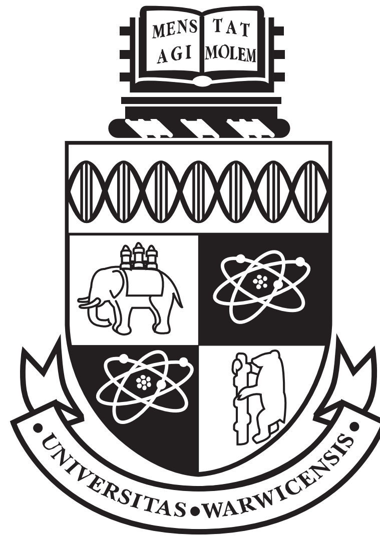


Compositional Software Verification Based on Game Semantics

by

Aleksandar Dimovski

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

July 2007

THE UNIVERSITY OF
WARWICK

Contents

List of Tables vi

List of Figures viii

Acknowledgments x

Declarations xii

Abstract xiii

Chapter 1 Introduction 1

1.1 Software Verification 2

1.2 Model Checking 3

1.3 Game Semantics Approach 6

1.3.1 The Idea 8

1.3.2 Advantages 10

1.3.3 Challenges 12

1.3.4 Methodological Considerations 13

1.4 Contributions 14

Chapter 2 The Programming Language 16

2.1 Syntax 17
2.2 Operational Semantics 22
2.3 Observational Safety 23

Chapter 3 Game Semantics **27**

3.1 Background: Category Theory 28
3.2 Translating AIA into EIAA 31
3.3 Games and Strategies 36
3.3.1 Games 36
3.3.2 Strategies 43
3.3.3 A Cartesian Closed Category 47
3.4 The Model 50
3.5 Soundness 55
3.6 Definability 59
3.7 Full Abstraction 62
3.8 Quotient Game Semantics 65

Chapter 4 The CSP Game Semantics Model **67**

4.1 The Second-Order Language Fragment 69
4.1.1 Syntactic Sugar 71
4.2 Regular-language Representation 72
4.3 CSP Representation 78
4.3.1 Background: CSP 79
4.3.2 Representation 84
4.3.3 Correctness and Property Verification 91
4.3.4 Type Inference System 97
4.3.5 Compiler and Applications 99

Chapter 5 Abstraction Refinement **108**

5.1 Interaction Game Semantics 111

5.2 Conservativity of Abstraction 114

5.3 Abstraction Refinement 117

5.4 Implementation 121

5.4.1 Representing Game Models in CSP 122

5.4.2 Implementing Abstraction Refinement Procedure . . . 123

5.4.3 Using the Tool 128

5.4.4 Examples 128

Chapter 6 Compositional Verification **134**

6.1 The Learning Algorithm 137

6.2 Compositional Verification 141

6.2.1 Overview 141

6.2.2 Assume-Guarantee Algorithm 145

6.2.3 Example 147

6.3 Implementation 150

Chapter 7 Conclusions **154**

Appendix A CSP Scripts for Case Studies **157**

A.1 Functions 157

A.2 Stack Implementation 163

List of Tables

2.1 Typing rules of AIA 20
2.2 Reduction rules for AIA (part 1) 24
2.3 Reduction rules for AIA (part 2) 25

3.1 Translating AIA into EIAA 34

4.1 Regular-language representation 75
4.2 Representation of language constructs 76
4.3 Copy-cat processes for free identifiers 85
4.4 Processes for expressions 86
4.5 Processes for op construct 87
4.6 An implementation of the equality operation 88
4.7 An implementation of the addition operation 89
4.8 Processes for commands 90
4.9 Processes for command constructs 91
4.10 Processes for variables 92
4.11 Processes for variable constructs 92
4.12 Processes for functionals 93
4.13 Type inference system 98

4.14 Model generation of IA_2 bubble sort 103
4.15 Checking safety for an erroneous IA_2 bubble sort 104

5.1 Experimental results for checking a stack implementation . . . 133

6.1 Experimental results for checking a stack implementation . . . 152

List of Figures

2.1 Graphical representation of some integer abstractions 18

3.1 Strategies 66

4.1 A strategy as a finite automaton 78

4.2 A labelled transition system 83

4.3 A strategy as a labelled transition system 97

4.4 Source code of AIA_2 bubble sort 101

4.5 LTS for AIA_2 bubble sort with $k = 2$ 101

4.6 LTS for IA_2 bubble sort with $k = 2$ 102

4.7 Effects of compressions for IA_2 bubble sort with $k = 20$ 104

4.8 A stack implementation 105

4.9 LTS for the stack with $k = 2$ 106

5.1 Verification procedure 109

5.2 A possible definition of $\sqsubseteq$ 118

5.3 Abstraction refinement procedure 120

5.4 A screen-shot of the tool 129

5.5 The tool architecture 130

6.1 The L^* algorithm for learning assumptions 140
6.2 Compositional verification procedure 145
6.3 The strategy for the running example 148
6.4 Strategies at AR iteration 0: (a) $[\![f \vdash M[-]]\!](\alpha)$ (b) $[\![f, x \vdash f(x := x + 1)]\!]$ 149
6.5 Observation table and assumption at AR iteration 0 150
6.6 Strategies at AR iteration 1: (a) $[\![f \vdash M[-]]\!](\alpha)$ (b) $[\![f, x \vdash f(x := x + 1)]\!]$ 151
6.7 Assumption at AR iteration 1 151

Acknowledgments

First of all, I would like to thank my supervisor, Ranko Lazić, for being so supportive all throughout my stay at Warwick University, for showing me the right way to approach research, and for helping me many times to make the right choices.

I would also like to thank my advisor, Marcin Jurjinski, for keeping an eye on my progress, for reading my articles with great care and providing many useful suggestions.

I thank my co-author, Dan Ghica, for listening to all my ideas and for deftly steering me towards solutions.

I have met many people during the course of my PhD. I thank everyone with whom I have had interactions, both in research and otherwise. I would particularly like to thank the following people: Chien-An Chen, Sarah Lim Choi Keung, Misra Kundan, Paul Isitt, Justin Dyson, Nick Papanikolaou, Ashutosh Trivedi, Ritesh Krishna, Hammad Qureshi, Rajagopal Nagarajan, David Lacey, Sara Kavala, Jane Sinclair, etc.

I thank for the financial support I have received from the Engineering and Physical Sciences Research Council (EPSRC) and the Overseas Research Students (ORS) Award.

Finally, I am very grateful to my parents Conka and Simeon Dimovski,

my sister Daniela, my grandparents Milica and Petar Zafirovski and Cvetanka and Ivan Vasilovi, for their support, encouragement and love. Special thanks go to Petar Dimov Zafirovski.

Declarations

I hereby declare that this thesis has not been previously submitted, either in the same or different form, to this or any other university for a degree.

Abstract

One of the major challenges in computer science is to put programming on a firmer mathematical basis, in order to improve the correctness of computer programs. Automatic program verification is acknowledged to be a very hard problem, but current work is reaching the point where at least the foundational aspects of the problem can be addressed and it is becoming a part of industrial software development.

This thesis presents a semantic framework for verifying safety properties of open sequential programs. The presentation is focused on an Algol-like programming language that embodies many of the core ingredients of imperative and functional languages and incorporates data abstraction in its syntax. Game semantics is used to obtain a compositional, incremental way of generating accurate models of programs. Model-checking is made possible by giving certain kinds of concrete automata-theoretic representations of the model. A data-abstraction refinement procedure is developed for model-checking safety properties of programs with infinite integer types. The procedure starts by model-checking the most abstract version of the program. If no counterexample, or a genuine one, is found, the procedure terminates. Otherwise, it uses a spurious counterexample to refine the abstraction for the next iteration. Abstraction refinement, assume-guarantee reasoning and the L^* algorithm for learning regular languages are combined to yield a procedure for compositional verification. Construction of a global model is avoided using assume-guarantee reasoning and the L^* algorithm, by learning assumptions for arbitrary subprograms. An implementation based on the FDR model checker for the CSP process algebra demonstrates practicality of the methods.

Chapter 1

Introduction

Software verification is one of the most important problems in computer science today. There is hardly any aspect of our lives where software systems do not play an often silent but yet crucial role. Failure of these systems has already caused serious consequences, including fatal accidents, shutdown of vital systems, and loss of money. Thus, erroneous software becomes a threat for the economy and even for human lives. The increasing dependence on software systems has ensured that their correctness is no longer a luxury but an urgent necessity.

Modern software systems are rarely monolithic entities that are single-handedly developed at one time. A number of groups of programmers, sometimes located in different places, work on developing different parts of programs (components). Software systems also evolve over time, with different components reaching maturity at different points. Assuring that all components will successfully work together is a nontrivial task. It is thus not surprising that verifying components independently, i.e. ensuring that every component performs correctly under all circumstances, has become crucial.

1.1 Software Verification

Software verification addresses the problem of checking that programs satisfy certain properties. There are two main classes of program properties of interest: *safety* and *liveness*. The safety properties demand that the program never performs an undesirable operation. For example, it never divides by zero. The liveness properties demand that the program eventually performs desirable operations. For example, it eventually terminates. In general, both problems are undecidable but, in the past decades, significant advances have been made by developing methods which show that verification problems are becoming increasingly feasible.

To improve software correctness, testing has traditionally been the main debugging technique in industry. Testing [88] is the process of sampling the executions of a system according to some criterion, and checking the given property for each execution. However, exhaustive testing is usually infeasible as the number of possible executions is too large (or even infinite). Thus, testing can be used to find errors, but it can not be used to show correctness of a software program.

In order to overcome the problem mentioned above, the scientific community has proposed the use of *formal methods*. This term covers all verification approaches based on mathematical formalisms. Their aim is to establish software correctness with mathematical rigour. As opposed to testing, formal verification methods trace every possible program execution as they work on a symbolic and abstract level. Thus, when a program is found to be correct by a formal verification method, it implies that all its executions have been explored, and the question of missed executions becomes irrelevant. Most ap-

proaches to formal software verification can be classified as belonging to two major categories: deductive verification and model checking.

In deductive verification [45, 69], the property to be established is expressed as a formula ϕ in some suitable logic. The meaning of elementary programming-language constructs is expressed by axioms, and that of larger constructs by inference rules in some proof system in the same logic as ϕ. Denoting the program to be verified by P, its correctness is shown by constructing a proof that $P \vdash \phi$ within this system. This is done using a theorem prover. Deductive verification is a comprehensive approach for establishing correctness, which can be used to verify programs with infinitely many states and with data from infinite domains (such as integers and reals). However, the main limitations are that it is highly time consuming and involves a lot of manual effort. Furthermore, it yields no diagnostic feedback that can be used for debugging if the property is found not to be correct.

1.2 Model Checking

In model checking [94, 28, 32], the program to be verified is represented by a model M. The model consists of a description of all possible program executions (behaviours) in a mathematical structure like a finite state transition system. The property to be established is a formula ϕ in a logic that is interpreted over such structures (e.g. temporal logic). Program correctness is then shown by computing that the formula is satisfied by the model, i.e. $M \models \phi$. This check is performed by exhaustively exploring the entire state space of the model to ensure that all possible behaviours generated indeed satisfy the property.

Compared to other approaches, model checking has two important advantages. It is fully automatic, and so its application requires no user supervision or expertise in logic and theorem proving. When the model fails to satisfy a desired property, it provides useful diagnostic feedback in the form of counterexamples, which trace some example program executions that violate the property of interest. Owing to these and other factors, the past couple of decades have witnessed the emergence of model checking as the eminent formal verification technique. Starting with relatively small finite state systems, by developing techniques like symbolic model checking [83], bounded model checking [20], compositional reasoning [29], abstraction [30] and others, it was made possible to verify systems with enormous state spaces.

The initial success of model checking has been mainly in the verification of hardware and communication protocols. The major reason for this is that model checking can only be used if a finite model of the system to be verified is available, and it is computationally demanding. While the modelling process is often straightforward for hardware, since hardware designs are typically finite state, it is much more involved for software. This is due to the complexity of general purpose programming languages (C, Java, ML, etc) as compared to hardware description languages (VHDL, Verilog, etc). Also, industrial software programs are large and have infinite state spaces. Thus, extracting a finite model often involves a process of abstraction as well.

The *traditional approach* to building models of software is based on operational semantics. The notion of a *program state* is central to this approach. The state captures the values of the program variables at a certain moment in the execution of the program. The models are then obtained by representing the state and the way it changes in the course of execution. By applying pred-

icate abstraction [58] on the state, i.e. by using truth assignments for a set of chosen predicates to abstractly represent the set of states where the truth assignments are satisfied, the models become finite and can be model checked.

This modelling technique has been applied successfully to verifying realistic industrial software. At the heart of many such tools, like SLAM [17], BLAST [66] and Magic [24], are algorithms based on *counterexample guided abstraction refinement* [31]. In this approach, the entire verification procedure is captured by the following three step loop:

Abstract A finite set of predicates is chosen, and a finite-state abstract model is extracted from the given program using predicate abstraction. Since abstractions are conservative over-approximations, additional behaviours, which are absent in the concrete program, are introduced in the abstract model (such behaviours are called spurious).

Verify A model checker is used to verify whether the abstract model satisfies the desired property. If the model is error free, then so is the original program; otherwise a counterexample is produced which demonstrates how the model violates the property.

Refine It is checked whether the counterexample is an actual behaviour of the original program. If so, then a program error has been found; otherwise, the chosen set of predicates does not contain enough information for proving program correctness and new predicates must be added. The selection of such predicates is guided by the failure to concretise all previous spurious counterexamples. The whole procedure is then repeated.

1.3 Game Semantics Approach

Denotational semantics [61, 101] is a syntax-independent approach of modelling a software program of a given language as a mathematical object in a fully compositional manner. The intention is that mathematical methods for reasoning about the model can be employed to understand and explain how programs behave. In particular, this approach can be used to deduce properties of programs, such as that two programs are equivalent, or that a program satisfies its specification. It is generally concerned with static properties, such as what a given program computes as opposed to exactly how the actual computations are performed.

