# Verification and Abstraction of Flow-Graph Programs with Pointers and Computed Jumps

M. Wahab
Dept. of Computer Science,
University of Warwick
wahab@dcs.warwick.ac.uk

November 9, 1998

**Abstract**

A flow-graph language which includes a simultaneous assignment, pointers and computed jumps is developed. The language is expressive enough that sequential composition can be defined as a function on commands, constructing a single command from its arguments. This allows the abstraction of a program to be constructed from the program text. This form of abstraction is the reverse of compilation: the abstraction of a program is also a program. The sequential composition operator can reduce the number of commands which must be considered when verifying a program. This provides a method for simplifying program verification. Proof rules are defined for reasoning about the liveness properties of flow-graph programs. The language is expressive enough to describe sequential object code programs and a program for the Alpha AXP processor is verified as an example.

## 1 Introduction

A program is verified to show that it is correct: the result of executing the program on a machine will satisfy its specification. High-level programs are difficult to verify since they are not directly executable but must be translated, by a compiler, to object code. As well as requiring a semantics for the language, which is often hard to obtain, verifying a high-level program requires a proof that that the compiler does not introduce errors into the object code. An object code program is written in a processor language and is executable on a machine. Although processor languages have a formal semantics, object code programs are also difficult to verify since their execution and data models of object code are more flexible than those of the programs which have been considered in verification. Object code also makes acute a general problem in verification: the more commands in the program to be verified, the more difficult it is to verify. Because a typical object code program has a large number of commands, the work required to verify object code is too great to be practical.

A processor language, often called an assembly language or an instruction set, is an instance of a flow-graph language with pointers and computed jumps. This paper describes such a language, called $\mathcal{L}$, which is expressive enough a processor instruction can be described as a single command of $\mathcal{L}$. Formal definitions for the syntax and semantics of $\mathcal{L}$ are given and used to define the proof rules of a program logic suitable for verifying the liveness properties of programs (Manna & Pnueli, 1981). The data and execution models of the language $\mathcal{L}$ generalise the models of processor languages and any sequential object code program, which does not modify itself, can be described as program of $\mathcal{L}$ by replacing each instruction with its equivalent $\mathcal{L}$ commands. Since the size of the $\mathcal{L}$ program will be equal to the size of the object code, verifying the program of $\mathcal{L}$ will be no more difficult than verifying the object code program.

Verifying a program of $\mathcal{L}$ can be simplified by constructing an *abstraction* of the program which reduces the number of commands to be considered during the course of a proof. The abstraction of a program $p$ is a program of $\mathcal{L}$ which is correct only if $p$ is correct. The method of abstracting programs used here is based on manipulating the commands of a program. *Sequential composition* is defined, as a function on commands of $\mathcal{L}$, to construct a single command which is equivalent to its arguments. An abstraction of a program is obtained by replacing commands of the program with the result of applying sequential composition to program commands. Since this reduce the number of commands to be considered, the resulting program will usually be simpler to verify, at worst no more difficult, than the original program. Both the verification and abstraction of programs are based on the text of a program, which allows the efficient implementation of automated proof tools to abstract from and verify programs.

### Notation

The types of Booleans, integers and natural numbers are denoted *boolean*, $\mathbb{Z}$ and $\mathbb{N}$ respectively. The type of functions from $S$ to $T$ are written $S \to T$, the $n$th cross-product of types $T_1 \ldots T_n$ is written $(T_1 \times \ldots \times T_n)$, a set of elements of type $T$ has type $\mathit{Set}(T)$. Predicates on a type $T$ are functions of type $(T \to \mathit{boolean})$. Types and sets are considered equivalent, that an identifier $x$ has type $T$ is written $x : T$ or $x \in T$. The boolean true, false, conjunction (and), disjunction (or), negation (not), implication (implies) and equivalence (iff) operators are written *true*, *false*, $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$. The universal and existential quantifiers are written $\forall$ and $\exists$ respectively. The equality is written $=$ and the defining equality is $\stackrel{\text{def}}{=}$. The addition, subtraction, multiplication, division and modulus operators are written $+$, $-$, $\times$, $\div$ and mod respectively; $x^y$ is $x$ to the power of $y$. Set membership, union, intersection, subset and proper subset are written $\in$, $\cup$, $\cap$, $\subseteq$ and $\subset$ respectively. The set of items satisfying a predicate $P$ is written $\{x \mid P(x)\}$. A set defined in terms of a BNF grammar contains all items satisfying that grammar. The simultaneous textual substitution of $e_1, \ldots, e_n$ for $v_1, \ldots, v_n$ in a term $t$ is written $t[e_1, \ldots, e_n/v_1, \ldots, v_n]$.

## 2 Background

Program verification is based on reasoning about the behaviour of a program during its execution, modelled by the changes made to the machine *state*. A state is a record of the values assigned to the program variables, a command begins execution in a state and produces a new state by an assignment of values to variables; the assignment can be conditional on the initial values of the variables. A command or program is specified by assertions on the states in which execution begins and ends (Hoare, 1969). The axioms of a program logic specify the semantics of commands and program while the proof rules of the logic are used to establish the specification from the semantics. A processor instruction makes simultaneous assignments to variables which are identified by expressions. These implement the *addressing modes* of the processor (Hayes, 1988) and are a generalisation of pointers to variables. A *pointer* is an expression which identifies, or refers to, a program variable. Pointers give rise to the aliasing problem: it is not possible to decide whether a pointer refers to an arbitrary variable by considering only the syntax of the expressions.

The aliasing problem affects the interpretation and specification of assignment commands. Assume that the simultaneous assignment of expressions $e_1, \ldots, e_n$ to variables $x_1, \ldots, x_n$ is written $x_1, \ldots, x_n := e_1, \ldots, e_n$. A command assigning different values to the same variable cannot be executed since no variable can have more than one value in a state. Without pointers, impossible assignments can be detected by a textual comparison of the variables $x_1, \ldots, x_n$. This test fails in the presence of pointers because of the aliasing problem. An alternative is to interpret a multiple assignment, with pointers, as a sequence of single assignments (Gries, 1981; Cartwright & Oppen, 1981). However, this increases the work needed to verify programs, such as object code, in which the majority of commands make simultaneous assignments. The specification of assignment commands is affected by aliasing since textual substitution is used to model the effect of the assignment on an assertion (Hoare, 1969; Dijkstra, 1976). Textual substitution is not adequate when the language include pointers and specialised substitution operators, which take into account the variable referred to by a pointer, must be used instead (Gries, 1981; Manna & Waldinger, 1981b; Francez, 1992). Because all non-trivial programs contain assignment commands, both these problems must be solved if the programs of a language which includes pointers are to be verified.

In a flow-graph language, a program is a set of commands each of which is uniquely *labelled* (Loeckx & Sieber, 1987) and each state identifies the command selected for execution by its label. In object code, the labels are address in memory and instructions are identified by a register called the program counter (or the instruction pointer), denoted *pc*. If $l$ is a label and $c$ a command $c$ then the labelled command $l : c$ is selected for execution in a state $s$ iff the value of *pc* in $s$ is $l$, $pc = l$. A command $c$ has *control* of the machine when it is executed and selects a *successor* $c'$ by assigning the label of $c'$ to the program counter *pc*. The *flow of control* through a program is the order in which the program commands are executed.

Flow-graph programs are often studied by extending a structured language with a *jump* command, usually called *goto*. A jump, or *computed jump*, is a command which does nothing except pass control to a successor, the *target*. The target of the jump *goto* $l$ is identified by the expression $l$, which can depend on program variables. Clint & Hoare (1972) describe such a language and interpret the *goto* as a command which passes control to a target but which does not terminates (similar approaches are used by Arbib & Alagić, 1970 and de Bruin, 1981). Jifeng He (1983) uses a different approach in which the *goto* terminates before its target begins but separates the program variables from the flow of control, making it possible for the jump command *goto* $l$ to terminate in a state where $pc \neq l$. Both interpretations are false for processor languages, where jump commands terminate and the flow of control is determined by a program variable (the program counter, *pc*).

The difficulties associated with the jump commands are a consequence of their use in structured languages. In these languages, programs are formed as compound commands using primitive syntactic constructs, such as *sequential composition* (Hoare, 1969; Loeckx & Sieber, 1987), to determine the order in which commands are executed. Jump commands do not cause difficulties in a flow-graph language which uses a program counter to select commands. A jump is simply an assignment to the program counter: the command *goto* $l$ is the assignment command $pc := l$. More generally, there is no need to distinguish between jumps and other commands of the language since all commands select a successor by an assignment to the program counter.

## Abstraction

The principal proof methods for verification are the method of inductive assertions Floyd (1967) and the method of intermittent assertions (Manna, 1974; Burstall, 1974). In both, a program is verified by reasoning about the properties established by sequences of program commands. A large part of a verification proof is to show that a property to be established by a sequence is a logical consequence of the properties established by individual commands of the sequence. This can be simplified by constructing a command $c$ which has the same effect on the machine state as the execution of a sequence of commands $c_1, \ldots, c_n$. The command $c$ will be an abstraction of the sequence, the properties of the commands $c_1, \ldots, c_n$ can be deduced by reasoning about the single command.

Abstraction is the reverse of refinement (Back & von Wright, 1989). A program $p_1$ is refined by a program $p_2$, written $p_1 \sqsubseteq p_2$, iff $p_2$ satisfies any specification satisfied by $p_1$. Program $p_1$ is an abstraction of $p_2$, verifying that $p_1$ satisfies a specification is enough to show that $p_2$ also satisfies that specification. Verifying program $p$ with respect to specification $S$ can be simplified by constructing a program $p'$ such that $p'$ satisfies $S$ and $p'$ is an abstraction of $p$, $p' \sqsubseteq p$. Because $p$ refines $p'$, this is enough to show that $p$ satisfies specification $S$. The abstraction of a program can be constructed from the text of the program: Hoare et al. (1987) describe a set of algebraic rules, for a language without pointers or jumps, which define the relationships between different combinations of commands. If it is know that command $c_1$ will pass control to command $c_2$, then a command can be constructed which is an abstraction of the sequence $c_1, c_2$. However, this is complicated by the presence of computed jumps and pointers. To abstract from commands $c_1$ and $c_2$ it must be known that control will pass from $c_1$ to $c_2$. When $c_1$ is a computed jump, the target cannot be determined from the syntax of $c_1$; whether control will pass from $c_1$ to $c_2$ is undecidable.

Pointers affect the abstraction of assignment commands, which is based on merging the list of variables to which assignments are made and on the substitution of values for variables. Assume $x, y, z$ are variables and $e_1, \ldots, e_4$ are expressions. The abstraction of the two simultaneous assignment commands $x, y := e_1, e_2$ and $y, z := e_3, e_4$ 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 expressions $e_1$ and $e_2$, assigned to variables $x$ and $y$ by the first command, are substituted for the variables in the the second command. The variable $y$, which occurs in both lists of assignments, is assigned $e_3[e_1, e_2/x, y]$, since the second assignment to $y$ supersedes the first. The lists of assignments, to $x, y$ and to $y, z$ and $z$ are merged to the single list $x, y, z$, using syntactic equality to compare the variables. Because of the aliasing problem, it is not possible to merge the lists of variables using syntactic equality nor is it possible to use textual substitution when the assignments are to pointers.

## Verification and Abstraction in the Language $\mathcal{L}$

The verification and abstraction of programs in the presence pointer and computed jumps is studied here in the context of the language $\mathcal{L}$. This is a flow-graph language with three commands: a simultaneous assignment, a conditional and a labelling command. The expressions of $\mathcal{L}$ generalise those found in processor languages, pointers are treated as instances of a class of expressions which identify variables. An equivalence relation between pointers is used to detect executable assignment commands and to define operators for substitution and for assignment list merging. This combination of commands and expressions is enough to allow the abstraction of commands based on their syntax. The rules for abstracting commands are then defined as a function on pairs of commands, which results in a command equivalent to the sequential composition of its arguments.

The programs of $\mathcal{L}$ are sets of labelled commands. Their semantics provide the basis for a refinement relation, which is used to show that a program can be replaced, during verification, with its abstraction. The refinement relation is also used to show that abstracting from the commands of a program results in an abstraction of the program. A program logic for the language $\mathcal{L}$ is justified from the semantics of the commands and the programs. The proof rules of the logic allow a program, or its abstraction, to be verified using the method of intermittent assertions (Manna, 1974; Burstall, 1974) while the proof rules for the commands are based on the *wp* predicate transformer (Dijkstra, 1976). As an example of verification and abstraction, a program of the Alpha AXP processor is defined in terms of $\mathcal{L}$ and shown to be correct.

### Outline

The remainder of this paper is structured as follows: the definition of the language $\mathcal{L}$ begins, in **Section 3**, with the expressions of the language. This includes the models of pointers and the definition of simultaneous substitution, needed for proof rules and to construct abstractions. The syntax and semantics of the commands of $\mathcal{L}$ are described in **Section 4** and are followed, in **Section 5**, by the syntax and semantics of the programs. The method for constructing abstractions of a program is described in **Section 6**, beginning with the definition of sequential composition for commands of $\mathcal{L}$. **Section 7** gives an example of the proof rules which can be defined for the language and is followed in **Section 8** by the verification of a program for the Alpha AXP processor. The paper ends with the conclusion in **Section 9**.

## 3 Expressions of $\mathcal{L}$

Commands of a programming language make changes to the machine state by assignments to variables and the value assigned to each variable is the result of evaluating an expression. In the language $\mathcal{L}$, the variables to which the assignments are made can also be the result of evaluating expressions. An expression is a constant, the name of a program variable or the application of a function to one or more expressions. In the language $\mathcal{L}$, constants are *values*, representing the program data, the *labels* of commands and the variable *names*. The result of a *value expression* is a value, these expressions perform operations such as arithmetic on the program data. Value expressions are also used, as *Boolean expressions*, to perform tests on the program variables. *Name expressions* generalise the pointers and always evaluate to the name of a variable. *Label expressions* are used to calculate the label of a command, to identify the target of a jump. To support program verification and abstraction, $\mathcal{L}$ also includes substitution expressions, which are used to describe the changes made by an assignment of values to variables.

### 3.1 Basic Model

The basic model of $\mathcal{L}$ determines the values, names and labels, and the functions which can occur in an expression. The arguments to all functions are values, as is the case in most programming languages. The result of a function application is either a value, a name or a label and this is used to group the functions of $\mathcal{L}$. The functions which result in a name are the basis for the name expressions, identifying a variable by performing some calculation on program data. The *values* are any non-empty set representing the program data and the *labels* are a subset of the values. The *names* of program variables are distinct from the values and there is a mapping from a subset, *Vars*, of the values to the names. The variable names include at least the *program counter*, *pc*, which identifies the command selected for execution. An interpretation of the values as Booleans allows tests to be made on the values of program variables.

**Definition 3.1** *Constants*

The set of values, *Values*, is of some type $T_1$, $\mathit{Values} : \mathit{Set}(T_1)$, and *Labels* is a subset of *Values*.

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

The set of variable names, $\mathit{Names} : \mathit{Set}(T_2)$ is distinct from the values. There is a subset *Vars* of the set of values, $\mathit{Vars} \subseteq \mathit{Values}$ and a function $\mathit{name} : \mathit{Vars} \to \mathit{Names}$, which constructs variables names from the elements of *Vars*. For $x, y \in \mathit{Vars}$:

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

There is a name $\mathit{pc} \in \mathit{Names}$.

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

$$\mathbf{true}, \mathbf{false} \in \mathit{Values} \quad \mathcal{B}(\mathbf{true}) = \mathit{true} \quad \mathcal{B}(\mathbf{false}) = \mathit{false}$$

□

All expressions are evaluated in a *state*, which records the values stored in the program variables. States are modelled as functions from names to values: if $s$ is a state and $v$ the name of a variable then $s(v)$ is the value of variable $v$ in $s$.

**Definition 3.2** *States*

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

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

□

The functions of $\mathcal{L}$ are defined by identifiers, which can occur in the syntax of an expression, and by an interpretation of these identifiers, which provides the function definitions and is used in the semantics of the expressions. The domain of each function, of arity $n$, is the $n$th product of the set of values. The functions range over either the set of values, labels or names and a function identifier is either a *value function*, a *name function* or a *label functions*. There is at least one value function, **equal**, which is interpreted as the equality between values.

