head*$(r)\})$, it follows that *mtrace*$($*body*$(r'), t', t)$. Otherwise there is a $c_1 \in$ *body*$(r) - \{$*head*$(r)\}$ which is necessary for the trace from t' to t but not in the region beginning with c'. Since the trace implies that c' reaches c_1, c_1 must be in the region beginning with c' leading to a contradiction. From the inductive hypothesis, $\mathbf{wp}_0(r', Q)(t')$ is *true*. By definition of $\mathbf{wp}_0$, and from $\mathbf{wp}($*head*$(r), \mathbf{wp}_0(r', Q))(s)$, it follows that $\mathbf{wp}_0(r, Q)(s)$ is also *true*. □

Lemma E.4 *Assume* $P, Q \in \mathcal{A}$, $s, t \in$ *State*, $r \in \mathcal{R}$. *Assertion* $\mathbf{wp}_1(r, Q)$ *is the weakest precondition required to establish Q by an arbitrary number of maximal traces through* r.

$$\mathbf{wp}_1(r, Q)(s) \Leftrightarrow (\textit{enabled}(r)(s) \wedge (\exists t : \textit{mtrace}^+(\textit{body}(r), s, t)) \wedge Q(t)))$$

Proof. ($\Rightarrow$)

By induction on $\mathbf{wp}_1(r, Q)(s)$.

Case $\mathbf{wp}_0(r, Q)(s)$. The proof is straightforward from Lemma (E.3).

Case $\mathbf{wp}_1(r, q)(s)$, $r_1 < r$ and $\vdash q \Rightarrow \mathbf{wp}_0(r, Q)$. The inductive hypothesis states that the property is *true* for $\mathbf{wp}(r, q)(s)$.

From the inductive hypothesis, there is a $t' \in$ *State* such that *mtrace*$^+($*body*$(r), s, t') \wedge q(t')$. From the assumptions, $q(t') \Rightarrow \mathbf{wp}_0(r_1, Q)(t')$. From Lemma (E.3), it follows that there is a $t \in$ *State* such that *mtrace*$($*body*$(r'), t', t) \wedge Q(t)$. Since *body*$(r') \subseteq$ *body*(r) and r' contains every command in r reachable from *head*(r'), it follows that *mtrace*$($*body*$(r), t', t)$ (otherwise there is a command in r enabled in t which is not reachable from *head*(r') which is a contradiction). The conclusion *mtrace*$^+($*body*$(r), s, t) \wedge Q(t)$ is straightforward from the transitive closure of *mtrace*.

□

Proof. ($\Leftarrow$)

By right induction on *mtrace*$^+$(*body*$(r), s, t)$.

Case mtrace(*body*$(r), s, t) \wedge Q(t)$. From Lemma (E.3), $\mathbf{wp}_0(r, Q)(s)$ is *true* and $\mathbf{wp}_1(r, Q)(s)$ follows immediately by definition.

Case mtrace$^+$(*body*$(r), s, t')$ and *mtrace*(*body*$(r), t', t)$ and $Q(t)$. The inductive hypothesis states that the property holds for *mtrace*$^+$(*body*$(r), s, t')$.

From the assumption *mtrace*(*body*$(r), t', t)$, it follows that there is a command $c \in r$ such that *enabled*$(c)(t')$. Let $r_1 =$ *region*(*label*(c), *body*$(r))$, by definition, $r_1 < r$ and *enabled*$(r_1)(t')$. Also from *mtrace*(*body*$(r), t', t)$, *mtrace*(*body*$(r_1), t', t)$ is *true* since otherwise there a command necessary for the trace through r which cannot be reached by c (which is a contradiction). Lemma (E.3), *mtrace*(*body*$(r_1), t', t)$ and $Q(t)$ establish $\mathbf{wp}_0(r_1, Q)(t')$. The inductive hypothesis and $\mathbf{wp}_0(r_1, Q)$ establishes $\mathbf{wp}_1(r, \mathbf{wp}_0(r_1, q))(s)$. The conclusion $\mathbf{wp}_1(r, Q)(s)$ follows immediately from $r_1 < r, \vdash \mathbf{wp}_0(r_1, Q) \Rightarrow \mathbf{wp}_0(r_1, Q)$. and $\mathbf{wp}_1(r, \mathbf{wp}_0(r_1, q))(s)$. □