There are two desirable features of such a denotational model: *soundness* and *completeness*. A model is *sound* iff all equivalences in the model are reflected in the language, so it can be used to prove properties of programs. A model is *complete* iff all equalities in the language are reflected in the model, so every observable program property is captured by the model. A model which is both sound and complete is called *fully abstract*.

The search for a syntax-independent fully abstract model of a very simple sequential functional language, PCF [92, 85], started in 1970's by the work of Scott and Strachey on a domain-theoretic model based on continuous functions [99, 92]. It was followed by Berry's bidomains model based on stable functions [18], the Bucciarelli-Ehrand model based on strongly stable functions [23], and the Berry-Curien model based on sequential algorithms [19]. In each case, the model is sound but fails to capture *definability*: there are elements of the model which are not definable in the language. Therefore, there are some equalities in the language which are not validated by the model [85],

i.e. the model is not complete. The solution of the long-standing full abstraction problem for PCF emerged in the past decade, when the *game-theoretical* model was developed independently by Abramsky, Jagadessan, and Malacaria on one hand [1], and by Hyland and Ong [74] (as well as by Nickau [90]) on the other hand. The two teams presented each a model of PCF in which types are interpreted by games and programs by strategies. The two models are now commonly referred to as the AJM and the HO model. Since then, game semantics has been employed to construct the first syntax-independent fully abstract models for a range of programming languages incorporating many other features such as block-allocated variables [3, 7], call-by-value evaluation [4, 72], control primitives [76], general references [6], recursive types [82], polymorphism [9], non-deterministic [63, 64] and probabilistic constructs [35], concurrency [52], etc.

Game theory was founded in the beginning of the past century with works by Zermelo [104] and von Neumann [100] on parlour games. Nash [89] then created a theory of economics based on parlour games. The game-theoretical methods were also used in logic [21], in models of reactive systems [84], natural language semantics [68], etc. The use of game theory in the semantics of programming languages is based on Lorenzen game models of logic [44, 47, 77]. There, a logical formula is interpreted by a two player game between a Player trying to prove the formula and an Opponent trying to disprove it. This is done inductively on the structure of the given formula. The Curry-Howard isomorphism is then used to translate the Lorenzen game models of logic to the game models of programming languages, such that formulas are considered as types and proofs for a formula A as programs of type A.

1.3.1 The Idea

Game semantics is a particular kind of denotational semantics which constructs models of programs by looking at the ways in which a program can observably *interact* with its context (environment). In this approach, a kind of game is played by two participants. The first, Player, represents the program under consideration, while the second, Opponent, represents the environment in which the program is used. The two take turns to make moves, each of which is either a question (a demand for information) or an answer (a supply of information). Opponent always plays first. What these moves are, and when they can be played, is determined by the rules of each particular game. For example, in the game for integers, Opponent has a single move, the question "What is the number?", and Player can then respond by playing a number.

The game involved in modelling a function of type $\mathbb{Z} \to \mathbb{Z}$ is formed from "two copies of the game for $\mathbb{Z}$", one for input, and one for output. In the output copy, Opponent can demand output and Player can provide it. In the input copy, the situation is reversed, Player demands input and Opponent provides it. A play in this game when Player is playing the *successor* function might look like this:

Opponent	"What is the output?"
Player	"What is the input?"
Opponent	"The input is 5"
Player	"The output is 6"

So, the successor function becomes a strategy for Player: "When Opponent asks for output, Player responds by asking for input; when Opponent provides input n, Player supplies $n + 1$ as output". This is the key idea in game

semantics. Types are interpreted as *games*, and programs are interpreted as *strategies* for Player to respond to the moves Opponent can make.

The idea of using game semantics to explore the space of programming languages is based on the following considerations [5]. PCF is modelled by a category of games and highly restricted strategies which correspond to the discipline of purely functional programming. These *restrictions* are determinism, innocence, visibility, and bracketing. The relaxation of one of these restrictions on strategies leads to a larger category which can be used to model an extension of PCF by some non-functional features, such as state or control. For example, relaxing the restriction of innocence allows local state (block-allocated variables) to be modelled, relaxing determinism allows nondeterministic constructs to be modelled, relaxing visibility allows general references to be modelled, while relaxing bracketing allows control to be modelled. Moreover, definability for the model of the extended language can be reduced to definability in the restricted language by using a technique of *factorization theorems*. That is, every strategy in a larger category can be factored as the composition of a restricted strategy and a "generic" unrestricted strategy. Thus the original definability result for PCF can be transferred to a much richer class of languages.

Game semantics integrates denotational and operational semantics, retaining good structural properties of denotational semantics while capturing aspects of operational semantics. This is similar to the program of the geometry of interaction [57] carried out by Girard in the framework of linear logic [56], the work on interaction categories [2] giving rise to type systems for concurrency, and the Brookes model based on infinite stuttering sequences [22] providing full abstraction result for parallel programming languages.

In addition to being used to construct accurate semantics for a variety of programming languages, game models have also been used in program verification. The first steps in this direction were taken by Hankin and Malacaria who have applied the game models to program analysis, i.e. for data flow analysis [79, 80, 62] and for certifying secure information flows in programs [81]. The first application to model checking was proposed by Ghica and McCusker [51]; they show that the game models of second-order finitary Idealized Algol (IA) [95] can be represented in a remarkably simple form by regular languages. Subsequently, a tool based on these ideas was implemented in [10]. This thesis is a further investigation into this area. Specifically, we develop a new more efficient verification tool where game models are given concrete representation using the CSP process algebra [70, 98], and we propose novel algorithms for compositional modelling and verification of safety properties of open IA programs which can contain infinite integer data types.

Another interesting direction of research is using game-theoretical ideas in compositional design. Interface models [36, 37] can support interface compatibility checking and interface refinement checking [26], and therefore compositional design. Many aspects of interface models, such as compatibility and refinement checking between interfaces, are properly viewed in a game-theoretical setting, where the input and output values of an interface are chosen by different players.

1.3.2 Advantages

Several features of game semantics make it very promising for software model checking. Compared with the traditional state-based approach, game semantics based model checking has the following important advantages:

Modularity There is a model for any open program with free (undefined) identifiers in a high-level language with procedures, local variables and data types. Since modern software programs are not monolithic entities, this approach enables building models of software components and compositional reasoning about their properties.

Correctness The generated model is correct (sound and complete), and it is set on a firm theoretical foundation. Thus, two programs have the same models if and only if they can not be distinguished with respect to operational tests (such as abnormal termination) in any program context. For example, all sorting algorithms have the same models in this setting. Moreover, the model can be adapted relatively easily for a quantitative analysis of programming languages, such as comparing programs for efficiency [54].

Compositionality Models are constructed inductively on the structure of programs, i.e. the model of a program is constructed from the models of its subprograms, using a notion of strategy composition. This feature is the key for achieving scalability, i.e. the possibility to break up a larger program into smaller subprograms which can be modelled and verified independently.

Efficiency Programs are modelled by how they observationally interact with their environments, and the details of local-state manipulation are not recorded, i.e. they are abstracted, which results in small models with a maximum level of abstraction. Moreover, since the model construction process is compositional all intermediate models are also observationally abstracted, and local reductions can be applied at every step of composi-

tion. For example, the program $\mathsf{newint}\, x := 0\, \mathsf{in}\, x := !x + 1;\ !x$ [1] and the constant 1 have the same models. The variable x is not represented in the model because it is local and so invisible to the outside world.

1.3.3 Challenges

Model checking tools, such as FDR [46], SPIN [71], NuSMV [27], are complex programs that have been crafted over many years by experts in the specific formalisms employed by the tools. They offer a number of algorithms for property specification and efficient verification. A re-implementation of these algorithms would likely produce inferior performance. For this reason, the focus in software verification tools is on the problem of extracting finite-state models and their concrete representation in formalisms which are supported by powerful model checkers. For example, the Bandera tool [34, 91] extracts finite-state models from Java programs for checking with SPIN and NuSMV. On the other hand, SLAM-like tools [17, 66, 24] use a different strategy: they use a dedicated model checking engine to process a model which is derived on-the-fly from a program. Since in this thesis we focus almost exclusively on the problem of model extraction and concrete representation, we choose the first strategy as more appropriate to our approach.

Challenge 1 Game models of finitary open programs of 2nd-order IA are regular languages. Can we give concrete representation of these models in a convenient formalism which is expressive enough so that the models can be readily encoded and supported by a powerful model checker highly optimised for verification of compositional models?

[1] $!x$ denotes de-referencing

As was mentioned earlier, the current state-based tools implement abstraction refinement algorithms for automatic verification of industrial size programs.

Challenge 2 Can we adapt counterexample guided abstraction refinement ideas to the setting of game models and thus enable verification of open programs with infinite integer data types?

One of the main problems in software model checking is the *state explosion problem*: industrial programs are large and the size of state spaces grows exponentially with the size of the programs, making model checking computationally demanding. Therefore, the tendency is to look for compositional methods, which attempt to verify different parts of a program separately, and then make conclusions about the program as a whole. In compositional verification, properties of the program are decomposed into properties of its components (subprograms), so that if each component satisfies its respective property, then so does the whole program. Compositional verification is thus very desirable.

Challenge 3 Can we utilise the compositional nature of the building of game models to achieve compositional verification?

1.3.4 Methodological Considerations

We now discuss some important methodological assumptions which will be used throughout this thesis. As a main presentation vehicle is considered the metalanguage AIA (Abstracted Idealized Algol). AIA is an expressive sequential programming language which combines the fundamental features of imperative and higher-order functional languages. We develop a framework

for verifying safety properties of open programs of second-order recursion-free fragment of AIA with iteration. Safety properties are specified by a designated unsafe command abort whose executability in a given program will be checked. The game semantics presented in this thesis is known as HO-style game semantics (after Hyland-Ong).

1.4 Contributions

The material contained in this thesis presents a framework for verifying safety properties of sequential open programs of IA with specific emphasis on addressing the challenges mentioned above. Challenge 1 is addressed in Chapter 4 by showing how game models can be represented compositionally in the CSP process algebra. This enables observational safe-equivalence and a range of safety properties of open programs to be checked using the FDR model checker [46]. Experimental results confirm the effectiveness of this approach. Then, in Chapter 5, a data-abstraction refinement procedure is proposed as a solution to Challenge 2. The procedure applies to open programs which can contain infinite integer types and is guaranteed to discover an error if it exists. Finally, a fully compositional verification procedure is presented in Chapter 6 to address Challenge 3. It combines counterexample guided abstraction refinement, assume-guarantee reasoning [75, 93] and the L^* algorithm for learning regular languages [14]. Overall model construction of a program is avoided using assume-guarantee reasoning and the L* algorithm, by learning assumptions and reasoning about arbitrary subprograms.

The thesis is organised in the following way:

Chapter 2 introduces the language considered in this thesis, Abstracted Ide-

alized Algol (AIA) which incorporates data-abstraction in its syntax. The syntax and operational semantics of the language are presented, as well as the notion of observational safety.

Chapter 3 defines a game semantics model of the language we consider and contains a proof of full abstraction for the model.

Chapter 4 shows how game models of an interesting fragment of AIA can be represented by CSP processes. It is then illustrated how, using this translation, several classes of verification problems can be decided.

Chapter 5 presents a data-abstraction refinement procedure for verifying safety properties and a tool which implements this procedure.

Chapter 6 extends the abstraction refinement procedure to allow compositional verification.

Chapter 7 contains a retrospective view of the goals and achievements of the work presented as well as discusses possible extensions.

Origin of the chapters. Although the material contained in this thesis has been published in the form of articles, it has been restructured and extended. Chapter 4 is based on [38, 39, 42], which are joint work with Ranko Lazić. Chapter 3 and Chapter 5 contain material from [40, 41], which are the result of a collaboration with Dan Ghica and Ranko Lazić. Chapter 6 is a revision of [43], which is a joint work with Ranko Lazić.

Chapter 2

The Programming Language

The standard approach in denotational semantics is the utilisation of metalanguages for the description of certain kinds of computational behaviour. The semantic model is defined for a metalanguage, and a real programming language (C, Java, ML, etc.) can then be studied by translating it into this metalanguage and considering the induced model. This approach allows the same metalanguage and the semantic model for it to be used in the study of many real languages. Moreover, the same metalanguage can also be used for building many different semantic models.

In this thesis, the focus is on the metalanguage: Idealized Algol with active expressions (IA for short) introduced by Reynolds in [95]. IA is similar to Core ML [86]. It is a compact language which combines imperative and higher-order functional programming. The basis of IA is a simply-typed call-by-name λ-calculus in which the standard constructs of imperative programming and locally-scoped variables can be represented. Storage allocation in IA obeys a stack discipline (sometimes called block structure), without any form of garbage collection. This means that after execution of a command, the values

of all variables declared within the command have no further effects on the program term. In order to obtain finite semantic models, the main presentation vehicle is a particular variant of IA, called *Abstracted Idealized Algol* (AIA for short). The key feature of this language is the use of abstraction schemes at the level of data types, which allows the writing of (finitely) abstracted programs in a syntax similar to that of concrete programs. In fact, a concrete program is a particular abstracted program, in which all the abstractions are identities.

In this chapter, we first introduce the syntax and operational semantics of AIA, and then define the notion of observational safety of program terms.

2.1 Syntax

The *data types* of AIA are *abstracted integers* and booleans,

$$D ::= \mathsf{int}_\pi \mid \mathsf{bool}$$

The abstractions π range over computable partitionings of the integers $\mathbb{Z}$. Any such partitioning consists of partitions, i.e. sets of integers, which are called abstracted integers. To say that $m, n \in \mathbb{Z}$ are in the same partition of π, we write $m \approx_\pi n$. In particular, we use the following finitary abstractions:

$$[\,] = \{\mathbb{Z}\} \qquad [n,m] = \{<\!n, \{n\}, \{n+1\}, \ldots, \{0\}, \ldots, \{m-1\}, \{m\}, >\!m\}$$

where $<\!n = \{n' \mid n' < n\}$, and $>\!n = \{n' \mid n' > n\}$. Instead of $\{n\}$, we may write just n. A graphical representation of the abstractions is shown in Figure 2.1.