**Definition 3.3** *Function identifiers and interpretation*

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

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

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, $\mathcal{F}_n$ such that $\mathcal{F}_v \cup \mathcal{F}_n \cup \mathcal{F}_l \subseteq \mathcal{F}$. For any $f \in \mathcal{F}$, these sets satisfy:

$$\begin{aligned} f \in \mathcal{F}_v &\Leftrightarrow \forall (e_1, \ldots, e_m) : \mathcal{I}_f(f)(e_1, \ldots, e_m) \in \mathit{Values} \\ f \in \mathcal{F}_l &\Leftrightarrow \forall (e_1, \ldots, e_m) : \mathcal{I}_f(f)(e_1, \ldots, e_m) \in \mathit{Labels} \\ f \in \mathcal{F}_n &\Leftrightarrow \forall (e_1, \ldots, e_m) : \mathcal{I}_f(f)(e_1, \ldots, e_m) \in \mathit{Names} \end{aligned}$$

where $m$ is the arity of the function identifier $f$.

There is a function identifier **equal** $\in \mathcal{F}_v$, with arity 2, which is interpreted as the equality between its arguments:

$$\mathcal{I}_f(\textbf{equal})(x, y) \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } x = y \\ \textbf{false} & \text{otherwise} \end{cases}$$

For any $x, y$, **equal**$(x, y)$ will usually be written $x =_o y$. □

The interpretation of function identifiers, $\mathcal{I}_f$, is a total function; a partial function whose identifier is in $\mathcal{F}_v \cup \mathcal{F}_n$ can be defined using undefined values. These are modelled by application of the Hilbert choice operator, $\epsilon$, to the empty set.

**Definition 3.4** *Undefined values*

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

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

□

This approach does not allow reasoning about the undefined values (as the approach of Barringer et al., 1984 does) but is sufficient for the expressions of $\mathcal{L}$ which will be considered here.

Note that because the labels are a subset of the values, the label functions are also a subset of the value functions, $\mathcal{F}_l \subset \mathcal{F}_v$. 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 is undefined. Value functions are used to calculate the result of an operation on data. These operations are also used to perform tests on the state of the machine, using the Boolean interpretation, $\mathcal{B}$, of the values. Since the Boolean constants, *true* and *false* are values, the Boolean negation and conjunction are value functions. This provides propositional (quantifier-free) logical formulas, which can be used in the commands and expressions of $\mathcal{L}$.

**Definition 3.5** *Boolean operators*

The negation and conjunction operators of $\mathcal{L}$ are the value functions **not** and **and**.

$$\textbf{not} \in \mathcal{F}_v \qquad\qquad \textbf{and} \in \mathcal{F}_v$$

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

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

$$\mathit{Values} \stackrel{\text{def}}{=} \mathbb{Z} \quad \mathit{Labels} \stackrel{\text{def}}{=} \{x : \mathbb{N} \mid x \leq 2^{64}\}$$
$$\mathit{Vars} \stackrel{\text{def}}{=} \mathbb{N} \quad \mathit{Regs} \stackrel{\text{def}}{=} \{pc, \mathbf{r0}, \mathbf{r1}, \ldots, \mathbf{r30}\} \quad \mathit{Names} = \{\mathit{name}(x) | x \in \mathit{Vars}\} \cup \mathit{Regs}$$

$$\{\mathbf{lt}, \mathbf{plus}, \mathbf{minus}, \mathbf{mult}, \mathbf{div}, \mathbf{mod}, \mathbf{exp}\} \subseteq \mathcal{F}_V \quad \{\mathbf{ref}\} \subseteq \mathcal{F}_n \quad \{\mathbf{loc}\} \subseteq \mathcal{F}_l$$

$$\mathcal{I}_f(\mathbf{plus})(x, y) \stackrel{\text{def}}{=} x + y \qquad \mathcal{I}_f(\mathbf{minus})(x, y) \stackrel{\text{def}}{=} x - y$$
$$\mathcal{I}_f(\mathbf{mult})(x, y) \stackrel{\text{def}}{=} x \times y \qquad \mathcal{I}_f(\mathbf{div})(x, y) \stackrel{\text{def}}{=} x \div y$$
$$\mathcal{I}_f(\mathbf{mod})(x, y) \stackrel{\text{def}}{=} x \bmod y \quad \mathcal{I}_f(\mathbf{exp})(x, y) \stackrel{\text{def}}{=} x^y$$
$$\mathcal{I}_f(\mathbf{lt})(x, y) \stackrel{\text{def}}{=} \begin{cases} \mathbf{true} & \text{if } x < y \\ \mathbf{false} & \text{otherwise} \end{cases}$$
$$\mathcal{I}_f(\mathbf{ref})(x) \stackrel{\text{def}}{=} \begin{cases} \mathit{name}(x) & \text{if } x \in \mathit{Vars} \\ \mathit{undef}(\mathit{Names}) & \text{otherwise} \end{cases}$$
$$\mathcal{I}_f(\mathbf{loc})(x) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \in \mathit{Labels} \\ \mathit{undef}(\mathit{Labels}) & \text{otherwise} \end{cases}$$

*Notation:* For any $x, y$, $\mathbf{lt}(x, y)$ is written $x <_o y$, $\mathbf{plus}(x, y)$ is written $x +_o y$, $\mathbf{mult}(x, y)$ is written $x \times_o y$, $\mathbf{div}(x, y)$ is written $x \div_o y$, $\mathbf{minus}(x, y)$ is written $x -_o y$, $\mathbf{mod}(x, y)$ is written $x \bmod_o y$ and $\mathbf{exp}(x, y)$ is written $x^y$

Figure 1: Basic Model for Alpha AXP

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

**Example 3.1** *Alpha AXP processor: Basic model*

The Alpha AXP is a $64$ bit processor architecture based on a RISC design (Sites, 1992). A detailed description which includes the semantics of the processor language, is given in the processor manual (Digital Equipment Corporation, 1996). The Alpha processor has 32 general purpose registers, which will be denoted **r0** to **r31** and the program counter *pc* is a register. The value stored in register 31, **r31**, is always $0$ and assignments to register **r31** are always ignored.

Data is stored in memory as bit-vectors of size 32 (called *long-words*), representing the natural numbers $0, \ldots, 2^{32} - 1$. Instructions are stored in a single long-word and all memory access is aligned on a long-word: an address $x$ identifies the location $x - (x \bmod 4) \times 4$ and addresses 4, 5, 6 and 7 are all synonyms for address $4$. This data model, described by Sites (1992), will be used throughout this paper. The model described in the processor manual (Digital Equipment Corporation, 1996), for a later design of the Alpha AXP, allows byte sized memory access: addresses $4, 5, 6, 7$ identify four consecutive memory locations.

A model of the Alpha AXP data operations as constants and functions of $\mathcal{L}$ is given in Figure (1). The set of values, *Values*, on which a program of the Alpha AXP operates is modelled by the set of integers. The names of program variables are the memory locations, modelled by the naturals, together with the processor registers. 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. The functions in $\mathcal{F}_V$ are the basic arithmetic operations from which operations on bit-vectors can be derived. □

## 3.2 Syntax of the Expressions

Expressions of $\mathcal{L}$ are made up of constants and function applications or are a substitution. As with the constants and functions, the expressions are grouped according to whether an expression results in value, a name or a label. This allows restrictions to be imposed on the occurrence of an expression. For example, assignments can be made only to expressions which result in a name. Substitution is defined as operator which constructs an expression of $\mathcal{L}$. This allows the syntax of substitution, which describes the changes to be made to an expression, to be separated from its semantics, which carries out the substitution in a given state.

The syntax of a substitution expression is made up of an expression of $\mathcal{L}$ and an *assignment list*, a data structure which associates name and value expressions. The assignment lists and the expressions are mutually dependent. Assignment lists are required for substitution expressions and depend on the value expressions; substitution constructs a value expression, since the result of a substitution is a value, and can occur in an assignment list. For simplicity, the syntax of assignment lists, and their semantic operations, will be generalised over the sets and relations which depend on the expressions of $\mathcal{L}$.

### Assignment Lists

The assignment lists are similar to the *association lists* used to define textual substitution (Manna & Waldinger, 1981a; Paulson, 1985). However, the syntax of the assignment lists includes an operator, for the combination of lists, to represent the merger of assignment lists, needed for the abstraction of commands. The semantic functions used to interpret substitution expressions take this operator into account when calculating the result of a substitution.

**Definition 3.6** *Assignment lists*

For sets $N$ and $V$, the set $\mathit{Alist}(N, V)$ contains all lists of assignments of elements of $V$ to elements of $N$. The set $\mathit{Alist}(N, V)$ is built up from the empty list, *nil* and from the operator *cons*, $\_ \cdot \_$, and the combining operator, $\_ \oplus \_$, and satisfies the grammar:

$$\mathit{Alist} \quad = \quad \mathit{nil} \mid (\langle N \rangle, \langle V \rangle) \cdot \langle \mathit{Alist} \rangle \mid \langle \mathit{Alist} \rangle \oplus \langle \mathit{Alist} \rangle$$

The sub-term relation, $\ll$ between assignment lists has type $(\mathit{Alist} \times \mathit{Alist}) \to \mathit{boolean}$ and definition:

$$\begin{aligned}
al \ll \mathit{nil} &\stackrel{\text{def}}{=} al = \mathit{nil} \\
al \ll (x, e) \cdot bl &\stackrel{\text{def}}{=} al = (x, e) \cdot bl \lor x \ll bl \\
al \ll (bl \oplus cl) &\stackrel{\text{def}}{=} al = bl \oplus cl \lor al \ll bl \lor al \ll cl
\end{aligned}$$

An assignment list $al$ is simple if no sub-term of $al$ is constructed by the operator $\oplus$.

$$
\begin{aligned}
\mathit{simple?}(al) &\stackrel{\text{def}}{=} \forall bl, cl : \neg(bl \oplus cl) \ll al \\
\mathit{Slist}(N, V) &\stackrel{\text{def}}{=} \{al : \mathit{Alist}(N, V) \mid \mathit{simple?}(al)\}
\end{aligned}
$$

□

The operations required for substitution use the syntax of the assignment lists to find the first value expression associated with some name in a state. The search, for the value associated with name $x$ in state $s$ is ordered: 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 $cl$ (the choice is arbitrary).

**Definition 3.7** *Membership and find*

For a relation $R$ between expressions of $\mathcal{E}_n$ with type $R : (\mathcal{E}_n \times \mathcal{E}_n) \to \mathit{State} \to \mathit{boolean}$, an assignment list $al \in \mathit{Alist}$ and state $s$, the name $x \in_s \mathcal{E}_n$ is a member in $s$ of $al$ iff there is a name expression $x'$ in $al$ such that $R(x, x')(s)$.

$$
\begin{aligned}
x \in_s \mathit{nil} &\stackrel{\text{def}}{=} \mathit{false} \\
x \in_s (x_1, e_1) \cdot al &\stackrel{\text{def}}{=} R(x, x_1)(s) \lor (x \in_s al) \\
x \in_s (al \oplus bl) &\stackrel{\text{def}}{=} x \in_s al \lor x \in_s bl
\end{aligned}
$$

Function *find* has type $(\mathcal{E}_n \times \mathit{Alist}) \to \mathit{State} \to \mathcal{E}$ and searches an assignment list for an expression assigned to a name.

$$
\begin{aligned}
\mathit{find}(x, \mathit{nil})(s) &\stackrel{\text{def}}{=} x \\
\mathit{find}(x, (x_1, e_1) \cdot al)(s) &\stackrel{\text{def}}{=} \begin{cases} e_1 & \text{if } R(x, x_1)(s) \\ \mathit{find}(x, al) & \text{otherwise} \end{cases} \\
\mathit{find}(x, (al \oplus bl))(s) &\stackrel{\text{def}}{=} \begin{cases} \mathit{find}(x, bl)(s) & \text{if } x \in_s bl \\ \mathit{find}(x, al)(s) & \text{otherwise} \end{cases}
\end{aligned}
$$

□

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

### Expressions

The expressions of $\mathcal{L}$ are made up of the value, the name and the label expressions. The set of value expressions is the largest, containing all expressions of the language $\mathcal{L}$ and includes the substitution expressions. The sets of name and label expressions are obtained by restrictions on the functions and constants. Name expressions are either constant names or the application of a name function to value expressions. The label expressions are either labels or the application of a label function to expressions. The Boolean expressions are synonymous with the value expressions.

**Definition 3.8** *Syntax of the expressions*

The sets of value expressions $\mathcal{E}$, name expressions $\mathcal{E}_n$ and label expressions $\mathcal{E}_l$ satisfy the grammar

$$\begin{array}{rcl} \mathcal{E}_n & = & \langle \mathit{Names} \rangle \mid \langle \mathcal{F}_n \rangle(\langle \mathcal{E} \rangle, \ldots, \langle \mathcal{E} \rangle) \\ \mathcal{E}_l & = & \langle \mathit{Labels} \rangle \mid \langle \mathcal{F}_l \rangle(\langle \mathcal{E} \rangle, \ldots, \langle \mathcal{E} \rangle) \\ \mathcal{E} & = & \langle \mathit{Values} \rangle \mid \langle \mathcal{F}_v \rangle(\langle \mathcal{E} \rangle, \ldots, \langle \mathcal{E} \rangle) \mid \langle \mathcal{E}_n \rangle \mid \langle \mathcal{E}_l \rangle \mid \langle \mathcal{E} \rangle \triangleleft \langle \mathit{Alist}(\mathcal{E}_n, \mathcal{E}) \rangle \end{array}$$

The set of boolean expressions, $\mathcal{E}_b$, is the set of value expressions, $\mathcal{E}_b = \mathcal{E}$.

The set $\mathit{Alist}(\mathcal{E}_n, \mathcal{E})$ will be abbreviated *Alist*. Note that, for $e \in \mathcal{E}$ and $al \in \mathit{Alist}$, the substitution of $al$ in $e$ is $e \triangleleft al$. □

Expressions which calculate the result of an operation on data are modelled by the value expressions $\mathcal{E}$ while the name and label expressions allow restrictions to be imposed on the expressions which occur in an command. 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.

## 3.3 Semantics of the Expressions

The evaluation of an expression is by interpretation functions which range over the set of values, names and labels. The value and label expressions share the same interpretation, a label expression is constrained, by its syntax, to result in a label constant (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 it is a constant name, or is 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 code generation techniques (see Aho et al., 1986).

The semantics of the expressions are complicated by substitution, which updates the state in which an expression is evaluated: if the substitution expression $e \triangleleft al$ is interpreted in state $s$ then $e$ is evaluated in a state $s'$, obtained by updating state $s$ with the assignments of $al$. The operations needed to update a state depend on the interpretation of both name and value expressions. These, in turn, depend on the interpretation of substitution since substitution expressions can occur in either a name or a value expression.

**Definition 3.9** *Interpretation of expressions*

The interpretation of the expressions are defined in terms of functions $\mathcal{I}_e$, $\mathcal{I}_n$ and $\mathcal{I}_l$ on value expressions, name expressions and label expressions respectively.

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

The *equivalence* relation between expressions in a state $s$ and under an interpretation $\mathcal{I} \in \{\mathcal{I}_e, \mathcal{I}_n, \mathcal{I}_l\}$ is defined:

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

The relation $R$ of Definition (3.7) is the equivalence under the interpretation of name expressions, $\mathcal{I}_n$: $R(e_1, e_2)(s) = e_1 \equiv_{(\mathcal{I}_n, s)} e_2$.
The *state update* function, *update*, is defined:

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

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

$$\mathcal{I}_e(e)(s) \stackrel{\text{def}}{=} \begin{cases} e & \text{if } e \in \mathit{Values} \\ s(\mathcal{I}_n(e)(s)) & \text{if } e \in \mathcal{E}_n \\ \mathcal{I}_f(f)(\mathcal{I}_e(a_1)(s), \ldots, \mathcal{I}_e(a_m)(s)) & \text{if } e = f(a_1, \ldots, a_m) \\ \mathcal{I}_e(e_1)(\mathit{update}(al, s)) & \text{if } e = e_1 \triangleleft al \end{cases}$$
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 $\mathcal{I}_n$.

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

The interpretation of the label expressions, $\mathcal{I}_l$, is the interpretation of the value expressions: $\mathcal{I}_l \stackrel{\text{def}}{=} \mathcal{I}_e$.
The function $\mathcal{I}_b$, of type $\mathcal{E} \to \mathit{State} \to \mathit{boolean}$ is the interpretation of an expression as a boolean, defined: $\mathcal{I}_b(e)(s) \stackrel{\text{def}}{=} \mathcal{B}(\mathcal{I}_e(e)(s))$. □

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 $s(x')$. The application of the interpretation as a name, $\mathcal{I}_n$, to the name expression $f(a_1, \ldots, a_n)$, in a state $s$, interprets the arguments $a_1, \ldots, a_n$ as values, using function $\mathcal{I}_e$ in state $s$. The interpretation of name function $f$ is applied to these values to obtain the result of the name expression, a constant name. Applying either interpretation to a constant $e$ results in the constant. However, the notion of a constant differs, a name $x \in \mathit{Names}$ is a constant only under the interpretation as a name, $\mathcal{I}_n$.