Proof. Theorem (6.4)

($\Rightarrow$)

By definition, $\mathbf{wp}(r, Q)(s) = \mathbf{wp}_1(r, Q \wedge$ *final?*$(r))(s)$. From Lemma (E.4), this establishes *enabled*$(r)(s)$ and $\exists t :$ *mtrace*$^+$(*body*$(r), s, t) \wedge Q(t) \wedge$ *final?*$(r)(t)$. From Lemma (D.22) (of Appendix D), this leads to *enabled*$(r)(s)$ and $\exists t : s \overset{r}{\leadsto} t \wedge$ *final?*$(r)(t) \wedge Q(t)$. The conclusion $\exists t : \mathcal{I}_r(r)(s, t) \wedge Q(t)$ is straightforward by definition.

($\Leftarrow$)

The proof is similar to that above except that $\mathcal{I}_r(r)(s, t)$, leads to *enabled*$(r)(s)$ and $s \overset{r}{\leadsto} t$ and *final?*$(r)(t)$. The truth of *mtrace*$^+$(*body*$(r), s, t)$ is established by Corollary (D.1) (of Appendix D). The remainder of the proof is straightforward from the definitions and Lemma (E.4).

□

E.3 Theorem (6.5)

The proof is by well-founded induction. The induction scheme for well-founded relation $\prec$: $(T \times T) \rightarrow$ *boolean* is:

$$\frac{\forall x : (\forall y : y \prec x \Rightarrow \Phi(y)) \Rightarrow \Phi(x)}{\forall z : \Phi(z)} \tag{E.1}$$

where $\Phi : (T \rightarrow$ *boolean*$), x, y, z : T$.

Proof. Theorem (6.5)

By well-founded induction on $\prec$ and n with $\Phi(n) = (\vdash G(n) \Rightarrow \mathbf{wp}_1(r, Q))$. From the induction scheme (Equation E.1), the property to prove is $\forall(x : T) : (\forall(y : T) : y \prec x \Rightarrow \Phi(y)) \Rightarrow \Phi(x)$.

Assume $x \in T$, the assumptions of Theorem (6.5) require a proof that $\forall j, r_1 : j \prec x \wedge r_1 < r \Rightarrow \vdash$ *enabled*$(r_1) \wedge G(j) \Rightarrow \mathbf{wp}_1(r_1, Q)$. Assume $j \in T$ and $r_1 \in \mathcal{R}$ such that $j \prec x$ and $r_1 < r$. By definition of $\vdash$ for assertions, there is a state s such that *enabled*$(r_1)(s)$ and $G(j)(s)$ and the property to prove is $\mathbf{wp}_1(r_1, Q)(s)$. From the induction scheme and $j \prec x$, $\Phi(j)$ is *true*. This leads to $\vdash G(j) \Rightarrow \mathbf{wp}_1(r, Q)$ and, by definition of $\vdash$ for assertions, this gives $G(j)(s) \Rightarrow \mathbf{wp}_1(r, Q)(s)$. By Lemma (E.4), $\mathbf{wp}_1(r, Q)(s) \Rightarrow$ *enabled*$(r)(s)$. Since *enabled*$(r_1)(s)$ is also *true*, *label*$(r) =$ *label*(r_1) and therefore $r = r_1$ (by definition of *region* and $<$ between regions). It follows from $\mathbf{wp}_1(r, Q)(s)$ and $r = r_1$ that $\mathbf{wp}_1(r_1, Q)(s)$ is *true*, completing the proof. $\square$