The *base types* are then

$$B ::= \mathsf{exp}D \mid \mathsf{var}D \mid \mathsf{com}$$

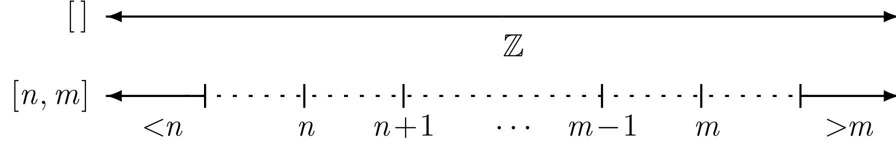


Figure 2.1: Graphical representation of some integer abstractions

where $\mathsf{exp}D$ is the type of expressions which yield values in D, $\mathsf{var}D$ is the type of assignable variables at which values of D can be stored, and com is the type of commands.

The *general phrase types* of AIA are defined by

$$T ::= B \mid T \to T$$

We say that a type is *concrete* if it contains no abstractions other than the identity abstraction $\kappa = \big\{\{i\} \mid i \in \mathbb{Z}\big\}$. For any type T, we write $\widetilde{T}$ for the concrete type obtained by replacing all abstractions with κ. For simplicity, we write int_κ as simply int.

The *terms* of the language are inductively defined as follows:

$$\begin{aligned} M ::= {} & x \mid n_\pi \mid b \mid v \mid \lambda\, x : T.M \mid MM \mid YM \mid M \,\mathsf{op}_D\, M \\ & \mathsf{skip} \mid \mathsf{abort} \mid M\,;\; M \mid \mathsf{if}_B\, M \,\mathsf{then}\, M \,\mathsf{else}\, M \mid \mathsf{while}\, M \,\mathsf{do}\, M \\ & M \;:=\; M \mid !M \mid \mathsf{new}_D\, x := v \,\mathsf{in}\, M \mid \mathsf{mkvar}_D\, MM \end{aligned}$$

where x ranges over a countable set of variables, n_π over abstracted integers int_π, b over booleans and v over constants of type D. The standard functional constructs are defined for forming and applying functions as well as recursion ($\lambda\, x : T.M$, MM, and YM, respectively). The usual arithmetic-logic

operations (op_D) are employed. The command constants are the "do nothing" command which always terminates successfully (skip), and a command which causes abnormal termination (abort). The imperative constructs are: sequencing (;), conditional (if_B), iteration (while), assignment (:=), and de-referencing (!). Block-allocated local variables are introduced by $\mathsf{new}_D\ x := v\ \mathsf{in}\ M$, where the local variable x is initialised to v and it is bound in M. Variables are considered as pairs of two objects: an acceptor (or write method) of type $\mathsf{exp}D \rightarrow \mathsf{com}$, which is a function from expressions to commands, i.e. it takes a value as input and executes a command to store that value in the variable, and a de-referencing operation (or read method) of type $\mathsf{exp}D$, which returns the current value stored in the variable. The constructor mkvar_D then takes an acceptor M and a de-referencing operation N and creates a variable $\mathsf{mkvar}_D\ MN$.

Well-typed terms are given by typing judgements of the form $\Gamma \vdash M : T$. Here Γ is a type *context* consisting of a finite number of typed free identifiers, i.e. of the form

$$x_1 : T_1, \ldots, x_k : T_k$$

where all identifiers x_i are distinct. In the rest of this thesis, typing judgements of the form $\Gamma \vdash M : T$ are referred to as *terms*, and identifiers $x_i : T_i$ in a context Γ are referred to as *free identifiers*. The typing rules of the language are defined in Table 2.1.

Note that op_D stands for any arithmetic-logic operator whose concrete type is $\mathsf{exp}\widetilde{D_1} \times \mathsf{exp}\widetilde{D_2} \rightarrow \mathsf{exp}\widetilde{D}$. For example, for any abstractions π_1 and π_2, AIA contains an equality operator $=_{\mathsf{bool}}$ of type $\mathsf{expint}_{\pi_1} \times \mathsf{expint}_{\pi_2} \rightarrow \mathsf{expbool}$.

For the cases of arithmetic-logic operations, conditional, assignment, new_D and mkvar_D, it is required that their corresponding types of subterms

$$x_1 : T_1, \ldots, x_k : T_k \vdash x_i : T_i,\ i \in \{1, \ldots, k\}$$

$$\Gamma \vdash n_\pi : \mathsf{expint}_\pi \qquad \Gamma \vdash b : \mathsf{expbool},\ b \in \{tt, f\!f\}$$

$$\frac{\Gamma, x : T \vdash M : T'}{\Gamma \vdash \lambda\, x : T.M : T \to T'} \qquad \frac{\Gamma \vdash M : T \to T' \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : T'}$$

$$\frac{\Gamma \vdash M : T \to T}{\Gamma \vdash YM : T}$$

$$\frac{\Gamma \vdash M : \mathsf{exp}D_1 \qquad \Gamma \vdash N : \mathsf{exp}D_2}{\Gamma \vdash M\ \mathsf{op}_D\ N : \mathsf{exp}D}\ \widetilde{D_1} = \widetilde{D_2}, \mathsf{op} \in \{+, -, <, \ldots\}$$

$$\Gamma \vdash \mathsf{skip} : \mathsf{com} \qquad \Gamma \vdash \mathsf{abort} : \mathsf{com}$$

$$\frac{\Gamma \vdash M : \mathsf{com} \qquad \Gamma \vdash N : B}{\Gamma \vdash M\,;\ N : B}\ B \in \{\mathsf{exp}D, \mathsf{com}\}$$

$$\frac{\Gamma \vdash M : \mathsf{expbool} \qquad \Gamma \vdash N_{tt} : B_1 \qquad \Gamma \vdash N_{f\!f} : B_2}{\Gamma \vdash \mathsf{if}_B\ M\ \mathsf{then}\ N_{tt}\ \mathsf{else}\ N_{f\!f} : B}\ \widetilde{B_1} = \widetilde{B_2} = \widetilde{B}$$

$$\frac{\Gamma \vdash M : \mathsf{expbool} \qquad \Gamma \vdash N : \mathsf{com}}{\Gamma \vdash \mathsf{while}\ M\ \mathsf{do}\ N : \mathsf{com}}$$

$$\frac{\Gamma \vdash M : \mathsf{var}D_1 \qquad \Gamma \vdash N : \mathsf{exp}D_2}{\Gamma \vdash M := N : \mathsf{com}}\ \widetilde{D_1} = \widetilde{D_2} \qquad \frac{\Gamma \vdash M : \mathsf{var}D}{\Gamma \vdash !M : \mathsf{exp}D}$$

$$\frac{\Gamma, x : \mathsf{var}D \vdash M : B \qquad \Gamma \vdash v : \mathsf{exp}D_1}{\Gamma \vdash \mathsf{new}_D\ x := v\ \mathsf{in}\ M : B}\ \widetilde{D} = \widetilde{D_1}, B \in \{\mathsf{exp}D', \mathsf{com}\}$$

$$\frac{\Gamma \vdash M : \mathsf{exp}D_1 \to \mathsf{com} \qquad \Gamma \vdash N : \mathsf{exp}D_2}{\Gamma \vdash \mathsf{mkvar}_D\ MN : \mathsf{var}D}\ \widetilde{D_1} = \widetilde{D_2} = \widetilde{D}$$

Table 2.1: Typing rules of AIA

have equal concretisations, but their abstractions can be different. For example, for any abstractions π_1 and π_2, we can assign expressions of type int_{π_2} to variables of type int_{π_1}. This flexibility enables abstractions within a term to be changed independently of each other while preserving well-typed-ness.

Whenever a term $\Gamma \vdash M : T$ is derivable using the typing rules above, there is a unique such derivation and the type T is unique with respect to Γ and M. For example,

$$x : \mathsf{varint}_{[0,4]} \vdash x := !x +_{[0,3]} 1_{[0,1]} : \mathsf{com}$$

means that the operator $+$ was used with type $\mathsf{expint}_{[0,4]} \times \mathsf{expint}_{[0,1]} \to \mathsf{expint}_{[0,3]}$. Although the syntax of terms is presented with the type annotations, we may omit them for succinctness if they are clear from the context or irrelevant.

Compared with Reynolds' original definition of Idealized Algol [95], in the language presented here, commands can be sequenced not only with commands but also with expressions. This means that evaluation of an expression can update a variable, i.e. *active expressions* or expressions with side effects such as

$$x : \mathsf{varint} \vdash x :=!x + 1;\ !x : \mathsf{expint}$$

are allowed. So, active expressions can perform assignments to non-local variables in Γ. This is a common feature of most real imperative languages.

A term is *concrete* if it contains no abstractions other than the identity abstraction κ. For any term $\Gamma \vdash M : T$, we write $\widetilde{\Gamma} \vdash \widetilde{M} : \widetilde{T}$ for the concrete term obtained by replacing all abstractions with κ. An abstraction π is *finitary* if it has finitely many partitions. A term is *finitely abstracted* if it contains only finitary abstractions. Finally, we say that a term M of type T is *closed* if $\vdash M : T$ is derivable.

2.2 Operational Semantics

Operational semantics is a clear and convenient way to specify a programming language, so it is common to use it as a benchmark by which to measure a denotational semantics. There are two standard ways of defining operational semantics: the *small-step* or structural operational semantics, and the *big-step* or natural semantics. We present here the latter.

We proceed by defining a notion of *state*. Given a context $\Gamma = x_1 : \mathsf{var} D_1, \ldots, x_k : \mathsf{var} D_k$ where all free identifiers are variables, which is called var-*context*, we define a Γ-*state* s as a function assigning data values to the variables $\{x_1, \ldots, x_k\}$. Given a Γ-state s, we write $(\mathrm{s} \mid x \mapsto v)$ for the state identical to s but that variable x is mapped to v. We use $(\mathrm{s} \mid x \mapsto v)$ both to update a Γ-state and to extend a Γ-state to a $\{\Gamma, x\}$-state where x is initialised to v, depending on whether x occurs in Γ or not.

The canonical forms (values) are defined by:

$$V ::= x \mid n_\pi \mid b \mid \lambda\, x : T.M \mid \mathsf{skip} \mid \mathsf{mkvar}_D MN$$

The operational semantics is now defined by a big-step reduction relation

$$\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{K}$$

where $\Gamma \vdash M : T$ is a term, Γ is a var-context, s is a Γ-state, and $\mathcal{K}$ is a final configuration. The final configuration can be either a pair V, s' with V a canonical form and s' a Γ-state, or a special error configuration $\mathcal{E}$.

The reduction rules are presented in Tables 2.2 and 2.3. We extend the $\approx_\pi$ notation to data types as follows. Let $\approx_{\mathsf{bool}}$ be the identity (i.e. equality) relation, and $\approx_{\mathsf{int}_\pi}$ mean the same as $\approx_\pi$. The notation $M[N/x]$ denotes the

capture-free substitution of term N for all free occurrences of x in the term M. Note that the λ-abstraction is the only binder in our language.

The reduction rules for AIA have the following characteristics. Firstly, whenever an integer n needs to be treated as belonging to a data type int_π, n is altered nondeterministically to any n' such that $n \approx_\pi n'$. Note that integers are nondeterministically altered to other integers in the same partition only in places in the syntax of terms where, instead of equal types, there are types which have equal concretisations. Secondly, the abort program with any state reduces to $\mathcal{E}$, and a composite program can reduce to $\mathcal{E}$ if a subprogram is reduced to $\mathcal{E}$.

If M is a closed term then we abbreviate the relation $\vdash M, \emptyset \Longrightarrow V, \emptyset$ with $M \Longrightarrow V$.