Although the constants and functions of the basic model provide simple expressions, these can be used to derive increasingly complex expressions. For example, the Boolean disjunction can be derived from the negation and conjunction functions of $\mathcal{F}_\mathcal{V}$.

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

Because the negation and conjunction form expressions of $\mathcal{E}$, the disjunction of two expressions is also an expression of $\mathcal{E}$.

**Example 3.2** *Alpha AXP: Data operations*

The largest number which can be represented on the Alpha AXP is $2^{64} - 1$ and all arithmetic operations of the processor are performed within this limit. The basic operations are given in Figure (2) as expressions of $\mathcal{E}$ derived from the values and value functions of Figure (1). The arithmetic operations

$$
\begin{array}{ll}
\textit{Arithmetic:} & x =_{64} y \stackrel{\text{def}}{=} (x \bmod_o 2^{64}) =_o (y \bmod_o 2^{64}) \\
& x <_{64} y \stackrel{\text{def}}{=} (x \bmod_o 2^{64}) <_o (y \bmod_o 2^{64}) \\
& x +_{64} y \stackrel{\text{def}}{=} (x +_o y) \bmod_o 2^{64} \\
& x -_{64} y \stackrel{\text{def}}{=} (x -_o y) \bmod_o 2^{64} \\
& x \times_{64} y \stackrel{\text{def}}{=} (x \times_o y) \bmod_o 2^{64} \\
& \mathbf{Long}(x) \stackrel{\text{def}}{=} x \bmod_o 2^{32} \\
& \mathbf{B}(x)(y) \stackrel{\text{def}}{=} (y \div_o 2^{x}) \bmod_o 2^{8} \\
\textit{Memory access:} & \mathbf{mem}(a) \stackrel{\text{def}}{=} \mathbf{ref}(a -_o (a \bmod_o 4)) \\
& \mathbf{inst}(a) \stackrel{\text{def}}{=} \mathbf{loc}(a -_o (a \bmod_o 4)) \\
& \\
& \text{where } x, y \in \mathcal{E}, a \in \textit{Values}
\end{array}
$$

Figure 2: Data Operations for the Alpha AXP

are written in the infix notation and are for bit-vectors of size 64. The function **Long** is the conversion of an arbitrary value to a bit-vector of size 32. The function **B** is the accessor for bytes: the $n$th byte of an expression $e \in \mathcal{E}$ is obtained by $\mathbf{B}(n)(e)$, where $\mathbf{B}(0)(e)$ is the least significant byte of $e$. Note that the expression $\mathbf{Long}(e)$ is equivalent to the expression

$$(\mathbf{B}(3)(e) \times_o 2^3) +_o (\mathbf{B}(2)(e) \times_o 2^2) +_o (\mathbf{B}(1)(e) \times_o 2^1) +_o \mathbf{B}(0)(e))$$

The name function **mem** in Figure (2) provides access to variables in memory. For $x \in \mathcal{E}$, the name resulting from $\mathbf{mem}(x)$ is aligned at a long-word; $x$ is assumed to be a multiple of four and rounded down if not. The label function, **inst**, applied to expression $x \in \mathcal{E}$ identifies the command stored at the address $x$. □

### 3.3.1 Equivalence between Expressions

Expressions can be compared by syntactic equality or by semantic equivalence. 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. To manipulate and simplify expressions, it is necessary to determine when expressions are equivalent. The equivalence relation of Definition (3.9), compares two expressions in a given state. A stronger relation, which can be used for rewriting (Duffy, 1991), asserts that two expressions have the same interpretation in any state.

**Definition 3.10** *Strong Equivalence*

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.

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

□

Names which are equivalent under the interpretation $\mathcal{I}_n$ will also be equivalent under the value interpretation $\mathcal{I}_e$ and arguments to functions can always by replaced with equivalent expressions.

**Lemma 3.1** *Properties of equivalence relations*

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

1. *Syntactic equality establishes strong equivalence:* $e_1 = e_2 \Rightarrow e_1 \equiv_{\mathcal{I}} e_2$.
2. *Equivalent names are equivalent as values:* $n_1 \equiv_{(\mathcal{I}_n,s)} n_2 \Rightarrow n_1 \equiv_{(\mathcal{I}_e,s)} n_2$.
3. *Arguments can be replaced with equivalents:* $e_1 \equiv_{(\mathcal{I},s)} e_2 \Rightarrow f(e_1) \equiv_{(\mathcal{I},s)} f(e_2)$.

**Proof.** Straightforward from definitions. □

Because the equivalence of name expressions under $\mathcal{I}_n$ is enough to establish equivalence under $\mathcal{I}_e$, the interpretation function will not, in general, 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 $\mathcal{I}_e$.

**Example 3.3** Assume $v_1, v_2 \in$ *Values*, $x_1, x_2 \in$ *Names*. If $v_1 = v_2$ then the expressions **ref**$(v_1)$ and **ref**$(v_2)$ are strongly equivalent, **ref**$(v_1) \equiv$ **ref**$(v_2)$. If **ref**$(v_1)$ is strongly equivalent to $x_1$, then **ref**$(v_1) +_o x_2$ is strongly equivalent to $x_1 +_o x_2$ and if $x_1 = x_2$ then $x_1 +_o x_2 \equiv 2 \times_o x_1$. □

## 3.4 Derived Substitution Expressions

The substitution operator of $\mathcal{E}$ constructs a value expression and cannot be used where a name expression is required. Separate substitution operators are needed for the name expressions and for the assignment lists, to apply a substitution to the value expressions which can occur in both. Substitution in name expressions, interpreted as names, is defined on the syntax of name expressions, applying substitution to the value expressions which occur as arguments to a name function.

**Definition 3.11** *Substitution in name expressions*

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

$$x \triangleleft al \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \in \textit{Names} \\ f(e_1 \triangleleft al, \ldots, e_m \triangleleft al) & \text{if } x = f(e_1, \ldots, e_m) \end{cases}$$

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

The interpretation of substitution in a name expression is consistent with its interpretation as a value. If substitution is applied to a constant name (in *Names*) then the name is unchanged. If substitution is applied to a function application then the arguments are evaluated in the updated state. The effect of a substituting assignment list $al$ in a name expression is therefore 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 $s$,*

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

**Proof.** Straightforward, by induction on $x$ and from the definitions. □

Substitution in assignment lists applies the substitution operators to each name-value pair in an assignment list.

**Definition 3.12** *Substitution in assignment lists*

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

$$\begin{aligned}
\mathit{nil} \triangleleft bl &\stackrel{\text{def}}{=} \mathit{nil} \\
((x, e) \cdot al) \triangleleft bl &\stackrel{\text{def}}{=} (x \triangleleft bl, e \triangleleft bl)) \cdot (al \triangleleft bl) \\
(cl \oplus dl) \triangleleft bl &\stackrel{\text{def}}{=} (cl \triangleleft bl) \oplus (dl \triangleleft bl)
\end{aligned}$$

□

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 $bl$. 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 $x \triangleleft al$.

**Lemma 3.3** *For name expression $x \in \mathcal{E}_n$, assignment lists $al, bl \in$ Alist and state $s$,*

$$x \in_{\mathit{update}(bl,s)} al = (x \triangleleft bl) \in_s al \triangleleft bl$$

**Proof.** Straightforward, by induction on $al$. □

Substitution and the combination of assignment lists allow the changes made to a state by two assignment commands, executed in sequence, to be described as an assignment list.

**Theorem 3.1** *For assignment lists $al, bl$ and state $s$,*

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

**Proof.** By induction on $al$ and by extensionality with $v \in \mathit{Names}$. The property to be proved is $\mathit{update}(al, \mathit{update}(bl, s))(v) = \mathit{update}(bl \oplus (al \triangleleft bl), s)(v)$. The cases when $al = \mathit{nil}$ or $al = al_1 \oplus al_2$ are straightforward from the definitions and the inductive hypothesis. Assume $al = (x, e) \cdot al_1$ for $x \in \mathcal{E}_n, e \in \mathcal{E}$ and $al_1 \in \mathit{Alist}$. Case $x \not\equiv_{\mathit{update}(bl,s)} v$. It follows that $x \triangleleft bl \not\equiv_s v$, if $x \in_{\mathit{update}(bl,s)} al$ then it must occur in $al_1$ and the proof follows from the inductive hypothesis and Lemma (3.3). Case $x \equiv_{\mathit{update}(bl,s)} v$. It follows that $x \triangleleft bl \equiv_s v$ and the result of $\mathit{update}(al, \mathit{update}(bl, s))(v)$ is $\mathcal{I}_e(e)(\mathit{update}(bl, s))$. From the definitions, the result of $\mathit{update}(bl \oplus (al \triangleleft bl), s)(v)$ is $\mathcal{I}_e(e \triangleleft bl)(s)$ and the proof is immediate from the definitions. □

Theorem (3.1) is the semantic basis for the abstraction of assignment 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 $c_2$.

Rules for the substitution operator are given in Figure (3), every substitution expression is the substitution of value expressions. 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 before the expression is reduced to a name $x$; the assignment list is then searched for a value associated with $x$.

## 4 Commands of $\mathcal{L}$

The changes made to a state during the execution of a program are determined by the commands of the program. In the language $\mathcal{L}$, the selection of commands for execution is also a function of the commands. The language $\mathcal{L}$ has a *labelling*, a *conditional* and an *assignment* command. The labelling command associates commands with labels; the label assigned to the name *pc* in a state $s$ identifies the command selected for execution in $s$. The assignment command describes the changes made to the state in which it begins execution as an assignment list which updates the state. It is also used to select a successor, each command makes at least one assignment to the program counter. The conditional command allows the changes made to a state $s$ to depend on the values of the variables in $s$. A Boolean expression $b$ is evaluated in $s$ and depending on the result of the expression, one of two branches is executed.

**Definition 4.1** *Syntax of the commands*

The set of all commands of $\mathcal{L}$ is denoted $\mathcal{C}_0$ and satisfies the grammar:

$$\begin{array}{rcl} \mathit{com} & = & \textbf{if } \langle \mathcal{E}_b \rangle \textbf{ then } \langle \mathit{com} \rangle \textbf{ else } \langle \mathit{com} \rangle \\ & | & := (\langle \mathit{Alist} \rangle, \langle \mathcal{E}_l \rangle) \\ \mathcal{C}_0 & = & \langle \mathit{com} \rangle \mid \langle \mathit{Labels} \rangle : \langle \mathit{com} \rangle \end{array}$$

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

$$\mathcal{C} \stackrel{\text{def}}{=} \{c : \mathcal{C}_0 \mid \exists l', c' : c = l' : c'\}$$

The *successor expression* of assignment command $:= (al, l)$ is $l$ (a label expression). The label of a labelled command $l : c$ is $l$, $\mathit{label}(l : c) \stackrel{\text{def}}{=} l$.

An assignment command made up of simple list will be written using infix notation. e.g. The command $:= ((x_1, e_1) \cdot (x_2, e_2) \cdot \cdots \cdot (x_n, e_n) \cdot \mathit{nil}, l)$ is written $x_1, x_1, \ldots, x_n := e_1, e_2, \ldots, e_n, l$. $\Box$

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

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

$$\frac{f \notin \mathcal{F}_n}{f(a_1, \dots, a_n) \lhd al \equiv f(a_1 \lhd al, \dots, a_n \lhd al)} \qquad \text{(sr5)}$$

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

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

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

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

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

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

Figure 3: Rules for the Substitution Operator

Every assignment command selects a successor by assigning a label expression to the name *pc*. If an assignment command $c$ has assignment list $al$ and label expression $l$ then the assignments of $al$ are made simultaneously with the assignment of $l$ to the program counter. The full list of assignments made by $c$ is therefore $(pc, l) \cdot al$.

## 4.1 Correct Assignment Lists

The assignment command of $\mathcal{L}$ is a simultaneous assignment and its semantics require a means of 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. Every name expression in the assignment list of a command is evaluated in the state in which execution of the command begins, and the correctness of the list depends on this state.

An assignment list formed by the combination of lists is correct in state $s$ iff each name is associated in $s$ with at most one value by each list combined with the operator $\oplus$. The correctness of an assignment list $al$ is determined by considering each simple list which occurs in $al$. A simple assignment list $al$ is correct in a state $s$ iff every value associated with a name $x$ is equivalent in $s$. If $al$ is a combined assignment list then it is correct iff every simple assignment list occurring in $al$ is correct in $s$.

**Definition 4.2** *Correct assignment lists*

For assignment list $al \in \mathit{Alist}$ and state $s$, the name-value pair $(x, v) \in (\mathcal{E}_n \times \mathcal{E})$ occurs in $al$ in state $s$ 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 $v$.

$$\begin{aligned}
\mathit{occs?}((x, e), \mathit{nil})(s) &\stackrel{\text{def}}{=} \mathit{false} \\
\mathit{occs?}((x, e), (x_1, e_1) \cdot al)(s) &\stackrel{\text{def}}{=} (x \equiv_s x_1 \land e \equiv_s e_1) \lor \mathit{occs?}((x, e), al)(s) \\
\mathit{occs?}((x, e), (al \oplus bl))(s) &\stackrel{\text{def}}{=} \mathit{occs?}((x, e), al)(s) \lor \mathit{occs?}((x, e), bl)(s)
\end{aligned}$$

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

$$\mathit{Assoc}(x, al)(s) \stackrel{\text{def}}{=} \{e : \mathcal{E} \mid \mathit{occs?}(x, e)(al)(s)\}$$

The initial prefix of an an assignment list is a simple list.

$$\begin{aligned}
\mathit{initial}(\mathit{nil}) &\stackrel{\text{def}}{=} \mathit{nil} \\
\mathit{initial}((x, e) \cdot al) &\stackrel{\text{def}}{=} (x, e) \cdot \mathit{initial}(al) \\
\mathit{initial}(bl \oplus cl) &\stackrel{\text{def}}{=} \mathit{nil}
\end{aligned}$$

*correct?* is a predicate on assignment lists and states with type $\mathit{Alist} \to \mathit{State} \to \mathit{boolean}$. An assignment list $al$ is correct in a state $s$ iff $\mathit{correct?}(al)(s)$. The predicate *correct?* is defined:

1. *Simple lists*: If $al$ is a simple list, $al \in \mathit{Slist}$, and every name is associated by $al$ with at most one value in a state $s$ then $al$ is correct.

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

2. *Assignment lists*: If every simple list in $al$ is correct in state $s$ then so is $al$.

$$correct?(al) \Leftrightarrow \begin{cases} correct?(initial(al))(s) \\ \wedge(\forall bl, cl : (bl \oplus cl) \ll al \Rightarrow correct?(bl)(s) \wedge correct?(cl)(s)) \end{cases}$$

□

The correctness of an assignment list in an updated state can be be described, syntactically, by updating the assignment list with the newly assigned values.

**Theorem 4.1** *For assignment lists $al, bl \in Alist$ and state $s$,*

$$correct?(al)(update(bl, s)) \Leftrightarrow correct?(al \triangleleft bl)(s)$$

**Proof.** By induction on $al$. The case when $al = nil$ or $al = al_1 \oplus al_2$ is straightforward from the inductive hypothesis. Assume $al = (x, e) \cdot al_1$ for $x \in \mathcal{E}_n$ and $e \in \mathcal{E}$. If $al_1 = nil$ then the property is immediate from the definitions, assume $al_1 \neq nil$.

($\Rightarrow$), the proof for ($\Leftarrow$) is similar: Since $al_1$ is correct, from the inductive hypothesis, every combined assignment list $bl \oplus cl$ occurring in $al$ is also correct. The property to be proved is therefore $correct?(initial(al))(update(bl, s))$. Since $al$ is correct in $update(bl, s)$, there is no name $x_1$ and value $e_1$ such that $occs?((x_1, e_1), al)(update(bl, s))$, $x_1 \equiv_{update(bl,s)} x$ and $e_1 \not\equiv_{update(bl,s)} e$. Assume that $correct?(initial(al \triangleleft bl))(s)$ is *false* then there is a name $x' \in Names$ and value $e' \in Values$ such that $(x', e')$ occurs in $al \triangleleft bl$, $x' \equiv_s x \triangleleft bl$ and $e' \not\equiv_s e \triangleleft bl$. It follows from the definitions (and from $x' \in Names, e' \in Values$) that $x' \equiv_{update(bl,s)} x$ and $e' \not\equiv_{update(bl,s)} e$. From the definition of *initial*, of substitution in an assignment list (Definition 3.12) and of *occs?* that $occs?((x', e'), al)(update(bl, s))$. Since $e' \not\equiv_{update(bl,s)} e$ and from the definition of *Assoc*, there are two distinct values in $Assoc(x, al)(update(bl, s))$. From the definition of *correct?*, the name expression $x$ is uniquely associated with $e$ which is a contradiction. □

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

## 4.2 Semantics of the Commands

The semantics of the commands are defined as relations between the state in which a command begins execution and the state in which it ends. A command $c$ which begins in state $s$ produces a state $t$ from $s$ by assigning values to names. A command labelled with $l$ begins only if it is selected for execution: 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 to terminate.