2.3 Observational Safety

Since we are interested in verifying safety properties, we want to define the notion of equivalence of programs in terms of an *observational safety*. The property of programs that should be observable in order to define this equivalence, is the *termination* (convergence) of a program. But in AIA, it is possible that a term evaluates to more than one final configuration. For example, the constant $1_{[0,0]}$ may evaluate to $1, 2, 3, \ldots$, i.e. to any integer n such that $n \approx_{[0,0]} 1$. Thus, if $\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{K}$ then we can only say that M in state s *may* evaluate to final configuration $\mathcal{K}$. This means that we will be interested in defining may-termination equivalence. Note that in this setting, two programs are considered equivalent if they can produce the same range of output values. However, this notion of equivalence gives no account of the possibility

$$\Gamma \vdash n_\pi, \mathrm{s} \Longrightarrow n', \mathrm{s} \;\; (n \approx_\pi n')$$

$$\Gamma \vdash V, \mathrm{s} \Longrightarrow V, \mathrm{s} \;\; (\text{if } V \neq n_\pi) \qquad \Gamma \vdash \mathsf{abort}, \mathrm{s} \Longrightarrow \mathcal{E}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash MN, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \lambda\, x : T.M', \mathrm{s}' \qquad \Gamma \vdash M'[N/x], \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash MN, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \lambda\, x : T.M', \mathrm{s}' \qquad \Gamma \vdash M'[N/x], \mathrm{s}' \Longrightarrow V, \mathrm{s}''}{\Gamma \vdash MN, \mathrm{s} \Longrightarrow V, \mathrm{s}''}$$

$$\frac{\Gamma \vdash M(YM), \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash YM, \mathrm{s} \Longrightarrow \mathcal{E}} \qquad \frac{\Gamma \vdash M(YM), \mathrm{s} \Longrightarrow V, \mathrm{s}'}{\Gamma \vdash YM, \mathrm{s} \Longrightarrow V, \mathrm{s}'}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash M \;\mathsf{op}_D\; N, \mathrm{s} \Longrightarrow \mathcal{E}} \qquad \frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow v_1, \mathrm{s}' \qquad N, \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash M \;\mathsf{op}_D\; N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow v_1, \mathrm{s}' \qquad \Gamma \vdash N, \mathrm{s}' \Longrightarrow v_2, \mathrm{s}''}{\Gamma \vdash M \;\mathsf{op}_D\; N, \mathrm{s} \Longrightarrow v, \mathrm{s}''} \; v_1 \;\mathsf{op}\; v_2 \approx_D v$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash M\,;\; N, \mathrm{s} \Longrightarrow \mathcal{E}} \qquad \frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}' \qquad \Gamma \vdash N, \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash M\,;\; N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}' \qquad \Gamma \vdash N, \mathrm{s}' \Longrightarrow V, \mathrm{s}''}{\Gamma \vdash M\,;\; N, \mathrm{s} \Longrightarrow V, \mathrm{s}''}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash \mathsf{if}_B\; M \;\mathsf{then}\; N_{tt} \;\mathsf{else}\; N_{f\!f}, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathbf{b}, \mathrm{s}' \qquad \Gamma \vdash N_{\mathbf{b}}, \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash \mathsf{if}_B\; M \;\mathsf{then}\; N_{tt} \;\mathsf{else}\; N_{f\!f}, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathbf{b}, \mathrm{s}' \qquad \Gamma \vdash N_{\mathbf{b}}, \mathrm{s}' \Longrightarrow v, \mathrm{s}''}{\Gamma \vdash \mathsf{if}_{\mathsf{exp}D}\; M \;\mathsf{then}\; N_{tt} \;\mathsf{else}\; N_{f\!f}, \mathrm{s} \Longrightarrow v', \mathrm{s}''} \; v \approx_D v'$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathbf{b}, \mathrm{s}' \qquad \Gamma \vdash N_{\mathbf{b}}, \mathrm{s}' \Longrightarrow \mathsf{skip}, \mathrm{s}''}{\Gamma \vdash \mathsf{if}_{\mathsf{com}}\; M \;\mathsf{then}\; N_{tt} \;\mathsf{else}\; N_{f\!f}, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}''}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathbf{b}, \mathrm{s}' \qquad \Gamma \vdash N_{\mathbf{b}}, \mathrm{s}' \Longrightarrow V, \mathrm{s}''}{\Gamma \vdash \mathsf{if}_{\mathsf{var}D}\; M \;\mathsf{then}\; N_{tt} \;\mathsf{else}\; N_{f\!f}, \mathrm{s} \Longrightarrow \mathsf{mkvar}_D(\lambda\, x : \mathsf{exp}D.\, V := x)(!\,V), \mathrm{s}''}$$

Table 2.2: Reduction rules for AIA (part 1)

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash \mathsf{while}\, M \,\mathsf{do}\, N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow tt, \mathrm{s}' \qquad \Gamma \vdash N \,;\, \mathsf{while}\, M \,\mathsf{do}\, N, \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash \mathsf{while}\, M \,\mathsf{do}\, N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow tt, \mathrm{s}' \qquad \Gamma \vdash N \,;\, \mathsf{while}\, M \,\mathsf{do}\, N, \mathrm{s}' \Longrightarrow \mathsf{skip}, \mathrm{s}''}{\Gamma \vdash \mathsf{while}\, M \,\mathsf{do}\, N, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}''}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow f\!f, \mathrm{s}'}{\Gamma \vdash \mathsf{while}\, M \,\mathsf{do}\, N, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'}$$

$$\frac{\Gamma \vdash N, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash M := N, \mathrm{s} \Longrightarrow \mathcal{E}} \qquad \frac{\Gamma \vdash N, \mathrm{s} \Longrightarrow v, \mathrm{s}' \qquad \Gamma \vdash M, \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash M := N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash N, \mathrm{s} \Longrightarrow v, \mathrm{s}' \qquad \Gamma \vdash M, \mathrm{s}' \Longrightarrow x, \mathrm{s}''}{\Gamma \vdash M := N, \mathrm{s} \Longrightarrow \mathsf{skip}, (\mathrm{s}'' \mid x \mapsto v')}\; v \approx_{D_1} v'$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash !\, M, \mathrm{s} \Longrightarrow \mathcal{E}} \qquad \frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow x, \mathrm{s}'}{\Gamma \vdash !\, M, \mathrm{s} \Longrightarrow v, \mathrm{s}'}\; \mathrm{s}'(x) = v$$

$$\frac{\Gamma \vdash M, (\mathrm{s} \mid x \mapsto v') \Longrightarrow \mathcal{E}}{\Gamma \vdash \mathsf{new}_D\, x := v \,\mathsf{in}\, M, \mathrm{s} \Longrightarrow \mathcal{E}}\; v \approx_D v'$$

$$\frac{\Gamma \vdash M, (\mathrm{s} \mid x \mapsto v') \Longrightarrow V, (\mathrm{s}' \mid x \mapsto v'')}{\Gamma \vdash \mathsf{new}_D\, x := v \,\mathsf{in}\, M, \mathrm{s} \Longrightarrow V, \mathrm{s}'}\; v \approx_D v'$$

$$\frac{\Gamma \vdash N, \mathrm{s} \Longrightarrow v, \mathrm{s}' \;\; \Gamma \vdash M, \mathrm{s}' \Longrightarrow \mathsf{mkvar}_D\, LL', \mathrm{s}'' \;\; \Gamma \vdash Lv', \mathrm{s}'' \Longrightarrow \mathcal{E}}{\Gamma \vdash M := N, \mathrm{s} \Longrightarrow \mathcal{E}}\; v \approx_D v'$$

$$\frac{\Gamma \vdash N, \mathrm{s} \Longrightarrow v, \mathrm{s}' \;\; \Gamma \vdash M, \mathrm{s}' \Longrightarrow \mathsf{mkvar}_D\, LL', \mathrm{s}'' \;\; Lv', \mathrm{s}'' \Longrightarrow \mathsf{skip}, \mathrm{s}'''}{\Gamma \vdash M := N, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'''}\; v \approx_D v'$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathsf{mkvar}_D\, NN', \mathrm{s}' \qquad \Gamma \vdash N', \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash !\, M, \mathrm{s} \Longrightarrow \mathcal{E}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathsf{mkvar}_D\, NN', \mathrm{s}' \qquad \Gamma \vdash N', \mathrm{s}' \Longrightarrow v, \mathrm{s}''}{\Gamma \vdash !\, M, \mathrm{s} \Longrightarrow v', \mathrm{s}''}\; v \approx_D v'$$

Table 2.3: Reduction rules for AIA (part 2)

of divergence. Therefore, it is sufficient for reasoning about safety properties, but not liveness properties.

Given a term $\Gamma \vdash M : \mathsf{com}$ where Γ is a var-context, we say that M *may terminate* in state s if there exists a configuration $\mathcal{K}$ such that $\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{K}$ where $\mathcal{K} = \mathcal{E}$ or $\mathcal{K} = \mathsf{skip}, \mathrm{s}'$ for some state s'. We say that M is *safe* if and only if it cannot be reduced from any state to $\mathcal{E}$.

Next, we define a *program context* $C[-] : \mathsf{com}$ *with hole* to be a term with (possibly several occurrences of) a hole in it, such that if $\Gamma \vdash M : T$ is a term of the same type as the hole then $C[M]$ is a well-typed closed term of type com, i.e. $\vdash C[M] : \mathsf{com}$. Then, we say that a term $\Gamma \vdash M : T$ is a *safe-approximate* of a term $\Gamma \vdash N : T$, denoted by $\Gamma \vdash M \sqsubseteq_{\sim} N$, if and only if for all program contexts $C[-] : \mathsf{com}$, if $C[M]$ may terminate successfully (resp., abnormally) then $C[N]$ may also terminate successfully (resp., abnormally), i.e. we have

$$\Gamma \vdash M \sqsubseteq_{\sim} N \text{ iff } \forall\, C[-] : \mathsf{com}.$$
$$\text{if } C[M] \Longrightarrow \mathsf{skip} \text{ then } C[N] \Longrightarrow \mathsf{skip} \;\wedge\; \text{if } C[M] \Longrightarrow \mathcal{E} \text{ then } C[N] \Longrightarrow \mathcal{E}$$

If two terms safe-approximate each other they are considered *safe-equivalent*, denoted by $\Gamma \vdash M \cong N$.

A context is safe if it does not include occurrences of the abort command. A term $\Gamma \vdash M : T$ is *safe* if for any safe context $C_{\mathrm{safe}}[-]$, program $\vdash C_{\mathrm{safe}}[M] : \mathsf{com}$ is safe (i.e. it cannot be reduced to $\mathcal{E}$); otherwise the term is *unsafe*.

Chapter 3

Game Semantics

In this chapter we present the game semantics model for AIA. Introductory accounts of game semantics can be found in [74, 5, 8], and the presentation here draws from all of them.

Game semantics is a denotational semantics which models types as *games*, computation as *plays* of a game, and programs as *strategies* for a game. Strategies compose, much like CSP-style processes, which makes it possible to define denotational models. Games and strategies form a cartesian closed category, which represents a (fully abstract) model of AIA.

AIA can be expressed as syntactic sugar for IA with Erratic choice (EIA) [64, 63] and exceptions (IAx) [76]. EIA is IA extended with a simple erratic choice operator (or), which encompasses nondeterminism in the language. Nondeterminism can be explained intuitively in the following way [64]: if, in the course of evaluating some expression, there is a choice between two possible continuations, the program picks one at random to decide which way to go. IAx is an extension of IA with dynamically bound, locally declared exceptions. These languages have been studied separately before, and combining

their key features is a straightforward exercise. For the purposes of this thesis we work with IA enriched with erratic choice operator and restricted global exceptions which can be only raised but they cannot be caught by any handler (EIAA for short).

After a short introductory section containing background information on category theory and in particular cartesian closed categories, we begin in Section 3.2 by describing how AIA can be translated into EIAA. We then proceed by introducing the basic notions of arenas, games and strategies. This leads to construction of a cartesian closed category, i.e. a fully abstract model for EIAA, which is essentially the model of AIA. This model can be used for proving safety of AIA programs.

3.1 Background: Category Theory

This section gives a brief introduction to some background material which will be used throughout this chapter. The notion of category theory, and in particular cartesian closed category, is commonly used in construction of denotational semantic models of programming languages. Here we introduce the most basic definitions of category theory. The standard references for category theory and its applications in denotational semantics are [78, 16].

Definition A *category* $\mathbf{C}$ consists of

- A set of *objects*.
- A set of *arrows* (often called *morphisms*).
- Operations assigning to each arrow f an object $dom(f)$, its *domain*, and an object $cod(f)$, its *codomain*. We write $f : A \to B$ or $A \xrightarrow{f} B$ to

show that $dom(f) = A$ and $cod(f) = B$. We write $\mathbf{C}(A, B)$ for the collection of all arrows with domain A and codomain B.

- A composition operator assigning to each pair of arrows f and g, with $cod(f) = dom(g)$, a *composite* arrow $f \fatsemi g : dom(f) \to cod(g)$ satisfying the following associative law:

$$(f \fatsemi g) \fatsemi h = f \fatsemi (g \fatsemi h) \text{ for any } f : A \to B, g : B \to C, h : C \to D.$$

- For each object A, an *identity* arrow $id_A : A \to A$ satisfying the following identity law:

$$id_A \fatsemi f = f = f \fatsemi id_B \text{ for every } f : A \to B.$$

Definition An object $\mathbf{1}$ is called a *terminal* object in a category $\mathbf{C}$ if, for every object A, there is a unique arrow from A to $\mathbf{1}$.

In categorical terms, an arrow from a terminal object to an object A is called a *constant* of A.

Definition Let $\mathbf{C}$ be a category. A *product* of a pair of objects A and B is an object $A \times B$ and two *projection* arrows $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, such that for any object C and a pair of arrows $f : C \to A$ and $g : C \to B$ there is a unique arrow $\langle f, g \rangle : C \to A \times B$ making the following diagram

$$\begin{array}{ccccc} & & C & & \\ & \swarrow^{f} & \downarrow{\scriptstyle \langle f,g \rangle} & \searrow^{g} & \\ A & \xleftarrow[\pi_1]{} & A \times B & \xrightarrow[\pi_2]{} & B \end{array}$$

commute, i.e. $\langle f, g \rangle \fatsemi \pi_1 = f$ and $\langle f, g \rangle \fatsemi \pi_2 = g$.

If a category **C** has a product for every pair of objects, we say that **C** has (finite) *products*.

In some categories, the collection of arrows from an object A to an object B can be reflected as itself an object B^A of the category.

Definition Let **C** be a category with all products. An *exponential* of a pair of objects A and B is an object B^A and an arrow $eval_{A,B} : (B^A \times A) \to B$, such that for any object C and arrow $g : (C \times A) \to B$ there is a unique arrow $\Lambda(g) : C \to B^A$ making the following diagram

$$\begin{array}{ccc} B^A \times A & \xrightarrow{eval_{A,B}} & B \\ \uparrow {\scriptstyle \Lambda(g) \times id_A} & \nearrow_{g} & \\ C \times A & & \end{array}$$

commute, i.e. $(\Lambda(g) \times id_A) \mathbin{;} eval_{A,B} = g$.

If a category **C** has an exponential for every pair of objects, we say that **C** has *exponentiation*.

Definition A *cartesian closed category* (CCC) is a category with a terminal object, products, and exponentiation.

A *partial order* is a set D equipped with a binary relation $\leq$ on D that is reflexive, antisymmetric, and transitive, i.e. for all x, y, and z in D, we have:

- $x \leq x$ (reflexivity).
- if $x \leq y$ and $y \leq x$ then $x = y$ (antisymmetry).
- if $x \leq y$ and $y \leq z$ then $x \leq z$ (transitivity).

An *upper bound* of $X \subseteq D$ is an element $x \in D$, such that $y \leq x$ for each $y \in X$. A *least upper bound* of $X \subseteq D$ is an upper bound $\bigsqcup X$ of X, such

that $\bigsqcup X \leq y$ for any other upper bound y of X. A subset $X \subseteq D$ is *directed* if every nonempty finite subset of X has an upper bound. A partial order D is *complete* (or a *cpo*) if D has a least element $\perp$ and every directed subset $X \subseteq D$ has a least upper bound. A function is *monotone* if $x \leq y$ implies $f(x) \leq f(y)$. Given cpo's D and E, a monotone function $f : D \to E$ is said to be *continuous* if $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for any directed X.

Definition A cartesian closed category is *cpo-enriched* if:

- The collection of arrows $\mathbf{C}(A, B)$ between every two objects A and B forms a cpo with respect to an ordering $\leq$.
- Arrow composition is monotone and continuous with respect to the orderings on arrows.

3.2 Translating AIA into EIAA

The data types of EIAA are infinite integers and booleans, and the base types are extended with the empty type ($\mathbf{0}$).

$$
\begin{aligned}
D &::= \mathsf{int} \mid \mathsf{bool} \\
B &::= \mathbf{0} \mid \mathsf{exp}D \mid \mathsf{var}D \mid \mathsf{com} \\
T &::= B \mid T \to T
\end{aligned}
$$

The term constructors of EIAA are those of AIA plus constructors for erratic choice ‘or’ and for raising exceptions ‘raise’:

$$M ::= \ldots \mid M \,\mathsf{or}\, M \mid \mathsf{raise}\, M$$

Note that no type annotations are needed now for integer constants, op, if and mkvar constructs. The typing rules and operational semantics for EIAA are

similar to those of AIA (see Chapter 2) with the differences imposed by using infinite rather than abstracted integers. We have already discussed them in Section 2.2. The typing rules for 'or' and 'raise' are the following:

$$\frac{\Gamma \vdash M : \mathsf{exp}D \qquad \Gamma \vdash N : \mathsf{exp}D}{\Gamma \vdash M \,\mathsf{or}\, N : \mathsf{exp}D} \qquad \frac{\Gamma \vdash M : \mathbf{0}}{\Gamma \vdash \mathsf{raise}\, M : B}$$

The reduction relations are given by:

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{K}}{\Gamma \vdash M \,\mathsf{or}\, N, \mathrm{s} \Longrightarrow \mathcal{K}} \qquad \frac{\Gamma \vdash N, \mathrm{s} \Longrightarrow \mathcal{K}}{\Gamma \vdash M \,\mathsf{or}\, N, \mathrm{s} \Longrightarrow \mathcal{K}}$$

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow h}{\Gamma \vdash \mathsf{raise}\, M, \mathrm{s} \Longrightarrow \mathsf{raise}\, h} \qquad \frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash \mathsf{raise}\, M, \mathrm{s} \Longrightarrow \mathcal{E}}$$

where $\mathcal{K}$ is a final configuration, which can be either a pair V, s' or an exception $\mathcal{E} = \mathsf{raise}\, h$ for some exception name h. Evaluation takes place in a type context $\Gamma = E, L$, where $E = e_1 : \mathbf{0}, \ldots, e_n : \mathbf{0}$ is called exn-*context* and L is a var-context, and a L-state s. By convention, mention of the exn-context will be omitted where possible.

Finally, the may-termination safe-equivalence is defined as: $E, L \vdash M \cong N$ if and only if for all contexts $C[-] : \mathsf{com}$,

$$E \vdash C[M] \Longrightarrow \mathsf{skip} \text{ iff } E \vdash C[N] \Longrightarrow \mathsf{skip} \;\wedge$$
$$E \vdash C[M] \Longrightarrow \mathcal{E} \text{ iff } E \vdash C[N] \Longrightarrow \mathcal{E}$$

For any integer abstraction π, let $\mathsf{blur}_{\mathsf{expint}_\pi} : \mathsf{expint} \rightarrow \mathsf{expint}$ denote an EIAA term which, given an integer n, returns a nondeterministically chosen integer m such that $m \approx_\pi n$. Since abstractions are assumed computable, such terms are definable in EIAA by iteratively testing all integers m. However, in addition to the possibilities to choose any m with $m \approx_\pi n$ nondeterministically, there is the possibility of divergence. Therefore, this approach works only for may-termination semantics, which is sufficient here. For all AIA types T, we

define EIAA terms $\mathsf{blur}_T : \widetilde{T} \to \widetilde{T}$ as follows:

$$\begin{aligned}
\mathsf{blur}_{\mathsf{expint}_\pi} = \lambda\, n : \mathsf{expint}.\mathsf{new}_{\mathsf{bool}}\, b := tt \text{ in } \mathsf{new}_{\mathsf{int}}\, m := 0 \text{ in}\\
\mathsf{while}\ (b)\ \mathsf{do}\\
\{\mathsf{if}\ (!m \approx_\pi n)\ \mathsf{then}\ b := tt\ \mathsf{or}\ \mathit{ff}\ \mathsf{else}\ \mathsf{skip};\\
\mathsf{if}\ (!m > 0)\ \mathsf{then}\ m := -!m\ \mathsf{else}\ m := -!m + 1;\ \}\\
;\ !m
\end{aligned}$$

$$\mathsf{blur}_{\mathsf{expbool}} = \lambda\, x : \mathsf{expbool}.x \qquad \mathsf{blur}_{\mathsf{com}} = \lambda\, x : \mathsf{com}.x$$

$$\mathsf{blur}_{\mathsf{var}D} = \lambda\, x : \mathsf{var}\widetilde{D}.\mathsf{mkvar}\,(\lambda\, y : \mathsf{exp}\widetilde{D}.x := \mathsf{blur}_{\mathsf{exp}D}\, y)\,(\mathsf{blur}_{\mathsf{exp}D}(!\, x))$$

$$\mathsf{blur}_{T \to T'} = \lambda f : \widetilde{T} \to \widetilde{T'}.\, \lambda\, x : \widetilde{T}.\mathsf{blur}_{T'}(f(\mathsf{blur}_T x))$$

For any AIA type T, its translation $\ulcorner T \urcorner$ into EIAA is $\widetilde{T}$. The translation of any AIA term into EIAA is defined in Table 3.1, where a context $\Gamma = x_1 : T_1, \ldots x_k : T_k$ is translated into $\ulcorner \Gamma \urcorner = x_1 : \widetilde{T_1}, \ldots, x_k : \widetilde{T_k}, \mathsf{abort} : \mathbf{0}$.

The semantic model of AIA is therefore essentially a straightforward combination of the may-termination models of EIA, which is presented in detail in [64], and of IAx presented in [76]. Since we only consider a sublanguage of IAx, where exceptions are defined in global scope and they can be only raised but they cannot be caught, the additional contingency pointers which track the flow of control in the model of IAx are not needed here. So we only use a simplified form of the model of IAx. In the next subsections, we give a formal presentation of this model.

First of all, we show a correspondence between operational semantics and observational safe-equivalence of AIA and those of EIAA.

Proposition 3.2.1 *Let $\Gamma \vdash M : T$ be an AIA term where Γ is a* var*-context, $\ulcorner \Gamma \vdash M : T \urcorner$ be its translation into EIAA, and* s *be a Γ-state. If $\Gamma \vdash M, \mathrm{s} \Longrightarrow V, \mathrm{s}'$ then $\ulcorner \Gamma \urcorner \vdash \ulcorner M \urcorner, \mathrm{s} \Longrightarrow \ulcorner V \urcorner, \mathrm{s}'$, and vice versa. If $\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}$ then*

$\ulcorner \Gamma \vdash x : T \urcorner = \ulcorner \Gamma \urcorner \vdash \mathsf{blur}_T x : \widetilde{T},\ x$ is a free identifier
$\ulcorner \Gamma \vdash \mathsf{k} : T \urcorner = \ulcorner \Gamma \urcorner \vdash \mathsf{blur}_T \mathsf{k} : \widetilde{T},\ \mathsf{k}$ is a constant (n, b, skip)
$\ulcorner \Gamma \vdash \mathsf{abort} : \mathsf{com} \urcorner = \ulcorner \Gamma \urcorner \vdash \mathsf{raise\,abort} : \mathsf{com}$
$\ulcorner \Gamma \vdash \lambda\, x : T.M : T \rightarrow T' \urcorner = \ulcorner \Gamma \urcorner \vdash \lambda\, x : \widetilde{T}.\ulcorner M : T' \urcorner : \widetilde{T} \rightarrow \widetilde{T}'$
$\ulcorner \Gamma \vdash M\, N : T' \urcorner = \ulcorner \Gamma \urcorner \vdash \ulcorner M : T_1 \rightarrow T' \urcorner \ulcorner N : T_2 \urcorner : \widetilde{T'}$
$\ulcorner \Gamma \vdash YM : T \urcorner = \ulcorner \Gamma \urcorner \vdash Y \ulcorner M : T \urcorner : \widetilde{T}$
$\ulcorner \Gamma \vdash M\ \mathsf{op}_D\ N : \mathsf{exp}D \urcorner = \ulcorner \Gamma \urcorner \vdash \mathsf{blur}_{\mathsf{exp}D}(\ulcorner M : \mathsf{exp}D_1 \urcorner\, \mathsf{op}\, \ulcorner N : \mathsf{exp}D_2 \urcorner) : \mathsf{exp}\widetilde{D}$
$\ulcorner \Gamma \vdash M\, ;\ N : B \urcorner = \ulcorner \Gamma \urcorner \vdash \ulcorner M : \mathsf{com} \urcorner;\ \ulcorner N : B \urcorner : \widetilde{B}$
$\ulcorner \Gamma \vdash \mathsf{if}_B\ M\ \mathsf{then}\ N_{tt}\ \mathsf{else}\ N_{ff} : B \urcorner =$
$\qquad \ulcorner \Gamma \urcorner \vdash \mathsf{blur}_B(\mathsf{if}\, \ulcorner M : \mathsf{expbool} \urcorner\, \mathsf{then}\, \ulcorner N_{tt} : B_1 \urcorner\, \mathsf{else}\, \ulcorner N_{ff} : B_2 \urcorner) : \widetilde{B}$
$\ulcorner \Gamma \vdash \mathsf{while}\ M\ \mathsf{do}\ N : \mathsf{com} \urcorner = \ulcorner \Gamma \urcorner \vdash \mathsf{while}\, \ulcorner M : \mathsf{expbool} \urcorner\, \mathsf{do}\, \ulcorner N : \mathsf{com} \urcorner : \mathsf{com}$
$\ulcorner \Gamma \vdash M := N : \mathsf{com} \urcorner = \ulcorner \Gamma \urcorner \vdash \ulcorner M : \mathsf{var}D_1 \urcorner := \mathsf{blur}_{\mathsf{exp}D_1} \ulcorner N : \mathsf{exp}D_2 \urcorner : \mathsf{com}$
$\ulcorner \Gamma \vdash !\, M : \mathsf{exp}D \urcorner = \ulcorner \Gamma \urcorner \vdash !\, \ulcorner M : \mathsf{var}D \urcorner : \mathsf{exp}\widetilde{D}$
$\ulcorner \Gamma \vdash \mathsf{new}_D\ x := v\ \mathsf{in}\ M : B \urcorner =$
$\qquad \ulcorner \Gamma \urcorner \vdash \mathsf{new}_{\widetilde{D}}\ x := \mathsf{blur}_{\mathsf{exp}D} \ulcorner v : \mathsf{exp}D_1 \urcorner\, \mathsf{in}\, \ulcorner M : B \urcorner : \widetilde{B}$
$\ulcorner \Gamma \vdash \mathsf{mkvar}_D\ MN : \mathsf{var}D \urcorner =$
$\qquad \ulcorner \Gamma \urcorner \vdash \mathsf{blur}_{\mathsf{var}D}(\mathsf{mkvar}\, \ulcorner M : \mathsf{exp}D_1 \rightarrow \mathsf{com} \urcorner \ulcorner N : \mathsf{exp}D_2 \urcorner) : \mathsf{var}\widetilde{D}$

Table 3.1: Translating AIA into EIAA

$\ulcorner \Gamma \urcorner \vdash \ulcorner M \urcorner, \mathrm{s} \Longrightarrow \mathcal{E}$, *and vice versa.*

Proof Suppose that $\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{K}$. The proof is by induction on the derivation of $\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{K}$.

The result is immediate for the command constants: skip and abort.

Consider the case of any integer constant n_π. We have $\Gamma \vdash n_\pi, s \Longrightarrow n', s$ for some $n' \approx_\pi n$.

On the other hand, $\ulcorner n_\pi \urcorner = \mathsf{blur}_{\mathsf{expint}_\pi} n = \mathsf{blur}_{\mathsf{expint}_\pi} n'$. Thus, $\ulcorner \Gamma \urcorner \vdash \ulcorner n_\pi \urcorner, \mathrm{s} \Longrightarrow n'', s$ for any $n'' \approx_\pi n'$.