**Definition 4.3** *Semantics of the commands*

The interpretation function on commands has definition:

$$\begin{array}{rl}
\mathcal{I}_c : \mathcal{C}_0 \to (\mathit{State} \times \mathit{State}) \to \mathit{boolean} & \\
\mathcal{I}_c(l : c)(s,t) \stackrel{\text{def}}{=} \mathcal{I}_e(\mathit{pc})(s) = l \wedge \mathcal{I}_c(c)(s,t) & \\
\mathcal{I}_c(:= (al, l))(s,t) \stackrel{\text{def}}{=} & \begin{cases} \mathit{correct?}((\mathit{pc}, l) \cdot al)(s) \\ \wedge t = \mathit{update}((\mathit{pc}, l) \cdot al, s) \end{cases} \\
\mathcal{I}_c(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2)(s,t) \stackrel{\text{def}}{=} & \begin{cases} \mathcal{I}_c(c_1)(s,t) & \text{if } \mathcal{I}_b(b)(s) \\ \mathcal{I}_c(c_2)(s,t) & \text{if } \neg\mathcal{I}_b(b)(s) \end{cases}
\end{array}$$

□

The commands are deterministic: if a command $c$ begins in a state $s$ then there is a single state in which $c$ can end. Because of this, the relations defining the semantics of the commands can also be considered to be functions transforming states.

**Lemma 4.1** *For command $c \in \mathcal{C}_0$ and states $s, t, u \in \mathit{State}$,*

$$\mathcal{I}_c(c)(s,t) \wedge \mathcal{I}_c(c)(s,u) \Rightarrow t = u$$

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

An assignment command $:= (al, l)$ updates the program counter with the label expression $l$. If $\mathcal{I}_c(:= (al, l))(s,t)$ then $t = \mathit{update}((\mathit{pc}, l) \cdot al, s)$ and the value of $\mathit{pc}$ in $t$ is $\mathcal{I}_l(l)(s)$. Because the program counter, *pc*, acts as a guard on the execution of a labelled command, the assignment commands control the selection of commands for execution.An assignment command of the form $:= (\mathit{nil}, l)$ is a jump command: its only action is to select the command labelled $l$ and since $l$ is a label expression, this is a computed jump. Assignment commands also model the commands which cannot be executed. If assignment list $al$ is always incorrect then the interpretation of command $:= (al, l)$ is always *false*, the command can never terminate.

**Definition 4.4 goto** *and* **abort**

The computed jump command, **goto** $l$, is the assignment command with an empty assignment list and successor expression $l$. The command which always fails, **abort**, has an assignment list which is always incorrect.

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

□

Note that the assignment list of **goto** $l$ is correct in any state $s$, $\mathit{correct?}((\mathit{pc}, l) \cdot \mathit{nil})(s)$, and **goto** $l$ can be executed in any state. The command **abort** can never be executed since both **true** and **false** are assigned to the program counter, the assignment list of **abort** is never correct.

When a command of $\mathcal{L}$ is selected in state $s$, the command is said to be *enabled* in $s$. The command can begin execution in $s$ and must either terminate and update state $s$, or fail to terminate. If the command fails to terminate then it is said to *halt* in state $s$.

| Instruction | Command of $\mathcal{L}$ |
|---|---|
| `ldl` $\mathbf{r}_1, \mathtt{v}(\mathbf{r}_2)$ | $\mathbf{r}_1 := \mathbf{mem}(\mathbf{r}_2 +_{64} v), pc +_{64} 4$ |
| `stl` $\mathbf{r}_1, \mathtt{v}(\mathbf{r}(\mathrm{b}))$ | $\mathbf{mem}(\mathbf{r}_2 +_{64} v) := \mathbf{Long}(\mathbf{r}_1), pc +_{64} 4$ |
| `addl` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ | $\mathbf{r}_3 := \mathbf{Long}(\mathbf{r}_1 +_{64} \mathbf{r}_2), pc +_{64} 4$ |
| `cmpule` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ | **if** $\mathbf{r}_1 \leq \mathbf{r}_2$ **then** $\mathbf{r}_3 := 1, pc +_{64} 4$ **else** $\mathbf{r}_3 := 0, pc +_{64} 4$ |
| `br` $\mathbf{r}_1, \mathtt{v}$ | $\mathbf{r}_1 := pc +_{64} 4, \mathbf{inst}(pc +_{64} 4 +_{64} v)$ |
| `beq` $\mathbf{r}_1, \mathtt{v}$ | **if** $\mathbf{r}_1 =_{64} 0$ |
| | **then** $\mathbf{r}_1 := pc +_{64} 4, \mathbf{inst}(pc +_{64} 4 +_{64} v)$ |
| | **else goto** $pc +_{64} 4$ |
| `jsr` $\mathbf{r}_1, \mathbf{r}_2$ | $\mathbf{r}_1 := pc +_{64} 4, \mathbf{inst}(\mathbf{r}_2)$ |

where $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3 \in \{\mathbf{r0}, \ldots, \mathbf{r30}\}, v \in \mathit{Values}$

Figure 4: Example of Alpha AXP instructions

**Definition 4.5** *Halts*

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

$$\begin{aligned}&\mathit{halt?} : \mathcal{C} \to \mathit{State} \to \mathit{boolean}\\&\mathit{halt?}(c)(s) \stackrel{\text{def}}{=} pc \equiv_s \mathit{label}(c) \land \forall (t : \mathit{State}) : \neg \mathcal{I}_c(c)(s,t)\end{aligned}$$

□

For any label $l$, the command $l$ : **abort** halts in every state. The command $l$ : **goto** $l$ never halts and is the command which performs no action: if it is enabled in state $s$ then it will also terminate in $s$ (and be re-selected for execution).

**Example 4.1** *Alpha AXP: Instructions*

The Alpha AXP processor language includes instruction to access data items in memory; to perform arithmetic operations on these items and to control the flow of control through a program. Representative instructions for each of these classes, described as commands of $\mathcal{L}$, are given in Figure (4). The instructions are given with registers $\mathbf{r}_1, \mathbf{r}_2 \in \mathit{Names}$ and $v \in \mathit{Values}$. Note that $\mathbf{r}_1$ and $\mathbf{r}_2$ may be any of the registers $\mathbf{r0}, \ldots, \mathbf{r30}$.

*Data movement:* Instructions `ldl` and `stl` move long-words between registers and memory locations using indirect addressing to identify the memory variable to access. Instruction `ldl` $\mathbf{r}_1, \mathtt{v}(\mathbf{r}_2)$ the long-word in memory location $\mathbf{mem}(\mathbf{r}_2 +_{64} v)$ to register $\mathbf{r}_1$. Instruction `stl` $\mathbf{r}_1, \mathtt{v}(\mathbf{r}_2)$ stores the long-word value of register $\mathbf{r}_1$ in the memory variable $\mathbf{mem}(\mathbf{r}_2 +_{64} v)$.

*Arithmetic instructions:* Arithmetic instructions are used to calculate and to compare values. Instruction `addl` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ implements addition of long-words, storing the sum of register $\mathbf{r}_1$ and register $\mathbf{r}_3$ in register $\mathbf{r}_3$. The comparison instruction `cmpule` $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ sets $\mathbf{r}_3$ to $1$ if $\mathbf{r}_1$ is less than or equal to $\mathbf{r}_2$.. The test is calculated as the Boolean expression (of $\mathcal{E}_b$): $\mathbf{r}_1 \leq_{64} \mathbf{r}_2 \stackrel{\text{def}}{=} (\mathbf{r}_1 =_{64} \mathbf{r}_2)$ **or** $(\mathbf{r}_1 <_{64} \mathbf{r}_2)$.

*Program control:* Instructions for program control are made up of conditional and unconditional jumps (called branches) and computed jumps, used to pass control to and from sub-routines. The instruction `br` $\mathbf{r}_1, \mathtt{v}$, assigns the label of the next instruction to register $\mathbf{r}_1$ and passes control to the instruction with label $pc +_{64} 4 +_{64} v$. The conditional jump instruction `beq` $\mathbf{r}_1, \mathtt{v}$ passes control to the instruction at label $pc +_{64} 4 +_{64} v$ if $\mathbf{r}_1 =_{64} 0$. The computed jumps pass control to a target identified by a register. Instruction `jsr` $\mathbf{r}_1, \mathbf{r}_2$ stores the label of the next instruction in register $\mathbf{r}_1$ and passes control to the label stored in $\mathbf{r}_2$. The return from sub-routine, instruction `rts`, has the same semantics as the jump to sub-routine, instruction `jsr`.

*Byte sized memory access:* Although it is assumed here that all memory access is of aligned long-words, it is straightforward to base the semantics of the instructions on byte sized access. For example, the instruction `ldbu` $\mathbf{r}_1, \mathtt{v}(\mathbf{r}_2)$ is the byte size equivalent of the instruction `ldl`. Its semantics can be described by the command $\mathbf{r}_1 := \mathbf{B}(0)(\mathbf{ref}(\mathbf{r}_2 +_{64} v)), pc +_{64} 4$. The name function **ref** is used to model memory access and the value function **B** ensures that the value is a byte.

Long-word access is used in the examples to simplify the presentation. For example, using byte sized access to model the instruction `stl` $\mathbf{r}_1, \mathtt{v}(\mathbf{r}_2)$ requires an assignment to the four memory locations in which the long-word is stored. The semantics of the instruction would be described by the command:

$$\begin{array}{l}\mathbf{ref}(a),\mathbf{ref}(a +_{64} 1), \mathbf{ref}(a +_{64} 2), \mathbf{ref}(a +_{64} 3) \\ \qquad := \mathbf{B}(0)(\mathbf{r}_1), \mathbf{B}(1)(\mathbf{r}_1), \mathbf{B}(2)(\mathbf{r}_1), \mathbf{B}(3)(\mathbf{r}_1), pc +_{64} 4 \\ \text{where } a = \mathbf{r}_2 +_{64} v.\end{array}$$

Byte sized memory access adds complexity to the expressions used in the semantics of instructions. It does not otherwise affect the ability to define processor instructions in the language $\mathcal{L}$. □

# 5 Programs of $\mathcal{L}$

The language $\mathcal{L}$ is a flow-graph language, the order in which commands are executed is independent of the syntax of a program. Because the flow of control is determined by the selection of each command of its successor, it is enough for a program of $\mathcal{L}$ to uniquely identify its commands and the programs of $\mathcal{L}$ are sets of uniquely labelled commands. The labels index the commands of a program: if the program $p$ contains a command $l : c$ then the label $l$ is enough to obtain that command from program $p$.

**Definition 5.1** *Programs*

A set $p$ of labelled commands is a program of $\mathcal{L}$ iff every command in $p$ is uniquely labelled.

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

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

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

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

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

□

| Alpha Program | | $\mathcal{L}$ program |
|---|---|---|
| `swap :` | `bis` $\$2, \$0, \$0$ | $l_1 : \mathbf{r2} := \mathbf{r0}, l_2$ |
| | `bis` $\$0, \$1, \$1$ | $l_2 : \mathbf{r0} := \mathbf{r1}, l_3$ |
| | `bis` $\$1, \$2, \$2$ | $l_3 : \mathbf{r1} := \mathbf{r2}, l_4$ |

Figure 5: Alpha AXP: Register swapping

Every subset of a program $p \in \mathcal{P}$ is a program and, in particular, the empty set is a program. A program does not specify an initial command: execution can begin with any command and a program can contain commands which cannot be or are never executed. Moreover, the order in which commands of the program are executed is independent of any ordering of the labels.

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 $p'$ obtained 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 5.2** *Construction*

If there is no command in a program $p$ labelled $\mathit{label}(c)$ 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 combination of a 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 $p_1$, to fold changes made to a subset of the program into the program.

**Example 5.1** *Alpha AXP: Programs*

The Alpha AXP program of Figure (5) swaps the values of registers **r0** and **r1** using register **r2** as an intermediate variable. The movement of data between registers is by the instruction `bis` $\mathbf{r}_0, \mathbf{r}_1, \mathbf{r}_2$, which stores the bitwise disjunction of $\mathbf{r}_1$ and $\mathbf{r}_2$ in $\mathbf{r}_0$. The bitwise disjunction of a register $\mathbf{r}$ and itself is **r**. In the equivalent program of $\mathcal{L}$, the commands are uniquely labelled and for each $l_i, l_{i+1}$, $1 \leq i \leq 3$, $l_{i+1} \equiv l_i +_{64} 4$. In general, where the successor expression of a command is *pc* $+_{64} 4$ and the commands label $l \in$ *Labels* is known, the constant $l' \in$ *Labels*, $l' \equiv$ *pc* $+_{64} 4$, will be used instead.

□

## 5.1 Semantics of the Programs

A program of $\mathcal{L}$ is executed by the repeated selection and execution of its commands. The program relates the state in which it begins execution to the states which are produced during its execution: a program $p$, beginning in state $s$, *leads to* state $t$ if execution of $p$ beginning in $s$ eventually produces state $t$. The program defines a *transition relation* between the two states (Manna, 1974; Cousot, 1981; Gordon, 1994), which can be used to verify the liveness properties of the program (Manna & Pnueli, 1991). Transition relations can also be used to compare programs, based on the states which each program produces. This is the basis for a refinement relation between programs, which is used to show that one program is the abstraction of another.

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 5.3** *Leads to*

The *leads to* relation from state $s$ to state $t$ through program $p \in \mathcal{P}$ is written $s \overset{a}{\leadsto} t$ and satisfies:

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

For any command $c \in \mathcal{C}$ and states $s, t \in$ *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 be written $s \overset{c}{\leadsto} t$. □

The commands of a program $p$ will also be the commands of a superset $p'$ of $p$ and any states produced by $p$ will also be produced by $p'$. Because the relation *leads to* is defined on a set of commands, in *Set*$(\mathcal{C})$, and not only for programs, in $\mathcal{P}$, the set through which a state leads to another can be extended arbitrarily.

The relation *leads to* 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 semantics of program $p$ are defined in terms of the states related by the relation *leads to* through the program $p$.

**Definition 5.4** *Semantics of programs*

The interpretation function $\mathcal{I}_p$ applied to program $p \in \mathcal{P}$ and states $s, t \in$ *State* is *true* iff $p$ beginning in $s$ eventually produces $t$.

$$\begin{aligned}
&\mathcal{I}_p : \mathcal{P} \to (\mathit{State} \times \mathit{State}) \to \mathit{boolean} \\
&\mathcal{I}_p(p)(s,t) \overset{\text{def}}{=} s \overset{p}{\leadsto} t
\end{aligned}$$

□

A program terminates when a state is produced in which no command of the program is selected for execution. 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 and terminating programs, whether a program terminates is not decidable from the syntax of the program commands.

The programs have a number of basic properties: the interpretation of programs is transitive: if $\mathcal{I}_p(p)(s,u)$ and $\mathcal{I}_p(p)(u,t)$ then $\mathcal{I}_p(s,t)$ for any $p \in \mathcal{P}$ and $s,t,u \in$ *State* and if a command of a program relates two states then so does the program. Only one command of a program can be selected in any state, if command $c$ of program $p$ is enabled in a state $s$ and $c$ halts then the program $p$ must also halt. No other command can be executed in $s$, since each command is uniquely labelled, and no successor to command $c$ can be selected.

**Definition 5.5** *Program halts*

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

$$\mathit{halt?}(p)(s) \stackrel{\text{def}}{=} \exists(c \in p) : \mathit{halt?}(c)(s)$$

□

Programs $p, p' \in \mathcal{P}$ are equivalent iff programs $p$ and $p'$ produce the same states. Transforming a program $p$ by replacing commands of $p$ with equivalent commands will result in a program $p'$ which is equivalent to $p$.

**Lemma 5.1** *For any programs $p, p' \in \mathcal{P}$ and states $s, t$,*

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

**Proof.** By induction on *leads to*. ($\Rightarrow$), the case for ($\Leftarrow$) is similar. The inductive case is straightforward from the hypothesis. For the base case: from $\mathcal{I}_p(p)(s,t)$, there is a command $c \in p$ such that $\mathcal{I}_c(c)(s,t)$. From the assumptions, there is also a command $c' \in p'$ such that $\mathcal{I}_c(c')(s,t)$. The proof follows from the definition of $\mathcal{I}_p$. □

A useful transformation using the property of Lemma (5.1) is the systematic replacement of the program counter *pc* with the label of the command in which it appears.