Consider the case of any arithmetic-logic operator op_D. The first rule is

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash M\ \mathsf{op}_D\ N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

By the inductive hypothesis, we have that $\ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner, \mathrm{s} \Longrightarrow \mathcal{E}$. Then, $\ulcorner\Gamma\urcorner \vdash \ulcorner M \,\mathsf{op}_D\, N\urcorner, \mathrm{s} \Longrightarrow \mathcal{E}$. The proof for the second rule

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow v_1, \mathrm{s}' \qquad \Gamma \vdash N, \mathrm{s}' \Longrightarrow \mathcal{E}}{\Gamma \vdash M \,\mathsf{op}_D\, N, \mathrm{s} \Longrightarrow \mathcal{E}} \quad \text{is similar.}$$

The third rule is

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow v_1, \mathrm{s}' \qquad \Gamma \vdash N, \mathrm{s}' \Longrightarrow v_2, \mathrm{s}''}{\Gamma \vdash M \,\mathsf{op}_D\, N, \mathrm{s} \Longrightarrow v, \mathrm{s}''}\; v_1 \,\mathsf{op}\, v_2 \approx_D v$$

By the inductive hypothesis, we have $\ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner, \mathrm{s} \Longrightarrow \ulcorner v_1\urcorner, \mathrm{s}'$ and $\ulcorner\Gamma\urcorner \vdash \ulcorner N\urcorner, \mathrm{s}' \Longrightarrow \ulcorner v_2\urcorner, \mathrm{s}''$. So, $\ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner, \mathrm{s} \Longrightarrow v_1, \mathrm{s}'$ and $\ulcorner\Gamma\urcorner \vdash \ulcorner N\urcorner, \mathrm{s}' \Longrightarrow v_2, \mathrm{s}''$. Since $\ulcorner M \,\mathsf{op}_D\, N\urcorner = \mathsf{blur}_{\mathsf{exp}D}(\ulcorner M\urcorner \,\mathsf{op}\, \ulcorner N\urcorner)$, we have $\ulcorner\Gamma\urcorner \vdash \ulcorner M \,\mathsf{op}_D\, N\urcorner, \mathrm{s} \Longrightarrow \ulcorner v\urcorner, \mathrm{s}''$ for any $v \approx_D v_1 \,\mathsf{op}\, v_2$.

Consider the case of application. The first rule is

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash M\, N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

By the inductive hypothesis, we have that $\ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner, \mathrm{s} \Longrightarrow \mathcal{E}$. Then, $\ulcorner\Gamma\urcorner \vdash \ulcorner M\, N\urcorner, \mathrm{s} \Longrightarrow \mathcal{E}$.

The second rule is

$$\frac{\Gamma \vdash M, \mathrm{s} \Longrightarrow \lambda\, x : T.M', \mathrm{s}' \qquad \Gamma \vdash M'[N/x], \mathrm{s}' \Longrightarrow \mathcal{K}}{\Gamma \vdash M\, N, \mathrm{s} \Longrightarrow \mathcal{K}}$$

By the inductive hypothesis, we have $\ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner, \mathrm{s} \Longrightarrow \ulcorner\lambda\, x : T.M'\urcorner, \mathrm{s}'$ and $\ulcorner\Gamma\urcorner \vdash \ulcorner M'[N/x]\urcorner, \mathrm{s}' \Longrightarrow \ulcorner\mathcal{K}\urcorner$. Since $\ulcorner\lambda\, x : T.M'\urcorner = \lambda\, x : \widetilde{T}.\ulcorner M'\urcorner$, $\ulcorner M'[N/x]\urcorner = \ulcorner M'\urcorner[\ulcorner N\urcorner/x]$, and $\ulcorner M\, N\urcorner = \ulcorner M\urcorner\ulcorner N\urcorner$, we have that $\ulcorner\Gamma\urcorner \vdash \ulcorner M\, N\urcorner, \mathrm{s} \Longrightarrow \ulcorner\mathcal{K}\urcorner$. The proofs for the remaining cases are similar.

The proof for the converse is similar. ∎

Proposition 3.2.2 *For any AIA terms* $\Gamma \vdash M, N : T$*,*

$$\Gamma \vdash M \sqsubseteq_{\sim} \Gamma \vdash N \quad \textit{iff} \quad \ulcorner\Gamma \vdash M\urcorner \sqsubseteq_{\sim} \ulcorner\Gamma \vdash N\urcorner$$

Proof Let $\Gamma \vdash M, N : T$ be two AIA terms and $C[-] : \mathsf{com}$ be a program context. Suppose that $\ulcorner \Gamma \vdash M \urcorner \sqsubseteq \ulcorner \Gamma \vdash N \urcorner$.

Let $C[M]$ may terminate successfully (resp., abnormally). Then by Proposition 3.2.1, $\ulcorner C[M] \urcorner$ may also terminate successfully (resp., abnormally). Using the fact that $\ulcorner \Gamma \vdash M \urcorner \sqsubseteq \ulcorner \Gamma \vdash N \urcorner$ and Proposition 3.2.1, we obtain that $C[N]$ may terminate successfully (resp., abnormally). Therefore, we have $\Gamma \vdash M \sqsubseteq \Gamma \vdash N$.

For the opposite direction, the proof is analogous. ∎

3.3 Games and Strategies

In this section, we construct a cartesian closed category which can be used to interpret EIAA.

3.3.1 Games

In game semantics, a game is played in an *arena* which can be thought of as a playing area setting out certain basic conditions and rules for the game. The game has two participants: Player, which represents the program, and Opponent, which represents its environment (context). Opponent always moves first, and thereafter the two make moves alternately. Each of the moves can be either a question (a demand for information) or an answer (a supply of information).

Definition An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where

- M_A is a countable set of *moves*.
- $\lambda_A : M_A \rightarrow \{\mathrm{O}, \mathrm{P}\} \times \{\mathrm{Q}, \mathrm{A}\}$ is a labelling function which indicates

whether a move is by *Opponent*(O) or *Player*(P), and whether it is a *question*(Q) or an *answer*(A). We write the set $\{O, P\} \times \{Q, A\}$ as $\{OQ, OA, PQ, PA\}$, and use λ_A^{OP} to mean the composite of λ_A with the left projection, so that $\lambda_A^{OP}(m) = O$ if $\lambda_A(m) = OQ$ or $\lambda_A(m) = OA$. λ_A^{QA} is defined as λ_A followed by the right projection in a similar way. So, we have that $\lambda_A = (\lambda_A^{OP}, \lambda_A^{QA})$. We denote by $\overline{\lambda}_A$ the labelling with the O/P part reversed, i.e. $\overline{\lambda}_A^{OP}(m) = O$ iff $\lambda_A^{OP}(m) = P$.

- $\vdash_A$ is a binary relation between $M_A + \{*\}$ (where $+$ is a disjoint union and $*$ is a dummy symbol) and M_A, called *enabling* (if $m \vdash_A n$ we say that m enables move n), which satisfies the following conditions:

  (i) $* \vdash_A m \;\Rightarrow\; \lambda_A(m) = OQ \;\wedge\;$ for all $n, n \vdash_A m \Leftrightarrow n = *$

  (ii) $m \vdash_A n \;\wedge\; \lambda_A^{QA}(n) = A \;\Rightarrow\; \lambda_A^{QA}(m) = Q$

  (iii) $m \vdash_A n \;\wedge\; m \neq * \;\Rightarrow\; \lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$

The enabling relation defines, when a move is played, what moves are enabled to be made subsequently. A move enabled by the special enabler $*$ is called *initial*. Condition (i) says that initial moves are Opponent questions, and they are not enabled by any other moves besides $*$. Conditions (ii) and (iii) say that answers are enabled by questions, and that two participants always enable each other's moves, never their own (i.e. an Opponent move can only enable a Player move and vice versa).

Before proceeding further with the presentation, we introduce some notation. If Σ is an alphabet, then we denote by Σ^* the set of all finite sequences over Σ. If s and t are such sequences, then we write st or $s \cdot t$ for their concatenation. We also write sm or $s \cdot m$ for the sequence s with

element $m \in \Sigma$ appended. The empty sequence is written as ε, and $\sqsubseteq$ denotes the prefix ordering on sequences. We use meta-variables m, n to range over moves, m_{Q} over question moves, and m_{A} over answer moves.

Definition A *justified sequence* s in arena A is a finite sequence of moves of A together with a pointer from each non-initial move n to an earlier move m such that $m \vdash_A n$. We say that the move n is (explicitly) *justified* by m or that m *justifies* n.

Note that the first move in any justified sequence must be an initial move. It is also important to note that justification refers to particular instances of moves occurring in a justified sequence.

Definition A justified sequence s in arena A is a *legal play* if it satisfies the following *alternation* condition: Opponent and Player moves strictly alternate in s, i.e. if $s = s_1 mns_2$ then $\lambda^{\mathrm{OP}}(m) \neq \lambda^{\mathrm{OP}}(n)$. We write L_A for the set of all legal plays in arena A.

Definition We say that a move n is *hereditarily justified* by a move m in a legal play s if there is a subsequence of s starting with m and ending in n such that every move is justified by the preceding move in it, i.e.

$$s \;=\; \cdots\; m \;\cdots\; m_k \;\cdots\;\; \cdots\; m_1 \;\cdots\; n \;\cdots$$

We write $s \lceil m$ for the subsequence of s containing all moves hereditarily justified by m. We similarly define $s \lceil I$ for a set I of initial moves in s to be the subsequence of s consisting of all moves hereditarily justified by a move in I.

Definition A *game* is a structure $A = \langle M_A, \lambda_A, \vdash_A, P_A \rangle$ where

- $\langle M_A, \lambda_A, \vdash_A \rangle$ is an arena, and
- P_A is a non-empty, prefix-closed subset of L_A, called the *valid plays*, such that if $s \in P_A$ and I is a set of initial moves of s then $s \lceil I \in P_A$.

Example The simplest game is the *empty game* $I = \langle \emptyset, \emptyset, \emptyset, \{\varepsilon\} \rangle$. The only valid play of I is the empty sequence ε, because there are no moves. The next simplest game has a single Opponent question move *abort* [1] and is denoted by o.

The *base types* are interpreted by the following games.

$$[\![\mathbf{0}]\!] = o = \langle M_o, \lambda_o, \vdash_o, P_o \rangle$$

where

$$\begin{aligned} & M_o = \{abort\}, \\ & \lambda_o(abort) = \mathrm{OQ} \\ & * \vdash_o abort \\ & P_o = \{\varepsilon, abort\} \end{aligned}$$

$$[\![\mathsf{exp} D]\!] = \langle M_{[\![\mathsf{exp} D]\!]}, \lambda_{[\![\mathsf{exp} D]\!]}, \vdash_{[\![\mathsf{exp} D]\!]}, P_{[\![\mathsf{exp} D]\!]} \rangle$$

where

$$\begin{aligned} & M_{[\![\mathsf{exp} D]\!]} = \{q, v \mid v \in D\} \\ & \lambda_{[\![\mathsf{exp} D]\!]}(q) = \mathrm{OQ},\ \lambda_{[\![\mathsf{exp} D]\!]}(v) = \mathrm{PA} \\ & * \vdash_{[\![\mathsf{exp} D]\!]} q,\ q \vdash_{[\![\mathsf{exp} D]\!]} v \\ & P_{[\![\mathsf{exp} D]\!]} = \{\varepsilon, q, q \cdot v \mid v \in D\} \end{aligned}$$

[1] In [76], this move is referred to as *raise*, but here we call it *abort* since only one exception can be raised.

In the game for expressions, there is an initial move q (a question: "What is the value of the expression?") and corresponding to it a value from D (an answer to the question).

$$[\![\mathsf{com}]\!] = \langle M_{[\![\mathsf{com}]\!]}, \lambda_{[\![\mathsf{com}]\!]}, \vdash_{[\![\mathsf{com}]\!]}, P_{[\![\mathsf{com}]\!]} \rangle$$

where

$$\begin{aligned}
&M_{[\![\mathsf{com}]\!]} = \{\mathit{run}, \mathit{done}\} \\
&\lambda_{[\![\mathsf{com}]\!]}(\mathit{run}) = \mathrm{OQ},\ \ \lambda_{[\![\mathsf{com}]\!]}(\mathit{done}) = \mathrm{PA} \\
&* \vdash_{[\![\mathsf{com}]\!]} \mathit{run},\ \ \mathit{run} \vdash_{[\![\mathsf{com}]\!]} \mathit{done} \\
&P_{[\![\mathsf{com}]\!]} = \{\varepsilon, \mathit{run}, \mathit{run} \cdot \mathit{done}\}
\end{aligned}$$

In the game for commands, there is an initial move *run* to initiate a command, and an answer move *done* to signal successful termination of a command.

$$[\![\mathsf{var} D]\!] = \langle M_{[\![\mathsf{var} D]\!]}, \lambda_{[\![\mathsf{var} D]\!]}, \vdash_{[\![\mathsf{var} D]\!]}, P_{[\![\mathsf{var} D]\!]} \rangle$$

where

$$\begin{aligned}
&M_{[\![\mathsf{var} D]\!]} = \{\mathit{read}, v, \mathit{write}(v), \mathit{ok} \mid v \in D\} \\
&\lambda_{[\![\mathsf{var} D]\!]}(\mathit{read}, \mathit{write}(v)) = \mathrm{OQ},\ \ \lambda_{[\![\mathsf{var} D]\!]}(v, \mathit{ok}) = \mathrm{PA} \\
&* \vdash_{[\![\mathsf{var} D]\!]} \mathit{read},\ \ * \vdash_{[\![\mathsf{var} D]\!]} \mathit{write}(v),\ \ \mathit{read} \vdash_{[\![\mathsf{var} D]\!]} v,\ \ \mathit{write}(v) \vdash_{[\![\mathsf{var} D]\!]} \mathit{ok} \\
&P_{[\![\mathsf{var} D]\!]} = \big\{\varepsilon, \mathit{read}, \mathit{write}(v), \mathit{read} \cdot v, \mathit{write}(v) \cdot \mathit{ok} \mid v \in D\big\}
\end{aligned}$$

In the game for variables, for each $v \in D$ there is an initial move $\mathit{write}(v)$ to initiate an assignment, and an answer move *ok* to signal successful completion of the assignment. For de-referencing, there is an initial move *read*, to which Player may respond with any element of D. ∎

Given games A and B, we define new games $A \times B$, $A \otimes B$, and $A \multimap B$ as follows:

$$\begin{aligned}
M_{A\times B} &= M_A + M_B \text{ (disjoint union)} \\
\lambda_{A\times B} &= [\lambda_A, \lambda_B]\text{, the source tupling} \\
* \vdash_{A\times B} n &\Leftrightarrow * \vdash_A n \ \vee \ * \vdash_B n \\
m \vdash_{A\times B} n &\Leftrightarrow m \vdash_A n \ \vee \ m \vdash_B n \\
P_{A\times B} &= P_A + P_B
\end{aligned}$$

$$\begin{aligned}
M_{A\otimes B} &= M_A + M_B \\
\lambda_{A\otimes B} &= [\lambda_A, \lambda_B] \\
* \vdash_{A\otimes B} n &\Leftrightarrow * \vdash_A n \ \vee \ * \vdash_B n \\
m \vdash_{A\otimes B} n &\Leftrightarrow m \vdash_A n \ \vee \ m \vdash_B n \\
P_{A\otimes B} &= \{s \in L_{A\otimes B} \mid s \restriction A \in P_A, s \restriction B \in P_B\}
\end{aligned}$$

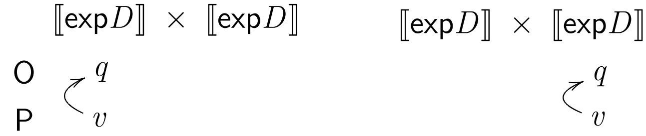
$$\begin{aligned}
M_{A\multimap B} &= M_A + M_B \\
\lambda_{A\multimap B} &= [\overline{\lambda}_A, \lambda_B] \\
* \vdash_{A\multimap B} n &\Leftrightarrow * \vdash_B n \\
m \vdash_{A\multimap B} n &\Leftrightarrow m \vdash_A n \ \vee \ m \vdash_B n \ \vee \ \big(* \vdash_B m \wedge * \vdash_A n\big)\text{, for } m \neq * \\
P_{A\multimap B} &= \{s \in L_{A\multimap B} \mid s \restriction A \in P_A, s \restriction B \in P_B\}
\end{aligned}$$

Here, $s \restriction A$ denotes the subsequence of s consisting of moves from M_A; $s \restriction B$ is analogous. A valid play of $A \times B$ is either a valid play from A or a valid play from B. In contrast, each play in $P_{A\otimes B}$ is an interleaving of a valid play from A with a valid play from B, and only Opponent can switch between interleaved plays. Since initial moves of A are enabled by initial moves of B in $A \multimap B$, an Opponent (resp., a Player) move of A is considered to be a Player (resp., an Opponent) move of $A \multimap B$. Hence, valid plays of $A \multimap B$ are interleavings

of single plays from A and B, and each such play has to begin in B and only Player can switch between the interleaved plays.

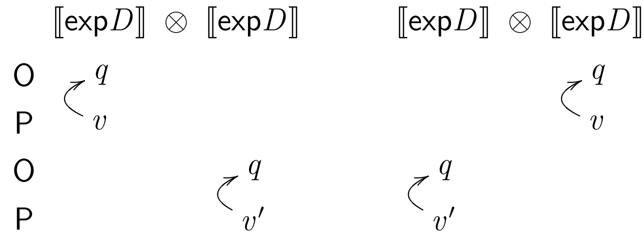
From now on, we sometimes omit the justification pointers in the valid plays, especially when they are clear from the context.

Example In the game $[\![\mathsf{exp}D]\!] \times [\![\mathsf{exp}D]\!]$, a valid play is either from the left-hand or the right-hand component.

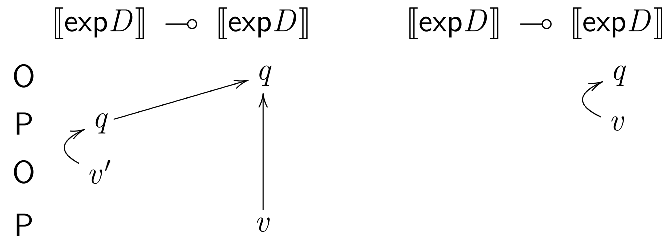


In diagrams such as the above, time flows downwards and every move is aligned with the type component in which it occurs. Here, Opponent starts by playing q in a component, and at the next step Player answers with a value v in the same component.

In the game $[\![\mathsf{exp}D]\!] \otimes [\![\mathsf{exp}D]\!]$, valid plays are shown below.



In the game $[\![\mathsf{exp}D]\!] \multimap [\![\mathsf{exp}D]\!]$, Opponent starts in the right-hand component, and Player may choose to switch to the left-hand component or not.



∎

Given a game A, the game $!A$ is defined as:

$$M_{!A} = M_A$$
$$\lambda_{!A} = \lambda_A$$
$$\vdash_{!A} = \vdash_A$$
$$P_{!A} = \{s \in L_{!A} \mid \text{for each initial move } m,\ s\lceil m \in P_A\}$$

Valid plays of $!A$ are interleavings of a finite number of plays from P_A, and only Opponent can switch between the interleaved plays. The following identities are easy to show: $!(A \times B) = !A \otimes !B$ and $!I = I$.

Finally, the game $A \Rightarrow B$ is defined as $!A \multimap B$. A valid play of $A \multimap B$ can contain a single play of A, and so it is similar to linear implication. On the other hand, a valid play of $A \Rightarrow B$ (i.e. $!A \multimap B$) can contain a finite number of plays of A, and so it is similar to classical implication.

Definition A game A is *well-opened* if and only if, for all $sm \in P_A$, if m is initial then $s = \varepsilon$.

So, in a well-opened game, initial moves can only happen at the first move. Note that although $!A$ is not usually well-opened for any game A, the games $A \multimap B$ and $!A \multimap B$ are well-opened whenever B is.

3.3.2 Strategies

Every term of the language can be interpreted by a strategy.

Definition A *strategy* σ for a game A (written as $\sigma : A$) is a prefix-closed non-empty set of even-length plays in P_A.

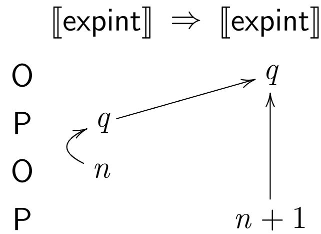
A strategy specifies what options Player has at any given point of a play, and it does not restrict the Opponent moves. We say that a play in σ is *complete* if it

is maximal and either all questions have been answered or the play terminates in *abort*.

Example The only strategy for the empty game I is the empty strategy $\bot = \{\varepsilon\}$.

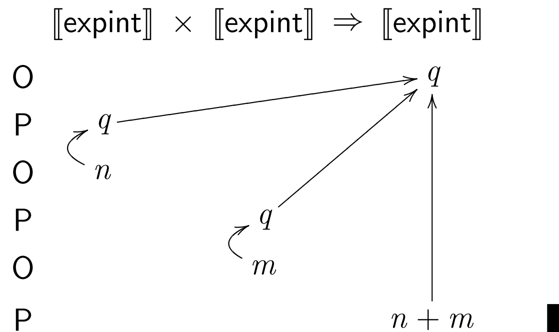
For the game $[\![\mathsf{expint}]\!]$, there is the empty strategy, and one strategy for each integer n, namely $\{\varepsilon, q \cdot n\}$.

A strategy for a successor function $\mathsf{succ} : \mathsf{expint} \to \mathsf{expint}$ [2] is as follows:



So, this strategy can be described as: "When Opponent asks for output of succ, Player will replay asking for input. When Opponent provides input n (which can be any integer since a strategy does not restrict O moves), Player will give output $(n + 1)$."

A strategy for addition $+ : \mathsf{expint} \times \mathsf{expint} \to \mathsf{expint}$ looks like this:



Note that since the empty game I has no moves, strategies for a game A and strategies for $I \Rightarrow A$ are the same.

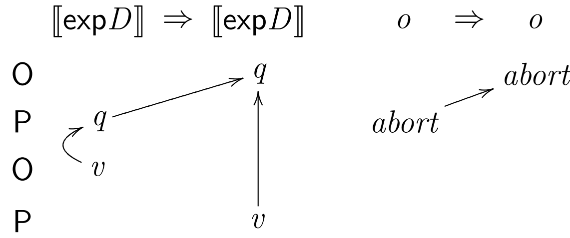
[2]Because strategies are prefix-closed, it suffices to give their sets of complete plays.

The *identity strategy* $id_A : A \Rightarrow A$ for a game A is given by the copy-cat strategy:

$$\{s \in P_{A \Rightarrow A} \mid \forall\, s' \sqsubseteq^{even} s.\ s' \upharpoonright A_l = s' \upharpoonright A_r\}$$

where we use the l and r tags to distinguish between the two occurrences of A, and $s' \sqsubseteq^{even} s$ means that s' is an even-length prefix of s. So, in any identity strategy id_A, a move by Opponent in either occurrence of A is immediately copied by Player to the other occurrence.

Example The identity strategies $id_{[\![\mathsf{exp} D]\!]}$ and id_o are as follows.



∎

The notion of composition of strategies is central to game semantics: just as small programs can be put together to form large ones, so strategies can be composed to form new strategies. Strategies compose in a way which is reminiscent of the two stage procedure of "parallel composition plus hiding" in CSP [70].

Given a strategy $\sigma : A \Rightarrow B$, we define its *promotion* $\sigma^\dagger : !A \multimap !B$, which can play several interleaved copies of σ, by:

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{for all initial}\, m,\ s\lceil m \in \sigma\}$$

Let $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ be two strategies. Then the composition $\sigma \mathbin{⨾} \tau : A \Rightarrow C$ is defined as $\sigma^\dagger\, ; \tau$, where ; is linear composition of strategies.

Given strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$, the linear composition $\sigma\, ;\ \tau : A \multimap C$ is defined in the following way. For a sequence u of moves

from games A, B, C with justification pointers, we define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C (if a pointer from one of these points to a move of A, delete that pointer). Similarly define $u \upharpoonright A, B$. We say that u is an *interaction sequence* of A, B, C if $u \upharpoonright A, B \in P_{A \multimap B}$ and $u \upharpoonright B, C \in P_{B \multimap C}$. The set of all such sequences is written as $int(A, B, C)$.

The parallel composition is defined by

$$\sigma \parallel \tau = \{u \in int(A, B, C) \mid u \upharpoonright A, B \in \sigma,\ u \upharpoonright B, C \in \tau\}$$

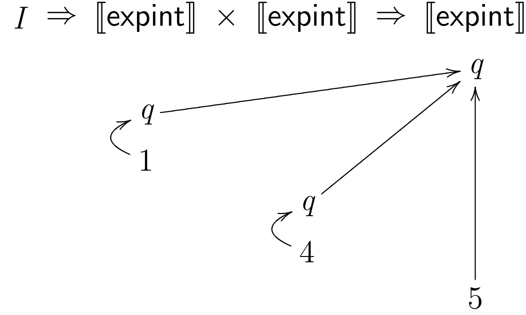
So $\sigma \parallel \tau$ consists of interaction sequences generated by playing σ and τ in parallel, making them synchronise on moves in B. When σ plays a move in B, it becomes a stimulus for τ to move, and vice versa.

Suppose $u \in int(A, B, C)$. Define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C, but where there was a pointer from a move $m_A \in M_A$ to an initial move $m \in I_B$ extend the pointer to the initial move in C which was pointed to from m. Thus, we complete the definition of composition by hiding the interaction between σ and τ in B.

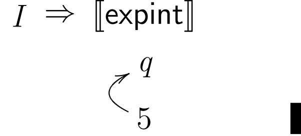
$$\sigma\, ; \tau = \{u \upharpoonright A, C \mid u \in \sigma \parallel \tau\}$$

The motivation for this definition comes from the geometry of interaction [57], interaction categories [2], and the CSP process algebra [70].

Example The strategy for the term $1 + 4 : \mathsf{expint}$ is obtained by composing the strategy for addition $+ : \mathsf{expint} \times \mathsf{expint} \rightarrow \mathsf{expint}$ with the strategy for the constant pair $(1, 4) : \mathsf{expint} \times \mathsf{expint}$. The parallel composition makes the strategies synchronise on the moves in the type of the arguments, $\mathsf{expint} \times \mathsf{expint}$.



Hiding eliminates all the moves from the type of the arguments. Thus, we obtain the strategy for $1 + 4 :$ expint,



Given $\sigma : A \multimap B$ and $\tau : C \multimap D$, we define $\sigma \otimes \tau : (A \otimes C) \multimap (B \otimes D)$ as follows:

$$\sigma \otimes \tau = \{s \in L_{(A\otimes C)\multimap(B\otimes D)} \mid s \upharpoonright A, B \in \sigma \ \wedge \ s \upharpoonright C, D \in \tau\}.$$

3.3.3 A Cartesian Closed Category

We now introduce several restrictions on strategies that are needed for the game-semantics model of EIAA. They rely on the following definition of the *Player view* of a non-empty valid play.

Definition The *Player view* $\ulcorner s \urcorner$ of a non-empty valid play $s \in P_A$ is defined by:

$$\begin{aligned} \ulcorner sm \urcorner &= m && \text{if } m \text{ is initial } (* \vdash_A m) \\ \ulcorner sm \urcorner &= \ulcorner s \urcorner m && \text{if } m \text{ is a P move} \\ \ulcorner smtn \urcorner &= \ulcorner s \urcorner mn && \text{if } n \text{ is a O move and } m \text{ justifies } n \end{aligned}$$

The restrictions we consider are the visibility condition and the bracketing condition.

Visibility condition A strategy σ satisfies *Player visibility* iff

for all $smn \in \sigma$, the justifier of the P-move n occurs in $\ulcorner sm \urcorner$.

We say that an answer move n *answers* a question move m in a valid play $s \in P_A$ iff n is justified by m in s.

Bracketing condition A strategy σ satisfies *Player bracketing* iff

for all $smn \in \sigma$, if n is an answer P-move then n answers the pending question of $\ulcorner sm \urcorner$, i.e. the most recently asked but not answered question in $\ulcorner sm \urcorner$.