**Example 5.2** Let $l, l_1, l_2$ be distinct labels and $c_1$ be some command. The program $p = \{l_1 : c_1, l_2 :$ **abort**$\}$ halts for all states in which $l_2$ : **abort** is enabled and also for all states $s, t$ such that $\mathcal{I}_c(l_1 : c_1)(s,t)$ and $pc \equiv_t l_2$.

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

## 5.2 Refinement of Programs

Refinement is mostly used in the development of programs, to show that a program satisfies its specification (Morgan, 1990) or to show that a compiler is correct (Hoare et al., 1993; Bowen and He Jifeng, 1994). A program $p$ is refined by program $p'$ if every state produced by $p$ is also produced by $p'$. Program $p$ is an abstraction of $p'$ and, to show that $p'$ produces a state $t$ satisfying a property, it is enough to show that $t$ can be produced by program $p$. Although $p'$ can produce more states than the abstraction $p$, attempting to show that $p$ produces a state which it does not can only lead to a failure of the proof and not to an incorrect proof.

**Definition 5.6** *Refinement between programs*

Program $p_1$ is refined by $p_2$, written $p_1 \sqsubseteq p_2$, if any states related by $p_1$ are related by $p_2$.

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

□

Defining refinement in terms of the relation *leads to* allows the properties established by a program to be separated from the number of program commands. An abstraction $p'$ of program $p$, $p' \sqsubseteq p$, may contain fewer commands but any state produced by $p'$ will be produced by $p$.

For verification the most useful property of refinement is that it is transitive. 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 $p$. A second property, that a program $p$ is always a refinement of any subset of $p$, allows a program to be verified or manipulated by considering a part of the program rather than the whole.

**Theorem 5.1** *Properties of refinement*

*For programs $p, p_1, p_2 \in \mathcal{P}$ and $s, t \in \mathit{State}$,*

1. *Refinement is transitive.*

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

2. *The abstraction $p_1$ of any program $p_2$ is an abstraction of any superset of $p_2$.*

$$
\frac{p_1 \sqsubseteq p_2 \quad p_2 \subseteq p}{p_1 \sqsubseteq p}
$$

**Proof.**

1. Definition of refinement and transitivity of $\mathcal{I}_p$ (from transitivity of *leads to*).
2. Since $p$ is a superset of $p_2$, for any $s, t \in \mathit{State}$, if $s \stackrel{p_2}{\leadsto} t$ then $s \stackrel{p}{\leadsto} t$. The conclusion follows immediately from this, from the assumption $p_1 \sqsubseteq p_2$ and from the definition of refinement.

□

A particular application of Item (2) of Theorem (5.1), is the abstraction of a program by abstracting subsets of the program. However, the subset must be chosen with care since the empty set is refined by any program: for any $s, t \in State$, $\mathcal{I}_p(\{\})(s,t) = false$ and $\{\} \sqsubseteq p$ for any program $p \in \mathcal{P}$. More generally, if a program always fails then it is refined by any other program.

**Lemma 5.2** *For programs $p_1, p_2 \in \mathcal{P}$ and states $s, t \in State$,*

$$\frac{(\forall s, t : \mathcal{I}_p(p_1)(s,t) = false)}{p_1 \sqsubseteq p_2}$$

**Proof.** By definition of refinement, $p_1 \sqsubseteq p_2$ reduces to $false \Rightarrow \mathcal{I}_p(p_2)(s,t)$ which is trivially *true*. □

Lemma (5.2) is an instance of a standard property of refinement: any program is an improvement on the program which always fails (Back & von Wright, 1989).

# 6 Abstraction

The abstraction of a program is formed by constructing a single command $c$, which abstracts from two program commands, then combining $c$ with the original program. Assume $c_1, c_2 \in \mathcal{C}$ are commands of program $p$, $c_1, c_2 \in p$, and that command $c \in \mathcal{C}$ abstracts $c_1$ and $c_2$. The abstraction $p'$ of program $p$ is formed as the combination of $c$ with $p$, $p' = p \uplus \{c\}$ and $p' \sqsubseteq p$. This method does not necessarily reduce the number of commands in the program but can reduce the number of commands which must be considered during the course of a proof.

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

$$\frac{\exists (u : State) : \mathcal{I}(c_1)(s,u) \wedge \mathcal{I}(c_2)(u,t)}{\mathcal{I}(c)(s,t)} \tag{1}$$

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

$$\frac{\mathcal{I}(c_1)(s,t) \quad pc \not\equiv_t label(c_1)}{\mathcal{I}(c)(s,t) = \mathcal{I}(c_1)(s,t)} \tag{2}$$

Any command which has both these properties can replace $c_1$ in a program.

The sequential composition operator of the structured languages (Hoare, 1969; Loeckx & Sieber, 1987) constructs the abstraction of two commands. This operator is usually a primitive syntactic construct, with $c_1; c_2$ considered a compound command of the language (see Hoare, 1969, Dijkstra, 1976 or Francez, 1992). The interpretation of this construct is

$$\mathcal{I}(c_1; c_2)(s,t) = \exists u : \mathcal{I}(c_1)(s,u) \wedge \mathcal{I}(c_2)(u,t) \tag{3}$$

which satisfies Property (1) but not Property (2). For $c_1 = \textbf{goto}\ l_1$, $c_2 = l_2 : \textbf{goto}\ l_1$, $l_1, l_2 \in Labels$ and $l_1 \neq l_2$, the interpretation would alway be *false*.

The sequential composition operator $\_;\_$ is defined here as a function on the commands of $\mathcal{L}$ which ranges over the set of commands $\mathcal{C}_0$. The definition is based on the the algebraic laws described of

Hoare et al. (1987), extended to take into account pointers and computed jumps. The substitution expressions of $\mathcal{L}$ and the combination of assignments lists, *Alist*, provide the operations on pointers. To treat the flow of control correctly, the sequential composition of commands $c_1, c_2$ uses the label of $c_2$ to guard execution of $c_2$. This forms a conditional command which compares the program counter and the label of $c_2$. If the two are equal then $c_1; c_2$ is the command satisfying Property (1); if the two are distinct then $c_1; c_2$ is the command satisfying Property (2).

## 6.1 Sequential Composition of Commands

The sequential composition of commands $c_1, c_2 \in \mathcal{C}_0$ is a command of $\mathcal{L}$ in the set $\mathcal{C}_0$. The sequential composition operator is 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 6.1** *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 then $c_2$ is composed with the command being labelled. When $c_1$ is a conditional command, the composition with $c_2$ is pushed into the branches of the conditional.

**Definition 6.2** *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 **if** $b$ **then** $c_1$ **else** $c_2$ with $c$ 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 then the composition of a labelled command $l : c_1$ with $c$, $l : c_1; c$, is labelled $l$ 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 $b$ is **true** and is equivalent to $c_f; c$ whenever $b$ is **false**.

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 $l : c$, the command $c$ can be executed only if the successor expression of the assignment command is equivalent to $l$.

**Definition 6.3** *Assignment commands*

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

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

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

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

□

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.

**Definition 6.4** *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 $:= (bl, l_2)$.

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

□

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

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

From the definition for *correct?*, this is equivalent to the correctness of $al$, $bl \triangleleft ((pc, l_1) \cdot al)$ and of $(pc, l_2 \triangleleft ((pc, l_1) \cdot al)) \cdot nil$. 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 $(bl \triangleleft (pc, l_1) \cdot al)$ in state $s$ is equivalent to the correctness of *bl* in $u$ and follows from the semantics of $:= (bl, l_2)$. The correctness of $(pc, l_2 \triangleleft ((pc, l_1) \cdot al)) \cdot nil$ is immediate.

## 6.2 Properties of Sequential Composition

Proofs for the properties of composition make use of the fact that the initial prefix $(pc, l)$ of an assignment list, which is present in the lists of composed assignment commands (Definition 6.4), can be removed.

**Lemma 6.1** *For $x \in \mathcal{E}_n$, $e \in \mathcal{E}$, $al, bl \in Alist$ and $s \in State$,*

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

**Proof.** By extensionality, with $n \in \mathit{Names}$. Assume $n \equiv_s x \triangleleft al$. By definition, $\mathit{update}(((x,e) \triangleleft al) \cdot (al \oplus ((x,e) \cdot bl \triangleleft al)), s)(n)$ is $\mathcal{I}_e(e \triangleleft al)(s)$. Also by definition, $\mathit{update}(al \oplus ((x,e) \cdot bl \triangleleft al)(n)$ is $\mathcal{I}_e(e \triangleleft al)(s)$, since $(x,e) \cdot bl \triangleleft al$ is search first (definition of *update* and *find*). This completes the proof for this case. Assume $n \not\equiv_s x \triangleleft al$. By definition of *update* and *find*, if $n$ occurs in the assignment lists then it must do so either in $((x,e) \cdot bl) \triangleleft al$ or in $al$. The initial assignment $(x,e) \triangleleft al$ is irrelevant, in this case, and the proof is straightforward from the definitions. □

Sequential composition satisfies Property (1) and Property (2). For any two commands $c_1, c_2 \in \mathcal{C}_0$ and states $s, u, t$, if $c_1$ begins in state $s$ and ends in state $u$ and $c_2$ begins in state $u$ and ends in state $t$ then $c_1; c_2$ also begins in state $s$ and ends in state $t$.

**Theorem 6.1** *Property (1)*

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

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

**Proof.** By induction on $c_1$ followed by induction on $c_2$. The cases when either $c_1$ or $c_2$ is a conditional or a labelled command are straightforward from the inductive hypothesis. Assume $c_1$ and $c_2$ are assignment commands: $c_1$ is $:= (al, l_1)$ and $c_2$ is $:= (bl, l_2)$. If $\mathcal{I}_c(c_1)(s,u)$ then, by definition of $\mathcal{I}_c$ $u = \mathit{update}((pc, l_1) \cdot al, s)$. Similarly, if $\mathcal{I}_c(c_2)(u,t)$ then $t = \mathit{update}((pc, l_2) \cdot bl, u)$. The correctness of the assignment list of $c_1; c_2$ is straightforward from assumptions $\mathcal{I}_c(c_1)(s,u)$ and $\mathcal{I}_c(c_2)(u,t)$, from $u = \mathit{update}((pc, l_1) \cdot al, s)$, the definition of *correct?* and from Theorem (4.1). The definition of $\mathcal{I}_c(c_1; c_2)(s,t)$ requires that $t$ is $\mathit{update}((pc, l_2) \triangleleft ((pc, l_1 \cdot cl), s))$, where $cl$ is $((pc, l_1) \cdot al) \oplus (bl \triangleleft (pc, l_1) \cdot al)$. Note that from Theorem (3.1), $\mathit{update}((pc, l_2) \cdot bl, \mathit{update}((pc, l_1) \cdot al, s))$ is equivalent to $\mathit{update}((pc, l_1) \cdot al \oplus ((pc, l_2) \cdot bl) \triangleleft (pc, l_1) \cdot al, s)$. That this is equivalent to $t$ is straightforward by extensionality. □

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 6.2** *Property (2)*

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

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

**Proof.** By induction on $c_1$, the cases when $c_1$ is a labelled or a conditional command are straightforward from the inductive hypothesis. Assume $c_1$ is the assignment command $:= (al, l_1)$. By Definition (6.3), $c_1; (l : c_2)$ is a conditional command with the test $\mathbf{equal}(l_1, l)$ and false branch $c_1$. From the assumption, $\mathcal{I}_c(c_1)(s,t)$, state $t$ is $\mathit{update}((pc, l_1) \cdot al, s)$. It follows that $pc \not\equiv_t l$ is $pc \triangleleft ((pc, l_1) \cdot al) \not\equiv_s l \triangleleft (pc, l_1) \cdot al$. Substituting for $pc$, and for the constant $l$, this is $l_1 \not\equiv_s l$. It follows that the test $\mathcal{I}_b(\mathbf{equal}(l_1, l))(s)$ is *false* and therefore $\mathcal{I}_c(c_1; l : c_2)(s,t)$ is equivalent to $\mathcal{I}_c(c_1)(s,t)$. □

Let $c$ be the result of the composition of any two commands $c_1, c_2 \in \mathcal{C}_0$. 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 6.3** *For commands $c_1, c_2 \in \mathcal{C}_0$, $l_1, l_2 \in$ Labels and states $s, t$,*

$$\frac{\mathcal{I}_c(l_1 : c_1; l_2 : c_2)(s,t)}{(\exists(u : \mathit{State}) : \mathcal{I}_c(l_1 : c_1)(s,u) \wedge \mathcal{I}_c(l_2 : c_2)(u,t)) \vee (\mathcal{I}_c(c_1)(s,t) \wedge pc \not\equiv_t l_2)}$$

**Proof.** By induction on $c_1$ followed by induction on $c_2$. The cases when either is a labelled or a conditional command are straightforward from the induction hypothesis. Let $c_1$ be the assignment command $:= (al, l_3)$ and $c_2$ the command $:= (bl, l_4)$. Assume $l_1 \equiv_s l_2$ and let $u = \mathit{update}((pc, l_3) \cdot al, s)$. It follows, from the assumption, that $(pc, l_3) \cdot al$ is correct in $s$, since it is contained in the assignment list of $c_1; c_2$. That $\mathcal{I}_c(c_1)(s,u)$ is *true* follows from the definition of $\mathcal{I}_c$. Since the assignment list of $c_1; c_2$ also contains $(pc, l_4) \cdot bl \triangleleft (pc, l_3) \cdot al$ and is correct in $s$, the list $(pc, l_4) \cdot bl$ must be also correct in $\mathit{update}((pc, l_3) \cdot al, s)$ (Theorem 4.1). From Theorem (3.1), state $t$ is $\mathit{update}((pc, l_3) \cdot al) \oplus ((pc, l_4) \cdot bl \triangleleft (pc, l_3) \cdot al), s)$ which is equivalent to $\mathit{update}((pc, l_4) \cdot bl, \mathit{update}((pc, l_3) \cdot al), s))$ (Theorem 3.1 and Lemma lem:3.4). The interpretation $\mathcal{I}(l_2 : c_2)(u,t)$ is therefore *true*, completing the proof for this case. Assume that $l_1 \not\equiv_s l_2$. By definition, $l_1 : c_1; l_2 : c_2$ is a conditional command with test $\mathbf{equal}(l_3, l_2)$ and false branch $c_1$. As in Theorem (6.2), it follows that $\mathcal{I}_c(c_1)(s,t)$ must be *true*, completing the proof. □

Theorems (6.1) to (6.3) 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 so does either $c_1$ or $c_2$. Conversely, if either command $c_1$ or $c_2$ halts then so does $c_1; c_2$.

**Theorem 6.4** *For command $c_1, c_2 \in \mathcal{C}$ and states $s, t, u \in$ State,*

1. *If $c_1; c_2$ halts in a state $s$ then $c_1$ either halts in $s$ or produces a state $u$ in which $c_2$ halts.*

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

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

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

3. *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 $s$.*

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

**Proof.**

1. Assume there is a state $u$ such $\mathcal{I}_c(c_1)(s,u)$ and a state $t$ such that $\mathcal{I}_c(c_2)(u,t)$. From Theorem (6.1), it follows that $\mathcal{I}_c(c_1; c_2)(s,t)$ contradicting the assumption, $\mathit{halt?}(c_1; c_2)$ that there is no state such that $\mathcal{I}_c(c_1; c_2)(s,t)$.
2. Assume there is a state $t$ such that $\mathcal{I}(c_1; c_2)(s,t)$ (and therefore $\neg \mathit{halt?}(c_1; c_2)(s)$). From Theorem (6.3), there is a state $u$ such that $\mathcal{I}_c(c_1)(s,u)$. This is a contradiction since the assumption, $\mathit{halt?}(c_1)(s)$, is that there is no such state.

3. From the assumption, *halt?*$(c_2)(u)$, command $c_2$ is enabled in $u$ and therefore *pc* $\equiv_u$ *label*$(c_2)$. Assume that there is a state $t$ such that $\mathcal{I}(c_1; c_2)(s, t)$. From Theorem (6.3), and from *pc* $\equiv_u$ *label*$(c_2)$, there is a state $u' \in$ *State* such that $\mathcal{I}_c(c_1)(s, u')$ and $\mathcal{I}_c(c_2)(u', t)$. The commands are deterministic (Lemma 4.1) therefore $u' = u$ and $\mathcal{I}_c(c_2)(u, t)$. This contradicts the assumption *halt?*$(c_2)(u)$.

□

Theorems (6.4) and (6.4), together with the earlier theorems, show that if the composition $c_1; c_2$ establishes a property then so will the commands $c_1$ and $c_2$, executed in sequence. If the composition $c_1; c_2$ cannot be executed or cannot establish the property, neither can the two commands considered individually. The advantage of sequential composition is that it is simpler to show that $c_1; c_2$ establishes a property than to verify $c_1$ and $c_2$ individually. Any property of the two commands, executed in sequence, can be established directly from the command $c_1; c_2$.

A property of sequential composition in a flow-graph language is that it is not associative. The label of a command determines whether the command is enabled in a state. The composition of command $l_1 : c_1$ with $c_2$, $(l_1 : c_1); c_2$ has label $l_1$. 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 $l_1 : c_1$.

**Example 6.1** 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** $l_4$. □

## 6.3 Applying Sequential Composition

The commands which result from sequential composition can be complex. To ensure that composition has Property (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.

To replace a command $c$ of a program with a simplified command $c'$, command $c$ must be equivalent to $c'$ in all states, $\mathcal{I}_c(c)(s, t) = \mathcal{I}_c(c')(s, t)$ (Lemma 5.1). This is possible by the replacement of expressions in the command with strongly equivalent expressions. The conditions for strong equivalence of expressions can often be established from the syntax of the commands and, in these cases, the simplification of a command can be carried out mechanically. For example, assume $l_1 = l$: in the conditional command **if** $(l_1 =_o l)$ **then** $c_1; c_2$ **else** $c_1$, the expression $(l_1 =_o l)$ can be replaced with **true**. From the semantics of the conditional command, the result is a command equivalent to $(c_1; c_2)$. This method is similar to the techniques used for symbolic execution (King, 1971).

**Example 6.2** 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:

$$\begin{aligned}\textbf{if } (x =_o v_2) \textbf{ then } & (\textbf{ref}(a(x)) \triangleleft (a(v_2), v_1) := e_1, l) \\ \textbf{else } & (\textbf{ref}(a(x)) \triangleleft (a(v_2), v_1) := e_2, l)\end{aligned}$$

is equivalent to the command:

$$\begin{aligned}\textbf{if } (x =_o v_2) \textbf{ then } & (\textbf{ref}(v_1) := e_1, l) \\ \textbf{else } & (\textbf{ref}(a(x) \triangleleft (a(v_2), v_1) := e_2, l)\end{aligned}$$

□

## 6.4 Abstraction of Programs

The method for abstracting from programs is to choose two commands $c_1, c_2$ of a program $p$ and form the subset $\{c_1, c_2\}$ of $p$. Sequential composition is applied to the two commands to obtain the singleton set $\{(c_1; c_2)\}$. This is combined with the original program $p$, removing command $c_1$ from $p$, to obtain the abstraction $p \uplus \{(c_1; c_2)\} \sqsubseteq p$. This can be repeated any number of times, allowing a sequence of commands to combined by sequential composition.

This method of abstraction is based on two properties of composition and refinement: the first that the singleton set $\{(c_1; c_2)\}$ is an abstraction of the set $\{c_1, c_2\}$.

**Theorem 6.5** *Composition forms an abstraction*

*Assume commands $c_1, c_2 \in \mathcal{C}$ with distinct labels so that $\{c_1, c_2\} \in \mathcal{P}$.*

$$\{(c_1; c_2)\} \sqsubseteq \{c_1, c_2\}$$

**Proof.** From the definition of refinement, the property to prove is that, for any $s, t \in$ *State*, if $s \overset{\{(c_1;c_2)\}}{\rightsquigarrow} t$ then $s \overset{\{c_1,c_2\}}{\rightsquigarrow} t$. By induction on $\rightsquigarrow$, the inductive case is immediate from the hypothesis. Base case, $\mathcal{I}_c(c_1; c_2)(s, t)$: from Theorem (6.3), either there is an intermediate state $u$ such that $\mathcal{I}_c(c_1)(s, u)$ and $\mathcal{I}_c(c_2)(u, t)$ or $\mathcal{I}_c(c_1)(s, t)$. In either case, the proof is immediate from the definition of *leads to*. □

The second property, of refinement, states that combining the singleton set $\{(c_1; c_2)\}$ with the program $p$ results in an abstraction of $p$. This is based on the ability to form an abstraction of a program $p$ by combining $p$ with any abstraction of $p$.

**Lemma 6.2** *For programs $p_1, p_2 \in \mathcal{P}$,*

$$\frac{p_1 \sqsubseteq p_2}{(p_1 \uplus p_2) \sqsubseteq p_2}$$

**Proof.** By definition of refinement, the property to prove is: if $\forall s, t : s \overset{p_1}{\rightsquigarrow} t \Rightarrow s \overset{p_2}{\rightsquigarrow} t$ then $\forall s, t : s \overset{(p_1 \uplus p_2)}{\rightsquigarrow} t \Rightarrow s \overset{p_2}{\rightsquigarrow} t$. By induction on $s \overset{(p_1 \uplus p_2)}{\rightsquigarrow} t$. The inductive case is straightforward from the hypothesis and from the transitivity of *leads to*. Base case, $c \in (p_1 \uplus p_2)$ and $\mathcal{I}_c(c)(s, t)$: if $c \in p_2$ then the proof is immediate from the definition of *leads to*. If $c \in p_1$ then $s \overset{p_1}{\rightsquigarrow} t$ and the proof follows from the assumption. □

Lemma (6.2) justifies the method used to combine an abstraction with a program. The program resulting from $p_1 \uplus p_2$ is the union of $p_1$ with the commands of $p_2$ which do not share a label with a command of $p_1$. Using the properties of Theorem (5.1) and Lemma (6.2), an abstraction of $p_2$ can be constructed by selecting a subset $p'$ of $p_2$ and manipulating $p'$ to construct an abstraction $p_1$ of $p'$. This can then be merged into the original program $p_2$ to obtain the abstraction of $p_2$.

**Theorem 6.6** *Abstraction of Programs*

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

$$\frac{c_1 \in p \quad c_2 \in p}{(p \uplus \{c_1; c_2\}) \sqsubseteq p}$$

**Proof.** Straightforward, by Lemma (6.2) and Theorem (6.5). □

The program resulting from $(p \uplus \{c_1; c_2\})$ is equivalent to $(p - \{c\}) \cup \{c_1; c_2\}$, replacing the command $c_1$ with $c_1; c_2$. Because $c_2$ may be the target of a jump, it is neither replaced nor removed.

There is no restriction on the choice of program commands $c_1$ and $c_2$, although it will normally be made to simplify verification of the program. In verification, the commands of a program are considered in the order which they are executed, the commands would therefore be chosen in the order in which they may be executed. It is also possible to construct abstractions from the abstraction of a program. Theorem (6.6) allows an abstraction to be constructed for any program $p$ of $\mathcal{L}$, including those formed by abstraction. Since refinement is transitive, the construction can be repeated an arbitrary number of times; the result will also be an abstraction of the original program.

# 7 Proof Rules

A program logic suitable for proving the liveness properties of a program, based on the method of intermittent assertions (Burstall, 1974; Cousot & Cousot, 1993), can be defined using the transition relation *leads to*. A formula of the logic associates a precondition $P$ with the state $s$ in which execution of program $p$ begins and asserts that eventually a state $t$ is produced which satisfies a postcondition $Q$. The proof rules for commands are defined using a *wp* predicate transformer (Dijkstra, 1976). The rules for programs are similar to rules defined by Francez (1992) and also allow a program to be replaced with an abstraction.

## Assertions

The pre- and postconditions of a program are assertions on states, describing the properties to be satisfied by the program variables. An assertion is a predicate on a state, the assertion language is denoted $\mathcal{A}$ and contains the operators of a first order logic, the Boolean expressions of $\mathcal{L}$ and a substitution operator.

**Definition 7.1** *Assertion Language*

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

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

An assertion $P \in \mathcal{A}$ is *valid* if it is *true* for all states and is is written $\vdash P$.

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

The negation and conjunction of assertions are defined:

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

The disjunction, $\vee$, and implication, $\Rightarrow$, operators have their definition in terms of the negation and conjunction.

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

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

Substitution in assertions is equivalent to updating the state.

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

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

To distinguish the assertions of $\mathcal{A}$ from the logical formulas used in the presentation, universally quantified assertions will be written using lambda notation. For example, $\forall(\lambda v : v +_o 1 >_o v)$ is an assertion of $\mathcal{A}$ which is *true* for any state (for $v \in \mathit{Values}$). The existential quantifier of $\mathcal{A}$, can be defined as $\exists F \stackrel{\text{def}}{=} \neg\forall(\lambda v : \neg F(v))$, where $v \in \mathit{Values}$ and $F \in \mathit{Values} \to \mathcal{A}$.

## Specification of Commands

The *wp* construct defines the weakest precondition necessary for a command to terminate and establish a postcondition. For assertions $P$, $Q$, command $c$ and states $s$ and $t$, $\mathbf{wp}(c, Q)$ is the assertion satisfying:

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

The weakest precondition required for command $c$ to establish postcondition $Q$ is calculated from the weakest precondition required by each command $c_1$ occurring in $c$.

**Definition 7.2** *Weakest precondition*

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

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

□

The rules for the weakest precondition are given in Figure (6). The assignment rule (tl1) is similar to that of Cartwright and Oppen (1981) and requires that the assignment list of the command be correct. The label rule (tl2) requires that a labelled command be 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 proof 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 (6) can be applied. The rule for sequential composition of Dijkstra (1976) can be obtained as an instance of Theorem (6.1):

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

This rule increases the difficulty of a proof, requiring the commands $c_1$ and $c_2$ to be considered individually. The direct approach, reasoning about the single command $c_1; c_2$, leads to a simpler proof.

### Specification of Programs

A program $p$ is specified by a assertion made up of a precondition $P$ and a postcondition $Q$. The specification of $p$ is written $[P]p[Q]$ and is an assertion of $\mathcal{A}$ stating that execution of program $p$ beginning in a state satisfying $P$ will eventually produce a state satisfying $Q$.

**Definition 7.3** *Program specifications*

For assertions $P, Q \in \mathcal{A}$ and program $p \in \mathcal{P}$, a triple $[P]p[Q]$ is an assertion on a state.

$$\begin{aligned}
&[_]_[_] : (\mathcal{A} \times \mathcal{P} \times \mathcal{A}) \to \mathcal{A} \\
&[P]p[Q] \stackrel{\text{def}}{=} \lambda(s : State) : P(s) \Rightarrow \exists(t : State) : \mathcal{I}_p(p)(s, t) \wedge Q(t)
\end{aligned}$$

□

The proof rules for programs are given in Figure (7) and are similar to rules defined by Francez (1992) for the intermittent assertions. Rule (tl6) describes the effect of a command of a program: if $P \Rightarrow wp(c, Q)$ then $c$ terminates 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 strong induction on the natural numbers.

## 8 Example: Division of Natural Numbers

As an example of verification and abstraction in the language $\mathcal{L}$, an object code program for Alpha AXP will be shown to establish a specification. The object code program is produced by compiling a program in the language C (Kernighan & Ritchie, 1978) and is then translated to a program of $\mathcal{L}$. An abstraction of the $\mathcal{L}$ program is then shown to be correct with respect to a specification.

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

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

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

Figure 6: Proof rules for the weakest precondition

Programs: $$\frac{c \in p \quad \vdash p \Rightarrow \mathbf{wp}(c, Q)}{\vdash [P]p[Q]} \quad \text{(tl6)}$$

Refinement: $$\frac{p \sqsubseteq p' \quad \vdash [P]p[Q]}{\vdash [P]p'[Q]} \quad \text{(tl7)}$$

Transitivity: $$\frac{\vdash [P]p[R] \quad \vdash [R]p[Q]}{\vdash [P]p[Q]} \quad \text{(tl8)}$$

Induction: $$\frac{i, j, n \in \mathbb{N} \quad \dfrac{j < i \quad \vdash [F(j)]p[Q]}{\vdash [F(i)]p[Q]}}{\vdash [F(n)]p[Q]} \quad \text{(tl9)}$$

where $c, c_1 \in \mathcal{C}$, $p \in \mathcal{P}$, $b \in \mathcal{E}_b$, $P, Q, R \in \mathcal{A}$ and $F \in (\mathbb{N} \rightarrow \mathcal{A})$

Figure 7: Proof rules for the programs

```
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 8: Division: C Program

The C program, given in Figure (8), implements the division of natural numbers. With arguments $n$ and $d$, the result of the function is the quotient, $n/d$. As a side-effect, the remainder, $n$ mod $d$, is stored in the memory location identified by argument $r$. The program was compiled to produce the object code program for the Alpha AXP processor of Figure (9). Note that although the Alpha AXP is a 64 bit processor, the compiler used to produce the object program represents integers as long-words and all data operations of the program are on long-words.

The object program begins at the instruction labelled *div*, with argument $n$ stored in register **r16**, argument $d$ stored in register **r17** and argument $r$ in register **r18**. The address to which control 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 $c$. The least significant long-words of registers **r16** and **r17** are copied to registers **r1** and **r2** respectively and **r17** is copied to register **r3**. The values of **r1** and **r2** are compared, if **r1** is less than **r2** ($n < d$), control passes to the instruction labelled $35 otherwise control passes to the instruction labelled $36. This begins the loop implementing the *while* statement of the C program. Register **r0** is incremented by 1 ($c = c + 1$); **r16** is decremented by the value of **r17** ($n = n - d$) and compared with **r3** (which is equal to **r17**). If **r16** is not less than **r3** ($n < d$), control passes to the instruction labelled $36, beginning another iteration of the loop. If **r16** is greater than or equal to **r3**, control passes to the instruction labelled $35. This stores the value of **r16** in the memory location identified by **r18**, implementing the assignment $*r = n$. This terminates the program passing control to the instruction identified by register **r26**. The result of the program (the C variable $c$) is stored in register **r0**.

## 8.1 $\mathcal{L}$ program

The object program of Figure (9) is modelled in the language $\mathcal{L}$ by replacing each instruction with its semantics, defined as a command of $\mathcal{L}$. The resulting $\mathcal{L}$ program, *div*, is given in Figure (10). Each of the commands of the $\mathcal{L}$ program are labelled and the commands at label $l_1$, $l_7$ and $l_{12}$ correspond to the processor instructions (of Figure 9) labelled *div*, $36 and $35 respectively.

```
div:
        bis $31,$31,$0          ; r0:= r31 OR r31 (equivalent to r0:=0)
        zapnot $16,15,$1        ; r1:= Long(r16)
        zapnot $17,15,$2        ; r2:= Long(r17)
        bis $2,$2,$3            ; r3:=r2 or r2 (equivalent to r3:=r2)
        cmpule $1,$2,$1         ; if r1<r2 then r1:=1 else r1:=0
        bne $1,$35              ; if not r1 = 0 then PC=label(35) else PC=label(36)
$36:
        addl $0,1,$0            ; longword add: r0:=r0+1
        subl $16,$17,$16        ; longword subtract: r16:=r16-r17
        zapnot $16,15,$1        ; r1:= Long(r16)
        cmpule $1,$3,$1         ; if r1<r3 then r1:=1 else r1:=0
        beq $1,$36              ; if r1 = 0 then PC=label(36) else PC=label(35)
$35:
        stl $16,0($18)          ; store longword: mem(r18):=r16
        ret $31,($26),1         ; return from subroutine: PC:=r26
```

Figure 9: Division: Alpha AXP Object Program

$l_1$ :**r0** $:= 0, l_2$
$l_2$ :**r1** $:=$ **Long**(**r16**)$, l_3$
$l_3$ :**r2** $:=$ **Long**(**r17**)$, l_4$
$l_4$ :**r3** $:=$ **r2**$, l_5$
$l_5$ :**if r1** $<_{64}$ **r2 then r1** $:= 1, l_6$ **else r1** $:= 0, l_6$
$l_6$ :**if not r1** $=_{64}$ 0 **then goto** $l_{12}$ **else goto** $l_7$

$l_7$ :**r0** $:=$ **r0** $+_{64} 1, l_8$
$l_8$ :**r16** $:=$ **r16** $-_{64}$ **r17**$, l_9$
$l_9$ :**r1** $:=$ **Long**(**r16**)$, l_{10}$
$l_{10}$ :**if r1** $<_{64}$ **r3 then r1** $:= 1, l_{11}$ **else r1** $:= 0, l_{11}$
$l_{11}$ :**if r1** $= 0$ **then goto** $l_7$ **else goto** $l_{12}$

$l_{12}$ :**mem**(**r18**) $:=$ **Long**(**r16**)$, l_{13}$
$l_{13}$ :**goto r26**

Figure 10: Division: $\mathcal{L}$ program *div*

$$\begin{aligned}
c_1 = l_1 : &\textbf{if } \mathbf{Long}(\mathbf{r16}) <_{64} \mathbf{Long}(\mathbf{r17}) \\
&\textbf{then } \mathbf{r0}, \mathbf{r1}, \mathbf{r2}, \mathbf{r3}, \mathbf{mem}(\mathbf{r18}) := 0, 1, \mathbf{Long}(\mathbf{r17}), \mathbf{Long}(\mathbf{r17}), \mathbf{Long}(\mathbf{r16}), \mathbf{r26} \\
&\textbf{else } \mathbf{r0}, \mathbf{r1}, \mathbf{r2}, \mathbf{r3} := 0, 0, \mathbf{Long}(\mathbf{r17}), \mathbf{Long}(\mathbf{r17}), l_7
\end{aligned}$$

$$\begin{aligned}
c_7 = l_7 : &\textbf{if not } \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3} \\
&\textbf{then } \mathbf{r0}, \mathbf{r1}, \mathbf{r16} := \mathbf{r0} +_{64} 1, 0, \mathbf{r16} -_{64} \mathbf{r17}, l_7 \\
&\textbf{else } \mathbf{r0}, \mathbf{r1}, \mathbf{r16}, \mathbf{mem}(\mathbf{r18}) := \mathbf{r0} +_{64} 1, 1, \mathbf{r16} -_{64} \mathbf{r17}, \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}), \mathbf{r26}
\end{aligned}$$

Figure 11: Program $div_1$: Abstraction of $\mathcal{L}$ program *div*

## 8.2 Specification

The specification of the $\mathcal{L}$ program 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**. In addition, when the program terminates, control must pass to the command at label, $l$, stored in register **r26** and $l$ must not be in the range $\{l_1, \ldots, l_{13}\}$.

For $n, d, a, l \in \mathbb{N}$, the assertion $Pre(n, d, a, l) \in \mathcal{A}$ states the precondition on the arguments to program *div*.

$$\begin{aligned}
Pre(n, d, a, l) \stackrel{\text{def}}{=}\ & d > 0 \wedge n \geq 0 \wedge (l < l_1 \vee l > l_{13}) \\
& \wedge \mathbf{r16} = n \wedge \mathbf{r17} = d \wedge \mathbf{r18} = a \wedge \mathbf{r26} = l
\end{aligned}$$

The postcondition, $Post(n, d, a, l) \in \mathcal{A}$, is the assertion to be established by the program.

$$\begin{aligned}
Post(n, d, a, l) \stackrel{\text{def}}{=}\ & n \geq 0 \wedge d > 0 \wedge \mathbf{r26} = l \wedge \mathbf{r18} = a \\
& \wedge n = (\mathbf{r0} \times_{64} d) +_{64} \mathbf{mem}(\mathbf{r18})
\end{aligned}$$

The specification of the program requires that when the command labelled $l_1$ is selected for execution in a state satisfying $Pre(n, d, a, l)$, then eventually the program *div* establishes $Post(n, d, a, l)$ and control passes to the command labelled $l$. The specification to be satisfied by the program is:

$$\vdash [pc = l_1 \wedge Pre(n, d, a, l)] div [pc = l \wedge Post(n, d, a, l)] \quad (\text{for } n, d, a, l \in \mathbb{N})$$

## 8.3 Abstraction

An abstraction $div_1$ of the program *div* is obtained by combining the commands of *div* using the property of Theorem (6.6). The commands are combined in the order in which they are executed. Because there is a loop, at the command labelled $l_7$, in the program *div*, there is a cut-point at the label $l_7$ (Floyd, 1967). The program is verified using the method of intermittent assertions and the cut-point must be preserved in the abstraction $div_1$. The abstraction $div_1$ is therefore formed from two commands. The first, which will be referred to as $c_1$, is labelled $l_1$, and obtained by the sequential composition of the commands labelled $l_1$ to $l_6$ and the commands labelled $l_{12}$ and $l_{13}$.

$$\begin{aligned}
c_1 = (((((( & at(div, l_1); at(div, l_2)); at(div, l_3)); at(div, l_4)); \\
& at(div, l_5)); at(div, l_6)); at(div, l_{12})); at(div, l_{13}))
\end{aligned}$$

The second command, denoted $c_7$, is obtained by the sequential composition of the commands which form the loop beginning at $l_7$ and the commands executed after the loop terminates.

$$c_7 = ((((((at(div, l_7); at(div, l_8)); at(div, l_9)); at(div, l_{10})); \\ at(div, l_{11})); at(div, l_{12})); at(div, l_{13}))$$

Obvious simplifications can be applied to commands $c_1$ and $c_7$, these result in the commands of Figure (11). For example, the result of $at(div, l_1); at(div, l_2)$ is the command:

$$\begin{array}{l} l_1 : \textbf{if } l_2 = l_2 \\ \quad \textbf{then } := (((pc, l_2) \cdot (\mathbf{r0}, 0)) \oplus ((\mathbf{r1}, \mathbf{Long}(\mathbf{r16})) \triangleleft ((pc, l_2) \cdot (\mathbf{r0}, 0))), l_3) \\ \quad \textbf{else } := ((\mathbf{r0}, 0), l_2) \end{array}$$

Since the registers $pc$, $\mathbf{r0}$, $\mathbf{r1}$ and $\mathbf{r16}$ are distinct in any state and since $l_2 = l_2$ is trivially true, the the command can be replaced with the equivalent but simpler command:

$$l_1 : (\mathbf{r0}, \mathbf{r1} := 0, \mathbf{r16}, l_3)$$

This is a simple application of symbolic execution (King, 1971) and a constant folding transformation (Aho et al., 1986). The abstraction $div_1$ is obtained by combining $c_1$ and $c_2$ with the program $div$, $div_1 = (div \uplus \{c_1, c_2\})$. This gives $div_1 \sqsubseteq div$, by repeated application of Theorem (6.5) and Theorem (6.6). However, only the two commands of Figure (11) are needed to verify the abstraction $div_1$.

## 8.4 Verification

The program $div$ is verified by showing that its abstraction $div_1$ satisfies the specification:

$$\vdash [pc = l_1 \wedge Pre(n, d, a, l)]div_1[pc = l \wedge Post(n, d, a, l)] \quad (\text{for } n, d, a, l \in \mathbb{N})$$

The property of the loop in the program, at command $c_7$, is verified by induction on the value of $\mathbf{r16}$. The invariant for the loop (Floyd, 1967; Burstall, 1974) specifies the values of the quotient and the remainder at each iteration of the loop.

$$Inv(q, n, d, a, l) \stackrel{\text{def}}{=} \begin{cases} n \geq 0 \wedge d > 0 \wedge \neg(l =_{64} l_7) \wedge q =_{64} \mathbf{r16} \\ \wedge l =_{64} \mathbf{r26} \wedge a =_{64} \mathbf{r18} \wedge \mathbf{Long}(d) =_{64} \mathbf{r3} \wedge \mathbf{Long}(d) =_{64} \mathbf{r17} \\ \wedge \mathbf{Long}((\mathbf{r0} \times_{64} \mathbf{r17}) +_{64} \mathbf{r16}) =_{64} \mathbf{Long}(n) \\ \wedge (pc =_{64} l \Rightarrow \mathbf{mem}(\mathbf{r18}) =_{64} \mathbf{Long}(\mathbf{r16})) \end{cases}$$

When the program terminates (with $pc =_{64} \mathbf{r26}$), the loop invariant, $Inv$, satisfies, for $q \in \mathbb{N}$, the postcondition.

$$\vdash pc = \mathbf{r26} \wedge \mathbf{Long}(q) =_{64} (\mathbf{Long}(n) \bmod_o \mathbf{Long}(d)) \wedge Inv(q, n, d, a, l) \Rightarrow Post(n, d, a, l)$$

Given the precondition, the postcondition and the loop invariant, the verification of $idiv_1$ is by 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 $c_7$ is selected for execution. The third to show, by induction, that the command $c_7$ establishes the loop invariant and that the loop terminates.

1. Precondition establishes postcondition:

$$\vdash [pc =_{64} l_1 \wedge \mathit{Pre}(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d)]\mathit{idiv}_1[pc =_{64} l \wedge \mathit{Post}(n,d,a,l)]$$

2. Precondition establishes invariant:

$$\vdash [pc =_{64} l_1 \wedge \mathit{Pre}(n,d,a,l) \wedge \neg\mathbf{Long}(n) <_{64} \mathbf{Long}(d)]\mathit{idiv}_1[pc =_{64} l_7 \wedge \mathit{Inv}(n,n,d,a,l)]$$

3. Invariant establishes postcondition, the proof is by induction on $q$ (rule tl9):

$$\vdash [pc =_{64} l_7 \wedge \mathit{Inv}(q,n,d,a,l)]\mathit{idiv}_1[pc =_{64} l \wedge \mathit{Post}(n,d,a,l)] \quad \text{for any } q \in \mathbb{N}$$

   (a) Base case, $\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3}$:

$$\begin{aligned}&\vdash(pc =_{64} l_7 \wedge \mathit{Inv}(q,n,d,a,l) \wedge \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})\\ &\quad\Rightarrow \mathit{wp}(c_7, pc =_{64} l \wedge \mathit{Post}(n,d,a,l))\end{aligned}$$

   (b) Inductive case, $\neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3}$:

$$\begin{aligned}&\vdash(pc =_{64} l_7 \wedge \mathit{Inv}(q,n,d,a,l) \wedge \neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})\\ &\quad\Rightarrow \mathit{wp}(c_7, pc =_{64} l \wedge \mathit{Post}(n,d,a,l))\end{aligned}$$

The proof of the first of these steps is representative of the way in which a command is shown to establish a property and will be given here. Proofs for the other steps are given in the appendix.

**Step (1):** *Precondition establishes postcondition.*

$$\vdash [pc =_{64} l_1 \wedge \mathit{Pre}(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d)]\mathit{idiv}_1[pc =_{64} l \wedge \mathit{Post}(n,d,a,l)]$$

Since $pc = l_1$, command $c_1$ of program $\mathit{idiv}_1$ is selected and, by rule (tl6), the assertion to prove is that the assumptions satisfy $\mathit{wp}(c_1, pc =_{64} l \wedge \mathit{Post}(n,d,a,l))$. From rule (tl3), the fact that $c_1$ is a conditional command and the assumptions $\mathbf{Long}(n) <_{64} \mathbf{Long}(d)$ and $\mathbf{r16} = \mathbf{Long}(n) \wedge \mathbf{r17} = \mathbf{Long}(d)$ (definition of *Pre*), it follows that the precondition must satisfy the assertion:

$$\begin{aligned}\vdash\quad & (pc =_{64} l_1 \wedge \mathit{Pre}(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d))\\ &\Rightarrow \mathit{wp}((\mathbf{r0},\mathbf{r1},\mathbf{r2},\mathbf{r3},\mathbf{mem}(\mathbf{r18}) := 0, 1, \mathbf{Long}(\mathbf{r17}), \mathbf{Long}(\mathbf{r17}), \mathbf{Long}(\mathbf{r16}), \mathbf{r26}),\\ &\qquad\quad pc =_{64} l \wedge \mathit{Post}(n,d,a,l))\end{aligned}$$

The assumptions can be weakened (rule tl5) with the postcondition updated with the assignments of $c_1$. This requires:

$$\begin{aligned}\vdash\quad & (pc =_{64} l_1 \wedge \mathit{Pre}(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d))\\ &\Rightarrow\\ &(pc =_{64} l \wedge \mathit{Post}(n,d,a,l)) \triangleleft (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{Long}(\mathbf{r17}))\\ &\qquad\qquad \cdot (\mathbf{r3}, \mathbf{Long}(\mathbf{r17})) \cdot (\mathbf{mem}(\mathbf{r18}), \mathbf{Long}(\mathbf{r16}))\end{aligned} \tag{4}$$

That this is *true* can be shown from the definitions and by substitution:

$$
\begin{array}{lll}
\vdash & pc =_{64} l_1 \wedge Pre(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d) & \text{(assumptions)} \\
\vdash & Pre(n,d,a,l) & \\
& \Rightarrow & \\
& n \geq 0 \wedge d > 0 \wedge \mathbf{r26} = l \wedge \mathbf{r18} = a & \\
& \wedge \mathbf{r16} = n \wedge \mathbf{r17} = n & \text{(definition } Pre\text{)} \\
\vdash & Pre(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d) & \\
& \Rightarrow n = (0 \times_{64} d) +_{64} n & (\mathbf{Long}(n) <_{64} \mathbf{Long}(d)) \\
\vdash & Pre(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d) & \\
& \Rightarrow n = (0 \times_{64} d) +_{64} \mathbf{r16} & (\mathbf{r16} = n) \\
\vdash & Pre(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d) & \\
& \Rightarrow & \\
& (n = (\mathbf{r0} \times_{64} d) +_{64} \mathbf{mem}(\mathbf{r18})) & \text{(substitution)} \\
& \quad \triangleleft (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{Long}(\mathbf{r17})) & \\
& \quad \cdot (\mathbf{r3}, \mathbf{Long}(\mathbf{r17})) \cdot (\mathbf{mem}(\mathbf{r18}), \mathbf{Long}(\mathbf{r16})) & \\
\vdash & Pre(n,d,a,l) & \\
& \Rightarrow & \\
& (pc =_{64} l) \triangleleft (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{Long}(\mathbf{r17})) & \text{(substitution and } l = \mathbf{r26}\text{)} \\
& \quad \cdot (\mathbf{r3}, \mathbf{Long}(\mathbf{r17})) \cdot (\mathbf{mem}(\mathbf{r18}), \mathbf{Long}(\mathbf{r16})) & \\
\vdash & Pre(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} \mathbf{Long}(d) & \\
& \Rightarrow & \\
& (pc =_{64} l \wedge Post(n,d,a,l)) \triangleleft (pc, \mathbf{r26}) \cdot (\mathbf{r0}, 0) & \text{(definition of } Post\text{)} \\
& \quad \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r2}, \mathbf{Long}(\mathbf{r17})) & \\
& \quad \cdot (\mathbf{r3}, \mathbf{Long}(\mathbf{r17})) \cdot (\mathbf{mem}(\mathbf{r18}), \mathbf{Long}(\mathbf{r16})) &
\end{array}
$$

By rule (tl1), Assertion (4) establishes the weakest precondition of the assignment and it follows that the precondition establishes the postcondition in the case when $\mathbf{r16} <_{64} \mathbf{r17}$.

### 8.4.1 Proof of the Program

The proof of correctness of program *idiv*$_1$ is by combining the assertions established by the two commands $c_1$ and $c_7$. For any $n, d, a, l \in \mathbb{N}$, when $n < d$ the proof is straightforward; at command $c_1$, Step (1) establishes:

$$\vdash [pc =_{64} l_1 \wedge Pre(n,d,a,l) \wedge \mathbf{Long}(n) <_{64} d] idiv_1 [pc =_{64} l \wedge Post(n,d,a,l)]$$

When $n > d$ the proof is by the transitivity rule (tl8), the Steps (2) and (3) establish

$$\vdash [pc =_{64} l_1 \wedge Pre(n,d,a,l) \wedge \neg \mathbf{Long}(n) <_{64} d] idiv_1 [pc =_{64} l_7 \wedge Inv(n,d,a,l)]$$
$$\vdash [pc =_{64} l_7 \wedge Inv(n,d,a,l)] idiv_1 [pc =_{64} l \wedge Post(n,d,a,l)]$$

The correctness of *idiv*$_1$ is follows, by cases of $\mathbf{Long}(n) <_{64} d$ and transitivity (rule tl8), establishing:

$$\vdash [pc =_{64} l_1 \wedge Pre(n,d,a,l)] idiv_1 [pc =_{64} l \wedge Post(n,d,a,l)]$$

By the refinement rule (tl7), this also establishes the correctness of *idiv*:

$$\vdash [pc =_{64} l \wedge Pre(n,d,a,l)] idiv [pc =_{64} l \wedge Post(n,d,a,l)]$$

Note that verifying the abstraction $div_1$ rather than program $div$, reduces the size of the proof. For example, to show that the precondition established the invariant, it was only necessary to consider one command and two cases, for the test of the conditional command. To prove the same property for the program $div$ would require the six commands, labelled $l_1, \ldots, l_6$, and four cases, two cases for each of the tests of the conditional commands at $l_5$ and $l_6$, to be considered.

# 9 Conclusion

Verification has been limited by the difficulty of reasoning about programs in the presence of pointers and computed jumps and by the work required to show that a program is correct. This means that for a large class of programs, which includes the object code which is executed on a machine, verification is not a practical prospect. These problems are solved for programs of the language $\mathcal{L}$ by generalising the operations needed for verification and by a method for constructing abstractions of a program. The operations required were substitution, for verification, and to merge assignment lists, for abstraction. Instead of the usual definition as functions on syntactic terms, these are defined as expressions of $\mathcal{L}$. This allows the operations to have a syntactic form, which can be manipulated, as well as a semantics, to calculate the result of the operations. The operations are used to simplify the verification of a program by constructing abstractions of the program. These abstractions are also programs of $\mathcal{L}$, allowing the use of a single program logic to verify both programs and their abstractions. Since both verification and abstraction are based on the use of the program text, efficient proof tools can be constructed and used to further simplify program verification.

An alternative approach to program verification uses the semantics of a language to construct an interpreter for programs of the language (Boyer & Moore, 1997). This approach solves the problems caused by pointers and computed jumps and allows the verification of object code programs (Yuan Yu, 1992). Because of the complexity of a proof of correctness based on an interpreter, the semantic approach to verification requires the use of automated proof tools to reason about the behaviour of the interpreter. The complexity of an interpreter for a practical language and the resources needed for the proof tool means that there are practical limits on the size of the program which can be verified. The interpreter also limits the simplifications which can be performed on the program, since the language may not be expressive enough to describe the result of simplifying a program.

The syntactic approach, used here, allows a program to be verified by reasoning about the properties to be established by the program. This leads to simpler proofs since only the properties which are needed must be considered. The syntactic approach is also more suitable for a mixture of manual and automated reasoning. The ability to verify the program in a system of logic means that the proof of correctness is constructed in a human-readable form. Automated tools which assist in a manually directed proof can be efficiently implemented, since only a mechanical manipulation of text is required. Because of the expressiveness of the language $\mathcal{L}$, proof tools and methods developed for $\mathcal{L}$ can be applied to a wide range of programs. This, together with the ability to describe object code as a program of $\mathcal{L}$, provides a practical means for verifying the programs of a range of processor languages.

The work described in this paper has been verified using the PVS theorem prover (Owre et al., 1993). The verified theory includes the definitions, theorems, lemmas, the substitution rules (Figure 3) and the proof rules for commands and programs (Figures 6 and 7). It does not include the examples.

## References


Aho, A., Sethi, R. and Ullman, J. (1986). *Compilers. Principles, Techniques and Tools*. Addison-Wesley.

Arbib, M. A. and Alagić, S. (1970). Proof rules for gotos. *Acta Informatica 11*, 139–148.

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.

Barringer, H., Cheng, J. H. and Jones, C. B. (1984). A logic covering undefinedness in program proofs. *Acta Informatica 21*(3), 251 – 269.

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. (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.

Burstall, R. M. (1974). Program proving as hand simulation with a little induction. In *Information Processing*, Volume 74, pp. 308–312. North-Holland.

Cartwright, R. and Oppen, D. (1981). The logic of aliasing. *Acta Informatica 15*, 365 – 384.

Clint, M. and Hoare, C. A. R. (1972). Program proving: Jumps and functions. *Acta Informatica 1*, 214–224.

Cousot, P. (1981). Semantic foundations of program analysis. In *Program Flow Analysis*, Chapter 10, pp. 303–342. Prentice-Hall.

Cousot, P. and Cousot, R. (1993). “A la Burstall” intermittent assertion induction principles for proving inevitability properties of programs. *Theoretical Computer Science 120*, 123–155.

de Bruin, A. (1981). Goto statements: Semantics and deduction systems. *Acta Informatica 15*, 384 – 424.

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.

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.

Francez, N. (1992). *Program Verification*. Addison Wesley.

Gordon, M. J. C. (1994). State transition assertions: A case study. In Bowen, J. (Ed.), *Towards Verified Systems*, Volume 2 of *Real-Time Safety Critical Systems*, Chapter 5, pp. 93–113. Elsevier Science.

Gries, D. (1981). *The Science of Programming*. Springer-Verlag.

Hayes, J. P. (1988). *Computer Architecture and Organization* (Second ed.). McGraw-Hill.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM 12*, 576–580.

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.

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. C. (1971, November). Proving programs to be correct. *IEEE Transactions on Computers C-20*(11), 1331–1336.

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. (1981a). Deductive synthesis of the unification algorithm. *Science of Computer Programming 1*, 5 – 48.

Manna, Z. and Waldinger, R. (1981b). Problematic features of programming languages: A situational-calculus approach. *Acta Informatica 16*, 371 – 426.

Morgan, C. (1990). *Programming from Specifications*. Prentice-Hall International.

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.

Sites, R. L. (1992). Alpha AXP architecture. *Digital Technical Journal 4*(4), 1–17.

Yuan Yu (1992). *Automated Proofs of Object Code for a Widely Used Microprocessor*. Ph. D. thesis, University of Texas at Austin.

# A Proof of Program *idiv*

The verification of the abstraction $\mathit{idiv}_1$ is in three steps of which the first is given in the main text of the paper. The remaining steps are as follows:

**Step (2):** *Precondition establishes invariant.*

$$\vdash [\mathit{pc} =_{64} l_1 \wedge \mathit{Pre}(n,d,a,l) \wedge \neg\mathbf{Long}(n) <_{64} \mathbf{Long}(d)]\mathit{idiv}_1[\mathit{pc} =_{64} l_7 \wedge \mathit{Inv}(n,n,d,a,l)]$$

As in the first step, command $c_1$ is selected for execution. Since the condition $\mathbf{Long}(n) <_{64} \mathbf{Long}(d)$ is false, the assertion to prove is:

$$\begin{array}{ll}\vdash & pc =_{64} l_1 \wedge Pre(n,d,a,l) \wedge \neg\mathbf{Long}(n) <_{64} \mathbf{Long}(d) \\ & \Rightarrow \\ & wp((\mathbf{r0},\mathbf{r1},\mathbf{r2},\mathbf{r3}, := 0, 0, \mathbf{Long}(\mathbf{r17}), \mathbf{Long}(\mathbf{r17}), l_7), pc =_{64} l_7 \wedge Inv(n,n,d,a,l))\end{array}$$

From the weakening rule (tl5), this can be reduced to the assertion:

$$\begin{array}{l}\vdash pc =_{64} l_1 \wedge Pre(n,d,a,l) \wedge \neg\mathbf{Long}(n) <_{64} \mathbf{Long}(d) \\ \quad \Rightarrow (pc =_{64} l_7 \wedge Inv(n,n,d,a,l)) \triangleleft (pc, l_7) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 0) \cdot (\mathbf{r2}, \mathbf{r17}) \cdot (\mathbf{r3}, \mathbf{r17})\end{array} \tag{5}$$

and, for the assignment rule (tl1), to the assertion:

$$\begin{array}{l}\vdash (pc =_{64} l_7 \wedge \wedge Inv(n,n,d,a,l)) \triangleleft (pc, l_7) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 0) \cdot (\mathbf{r2}, \mathbf{r17}) \cdot (\mathbf{r3}, \mathbf{r17}) \\ \quad \Rightarrow (pc =_{64} l_7 \wedge Inv(n,n,d,a,l))\end{array} \tag{6}$$

Both are straightforward from the definitions of *Inv* and *Pre* and by substitution. Note that result of the substitution in *Inv* can be determined from the syntax of the expressions:

$$\begin{array}{ll}\multicolumn{2}{l}{(pc =_{64} l_7 \wedge Inv(n,n,d,a,l)) \triangleleft (pc, l_7) \cdot (\mathbf{r0}, 0) \cdot (\mathbf{r1}, 0) \cdot (\mathbf{r2}, \mathbf{r17}) \cdot (\mathbf{r3}, \mathbf{r17})} \\ \equiv & (l_7 =_{64} l_7 \wedge n \geq 0 \wedge d > 0 \wedge \neg(l =_{64} l_7) \wedge n =_{64} \mathbf{r16} \\ & \wedge l =_{64} \mathbf{r26} \wedge a = \mathbf{r18} \wedge \mathbf{Long}(d) =_{64} \mathbf{r17} \wedge \mathbf{Long}(d) =_{64} \mathbf{r17} \\ & \wedge \mathbf{Long}((0 \times_{64} \mathbf{r17}) +_{64} \mathbf{r16}) =_{64} \mathbf{Long}(n) \\ & \wedge (l_7 =_{64} l \Rightarrow \mathbf{mem}(\mathbf{r18}) =_{64} \mathbf{Long}(\mathbf{r16})))\end{array}$$

The remainder of the proof for this case is straightforward by arithmetic and from the proof rules.

**Step (3):** *Invariant establishes postcondition.*

$$\vdash [pc =_{64} l_7 \wedge Inv(q,n,d,a,l)] idiv_1 [pc =_{64} l \wedge Post(n,d,a,l)] \quad \text{for any } q \in \mathbb{N}$$

When the value of the program counter is $l_7$, the command selected is $c_7$ which forms a loop in the program. The proof is therefore by induction on the quotient, $q$, using rule (tl9). By rule (tl6) the assertion to be established is:

$$\vdash (pc =_{64} l_7 \wedge Inv(q,n,d,a,l)) \Rightarrow wp(c_7, pc =_{64} l \wedge Post(n,d,a,l))$$

The inductive hypothesis (rule tl9) states that, for any $j < q$ (where $j \in \mathbb{N}$):

$$\vdash (pc =_{64} l_7 \wedge Inv(j,n,d,a,l)) \Rightarrow wp(c_7, pc =_{64} l \wedge Post(n,d,a,l))$$

The proof is for the base case, $q - d < d$, and for the inductive case, $q - d \geq d$.

**Step (3a):** *Invariant establishes postcondition (base case* $\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3}$*).*

$$\begin{array}{l}\vdash (pc =_{64} l_7 \wedge Inv(q,n,d,a,l) \wedge \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3}) \\ \quad \Rightarrow wp(c_7, pc =_{64} l \wedge Post(n,d,a,l))\end{array}$$

From the assumption $\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3}$ and rule (tl3), the assertion to establish is:

$$\begin{array}{l}\vdash (pc =_{64} l_7 \wedge Inv(q,n,d,a,l) \wedge \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3}) \\ \quad \Rightarrow \\ \quad wp(\mathbf{r0}, \mathbf{r1}, \mathbf{r16}, \mathbf{mem}(\mathbf{r18}) := \\ \qquad \mathbf{r0} +_{64} 1, 1, \mathbf{r16} -_{64} \mathbf{r17}, \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}), \mathbf{r26}, \\ \qquad pc =_{64} l \wedge Post(n,d,a,l))\end{array}$$

The proof for this is by application of the assignment rule (tl1), with the assertion:

$$\vdash (pc =_{64} l \wedge Post(n,d,a,l)) \triangleleft (pc, \mathbf{r26}) \cdot (\mathbf{r0}, \mathbf{r0} +_{64} 1) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r16}, \mathbf{r16} -_{64} \mathbf{r17}) \cdot (\mathbf{mem}(\mathbf{r18}, \mathbf{Long}(\mathbf{r16} - \mathbf{r17}))) \Rightarrow (pc =_{64} l \wedge Post(n,d,a,l)) \tag{7}$$

and by the weakening rule (tl5), with the assertion:

$$\begin{aligned}&\vdash (pc =_{64} l_7 \wedge Inv(q,n,d,a,l) \wedge \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})\\ &\Rightarrow\\ &(pc =_{64} l \wedge Post(n,d,a,l)) \triangleleft (pc, \mathbf{r26}) \cdot (\mathbf{r0}, \mathbf{r0} +_{64} 1) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r16}, \mathbf{r16} -_{64} \mathbf{r17})\\ &\qquad \cdot (\mathbf{mem}(\mathbf{r18}, \mathbf{Long}(\mathbf{r16} - \mathbf{r17})))\end{aligned} \tag{8}$$

For both assertion (7) and assertion (8), the result of the substitution into *Post* can be determined from the syntax of the expressions.

$$\begin{aligned}&(pc =_{64} l \wedge Post(n,d,a,l)) \triangleleft (pc, \mathbf{r26}) \cdot (\mathbf{r0}, \mathbf{r0} +_{64} 1) \cdot (\mathbf{r1}, 1) \cdot (\mathbf{r16}, \mathbf{r16} -_{64} \mathbf{r17})\\ &\qquad \cdot (\mathbf{mem}(\mathbf{r18}, \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17})))\\ &\equiv\\ &(\mathbf{r26} =_{64} l \wedge n \geq 0 \wedge d > 0 \wedge \mathbf{r26} =_{64} l \wedge \mathbf{r18} = a\\ &\wedge n =_{64} (((\mathbf{r0} +_{64} 1) \times_{64} d) +_{64} \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17})))\end{aligned}$$

The remainder of the proof for this case is straightforward from arithmetic.

**Step (3b):** *Invariant establishes postcondition (inductive case* $\neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3}$*).*

$$\begin{aligned}&\vdash (pc =_{64} l_7 \wedge Inv(q,n,d,a,l) \wedge \neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})\\ &\quad \Rightarrow wp(c_7, pc =_{64} l \wedge Post(n,d,a,l))\end{aligned}$$

In the case when $\neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{Long}(\mathbf{r17})) <_{64} \mathbf{r3}$, the proof is by showing that the assumptions of the inductive hypothesis are established by command $c_7$. The postcondition to be established is $pc =_{64} l_7 \wedge Inv(q-d,n,d,a,l)$: the remainder **r16** eventually decreases. From the conditional rule (tl3) and the assumption $\neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17})$, the proof is of:

$$\begin{aligned}&\vdash (pc =_{64} l_7 \wedge Inv(q,n,d,a,l) \wedge \neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})\\ &\quad \Rightarrow\\ &\quad wp((\mathbf{r0}, \mathbf{r1}, \mathbf{r16} := \mathbf{r0} +_{64} 1, 0, \mathbf{r16} -_{64} \mathbf{r17}, l_7),\\ &\qquad pc =_{64} l_7 \wedge Inv(q-d,n,d,a,l))\end{aligned}$$

As before, the proof is by the assignment rule (tl1) and the weakening rule (tl5). For the weakening rule, the assertion required is:

$$\begin{aligned}&\vdash (pc =_{64} l_7 \wedge Inv(q,n,d,a,l) \wedge \neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})\\ &\Rightarrow\\ &(pc =_{64} l_7 \wedge Inv(q-d,n,d,a,l)) \triangleleft (pc, l_7) \cdot (\mathbf{r0}, \mathbf{r0} +_{64} 1) \cdot (\mathbf{r1}, 0) \cdot (\mathbf{r16}, \mathbf{r16} -_{64} \mathbf{r17})\end{aligned} \tag{9}$$

For the assignment rule, the assertion required is:

$$\begin{aligned}&\vdash (pc =_{64} l_7 \wedge Inv(q-d,n,d,a,l)) \triangleleft (pc, l_7) \cdot (\mathbf{r0}, \mathbf{r0} +_{64} 1) \cdot (\mathbf{r1}, 0) \cdot (\mathbf{r16}, \mathbf{r16} -_{64} \mathbf{r17})\\ &\Rightarrow (pc =_{64} l_7 \wedge Inv(q-d,n,d,a,l) \wedge \neg\mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})\end{aligned} \tag{10}$$

For both note that the result of the substitution is the equivalence:

$$
\begin{aligned}
&(pc =_{64} l_7 \land \mathit{Inv}((q-d, n, d, a, l)) \triangleleft (pc, l_7) \cdot (\mathbf{r0}, \mathbf{r0} +_{64} 1) \cdot (\mathbf{r1}, 0) \cdot (\mathbf{r16}, \mathbf{r16} -_{64} \mathbf{r17})\\
&\equiv\\
&l_7 =_{64} l_7 \land n \geq 0 \land d > 0 \neg (l =_{64} l_7) \land \mathbf{Long}(q) =_{64} \mathbf{r0}\\
&\land l =_{64} \mathbf{r26} \land a =_{64} \mathbf{r18} \land \mathbf{Long}(d) =_{64} \mathbf{r3} \land \mathbf{Long}(d) =_{64} \mathbf{r17}\\
&\land \mathbf{Long}(((\mathbf{r0} +_{64} 1) \times_{64} \mathbf{r17}) +_{64} (\mathbf{r16} -_{64} \mathbf{r17})) =_{64} \mathbf{Long}(n)\\
&\land (l_7 =_{64} l \land \mathbf{mem}(\mathbf{r18}) =_{64} (\mathbf{r16} - \mathbf{r17}))
\end{aligned}
$$

The proof of the assertions is straightforward by arithmetic. This establishes the assumptions of the inductive hypothesis:

$$
\begin{aligned}
\vdash &[(pc =_{64} l_7 \land \mathit{Inv}(q, n, d, a, l) \land \neg \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})]\\
&\mathit{idiv}_1\\
&[pc =_{64} l_7 \land \mathit{Inv}(q-d, n, d, a, l)]
\end{aligned}
$$

From the inductive hypothesis, and transitivity (rule tl8), it follows that the postcondition of the program is established.

$$
\begin{aligned}
\vdash &[(pc =_{64} l_7 \land \mathit{Inv}(q, n, d, a, l) \land \neg \mathbf{Long}(\mathbf{r16} -_{64} \mathbf{r17}) <_{64} \mathbf{r3})]\\
&\mathit{idiv}_1\\
&[pc =_{64} l \land \mathit{Post}(n, d, a, l)]
\end{aligned}
$$

Completing the proof for this assertion and the steps needed to prove the program.