It is shown in [64, pp. 57–58] that the strategies satisfying the above restrictions are closed under composition. From now on, we proceed to work only with strategies that satisfy both restrictions.

We can now define a category $\mathbf{C}$ of games as follows:

Objects are well-opened games.

Arrows $\sigma : A \to B$ are strategies for $A \Rightarrow B$ which satisfy the above two conditions.

For any well-opened game A, the strategy $id_A : A \Rightarrow A$ is the *identity arrow* on A, and the *composition* of arrows $\sigma : A \to B$ and $\tau : B \to C$ is defined to be $\sigma \fatsemi \tau = \sigma^\dagger ;\ \tau$.

Proposition 3.3.1 *The category* $\mathbf{C}$ *of games is a cartesian closed category.*

Proof

Terminal object. $\mathbf{1} \stackrel{\text{def.}}{=} I$ where $I = \langle \emptyset, \emptyset, \emptyset, \{\varepsilon\} \rangle$ is the empty game.

Product. $A \times B \stackrel{\text{def.}}{=} A \times B$, with projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ defined by the obvious copy-cat strategies:
$\pi_1 = \{s \in P_{A \times B \Rightarrow A} \mid \forall\, s' \sqsubseteq^{even} s. s' \restriction A_l = s' \restriction A_r\}$
$\pi_2 = \{s \in P_{A \times B \Rightarrow B} \mid \forall\, s' \sqsubseteq^{even} s. s' \restriction B_l = s' \restriction B_r\}$

Exponential. $B^A \stackrel{\text{def.}}{=} A \Rightarrow B$. Note that the only difference between the games $A \times B \Rightarrow C$ and $A \Rightarrow (B \Rightarrow C)$ is in the tagging of the moves in the appropriate disjoint union, i.e. $\mathbf{C}(A \times B, C) \cong \mathbf{C}(A, B \Rightarrow C)$. Given a strategy $\sigma : A \times B \Rightarrow C$, we define $\Lambda(\sigma) : A \Rightarrow (B \Rightarrow C)$, called *currying*, as the strategy corresponding to σ across this isomorphism. Then, the evaluation strategy is defined as: $eval_{A,B} = \Lambda^{-1}(id_{A \Rightarrow B})$.

Further details of the construction of CCC are handled in the same way as in the proof in [64, pp. 61–73]. ∎

The set of all strategies for $A \Rightarrow B$ which satisfy the above two conditions forms forms a cpo, where the ordering relation on strategies is defined by:

$$\sigma \leq \tau \text{ iff } \sigma \subseteq \tau \text{ for any } \sigma, \tau : A \Rightarrow B$$

The least element is $\perp = \{\varepsilon\}$, and the least upper bound is given by unions, i.e. $\sigma \sqcup \tau = \sigma \cup \tau$ for any $\sigma, \tau : A \Rightarrow B$. It is easy to check that composition of strategies is monotone and continuous [64, pp. 54]. Therefore, the category $\mathbf{C}$ is also *cpo-enriched*.

3.4 The Model

In this section we construct the model of EIAA. Since the category **C** of games is cartesian closed, it can be used to model a typed λ-calculus. We begin by reviewing the interpretation of a typed λ-calculus in a cartesian closed category and then show how constants and constructs of the language can be interpreted.

A type T will be interpreted as a game (object) $[\![T]\!]$ in the category **C**, and a well-typed open term $\Gamma \vdash M : T$, where $\Gamma = x_1 : T_1, \ldots, x_k : T_k$, as a strategy (arrow) $[\![\Gamma \vdash M : T]\!]$ for the game:

$$[\![\Gamma \vdash T]\!] = [\![T_1]\!] \times \ldots \times [\![T_k]\!] \Rightarrow [\![T]\!]$$

If M is a closed term, then it is interpreted by $[\![\vdash M : T]\!] : I \Rightarrow [\![T]\!]$. The context $\Gamma = x_1 : T_1, \ldots, x_k : T_k$ is interpreted by $[\![\Gamma]\!] = [\![T_1]\!] \times \ldots \times [\![T_k]\!]$.

The base types have already been interpreted as games in the Example on page 39. The interpretation of other types is defined by induction: $[\![T \to T']\!] = [\![T]\!] \Rightarrow [\![T']\!]$.

Free identifiers are interpreted using projections:

$$[\![x_1 : T_1, \ldots, x_k : T_k \vdash x_i : T_i]\!] = \pi_i : [\![T_1]\!] \times \ldots \times [\![T_k]\!] \Rightarrow [\![T_i]\!],\ 1 \leq i \leq k$$

Abstraction is modelled using the currying isomorphism:

$$[\![\Gamma \vdash \lambda\, x : T.M : T \to T']\!] = \Lambda([\![\Gamma, x : T \vdash M : T']\!]) : [\![\Gamma]\!] \Rightarrow ([\![T]\!] \Rightarrow [\![T']\!])$$

Application is interpreted using the evaluation strategy:

$$[\![\Gamma \vdash MN : T']\!] = \big([\![\Gamma \vdash M : T \to T']\!], [\![\Gamma \vdash N : T]\!]\big) \,;\, eval_{[\![T]\!],[\![T']\!]} : [\![\Gamma]\!] \Rightarrow [\![T']\!]$$

Recursion $\Gamma \vdash YM : T$ is interpreted in the following way. Given that the term $\Gamma \vdash M : T \to T$ is modelled by a strategy $[\![\Gamma \vdash M : T \to T]\!]$, we

define a chain of strategies $(f_i)_{i\in\mathbb{N}}$ for the game $[\![\Gamma]\!] \Rightarrow [\![T]\!]$, such that $f_n \leq f_m$ whenever $n \leq m$, as:

$$\begin{aligned} f_0 &= \bot\colon [\![\Gamma]\!] \Rightarrow [\![T]\!] \\ f_{n+1} &= \left(id_{[\![\Gamma]\!]}, f_n\right) \fatsemi \Lambda^{-1}([\![\Gamma \vdash M : T \to T]\!]) : [\![\Gamma]\!] \Rightarrow [\![T]\!] \end{aligned}$$

Since the category $\mathbf{C}$ is cpo-enriched there exists a least upper bound of this chain, and it represents the interpretation of recursion, i.e.

$$[\![\Gamma \vdash YM : T]\!] = \bigsqcup_{i\in\mathbb{N}} f_i$$

Constants are interpreted as follows.

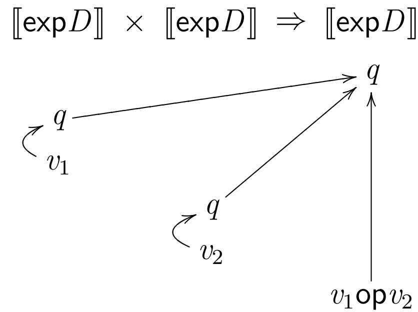
$$\begin{aligned} [\![n : \mathsf{expint}]\!] : I \Rightarrow [\![\mathsf{expint}]\!] &= \{\varepsilon, q\cdot n\} \\ [\![b : \mathsf{expbool}]\!] : I \Rightarrow [\![\mathsf{expbool}]\!] &= \{\varepsilon, q\cdot b\} \\ [\![\mathsf{skip} : \mathsf{com}]\!] : I \Rightarrow [\![\mathsf{com}]\!] &= \{\varepsilon, run\cdot done\} \end{aligned}$$

The semantics of arithmetic-logic operations is defined as

$$[\![\Gamma \vdash M \,\mathsf{op}\, N]\!] = \left([\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!]\right) \fatsemi \sigma_{\mathsf{op}} : [\![\Gamma]\!] \Rightarrow [\![\mathsf{exp}D]\!]$$

where the strategy $\sigma_{\mathsf{op}} : [\![\mathsf{exp}D]\!] \times [\![\mathsf{exp}D]\!] \Rightarrow [\![\mathsf{exp}D]\!]$ is given by:

$$[\![\mathsf{exp}D]\!] \;\times\; [\![\mathsf{exp}D]\!] \;\Rightarrow\; [\![\mathsf{exp}D]\!]$$



Sequencing is interpreted by:

$$[\![\Gamma \vdash M\,;\ N]\!] = \left([\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!]\right) \fatsemi \sigma_{\mathsf{seq}} : [\![\Gamma]\!] \Rightarrow [\![B]\!]$$

where σ_{seq} is

$$[\![\mathsf{com}]\!] \times [\![B]\!] \Rightarrow [\![B]\!]$$

$$\begin{array}{llll} & & & m_Q \\ run & & & \\ done & & & \\ & m_Q & & \\ & m_A & & \\ & & & m_A \end{array}$$

Conditional is interpreted as

$$[\![\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_{tt}\ \mathsf{else}\ N_{\mathit{ff}}]\!] = \big([\![\Gamma \vdash M]\!], [\![\Gamma \vdash N_{tt}]\!], [\![\Gamma \vdash N_{\mathit{ff}}]\!]\big) \,{}_9^{\circ}\, \sigma_{\mathsf{if}} : [\![\Gamma]\!] \Rightarrow [\![B]\!]$$

where the strategy $\sigma_{\mathsf{if}} : [\![\mathsf{expbool}]\!] \times [\![B]\!] \times [\![B]\!] \Rightarrow [\![B]\!]$ is represented by its complete plays:

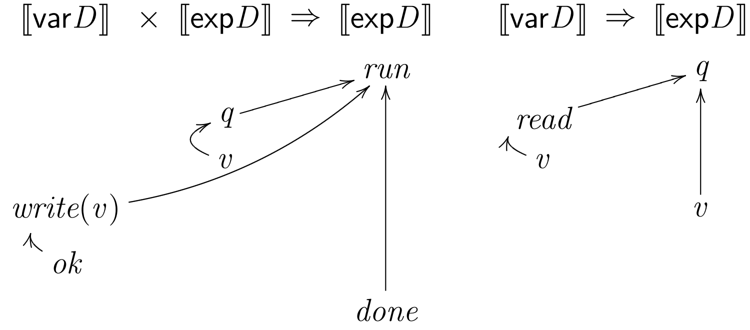
$$[\![\mathsf{expbool}]\!] \times [\![B]\!] \times [\![B]\!] \Rightarrow [\![B]\!] \qquad [\![\mathsf{expbool}]\!] \times [\![B]\!] \times [\![B]\!] \Rightarrow [\![B]\!]$$

$$\begin{array}{llll} & & & m_Q \\ q & & & \\ tt & & & \\ & m_Q & & \\ & m_A & & \\ & & & m_A \end{array} \qquad \begin{array}{llll} & & & m_Q \\ q & & & \\ \mathit{ff} & & & \\ & & m_Q & \\ & & m_A & \\ & & & m_A \end{array}$$

Assignment and de-referencing are defined by:

$$\begin{aligned} [\![\Gamma \vdash M := N]\!] &= \big([\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!]\big) \,{}_9^{\circ}\, \sigma_{\mathsf{assign}} : [\![\Gamma]\!] \Rightarrow [\![\mathsf{com}]\!] \\ [\![\Gamma \vdash\, !\, M]\!] &= [\![\Gamma \vdash M]\!] \,{}_9^{\circ}\, \sigma_{\mathsf{deref}} : [\![\Gamma]\!] \Rightarrow [\![\mathsf{exp}D]\!] \end{aligned}$$

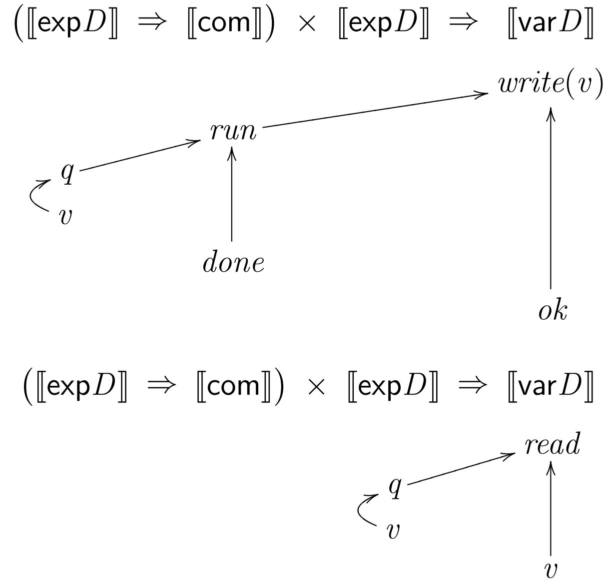
where the strategies $\sigma_{\mathsf{assign}} : [\![\mathsf{var}D]\!] \times [\![\mathsf{exp}D]\!] \Rightarrow [\![\mathsf{com}]\!]$ and $\sigma_{\mathsf{deref}} : [\![\mathsf{var}D]\!] \Rightarrow [\![\mathsf{exp}D]\!]$ are:



The constructor ‘mkvar’ is interpreted by:

$$[\![\Gamma \vdash \mathsf{mkvar}\, MN]\!] = \big([\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!]\big) \,;\, \sigma_{\mathsf{mkvar}} : [\![\Gamma]\!] \Rightarrow [\![\mathsf{var}D]\!]$$

where the transformation strategy σ_{mkvar} is



The ‘new’ constructor is interpreted using a “storage cell” strategy $\mathsf{cell} : I \Rightarrow ![\![\mathsf{var}D]\!]$, whose plays are of the form

$$read \cdot v_0 \cdot write(v_1) \cdot ok \cdot read \cdot v_1 \ldots$$

where each *read* and *write*(−) move is initial, and all other moves are justified by the immediately preceding move. So, cell responds to a *write*(v) with *ok*,

and to *read* with the last value written, if any. If there has been no value written yet, an initialised cell cell_{v_0} will respond to *read* with v_0. Now, we can interpret the new constructor as

$$[\![\Gamma \vdash \mathsf{new}_D\, x := v \,\mathsf{in}\, M]\!] = \big(id^{\dagger}_{[\![\Gamma]\!]} \otimes \mathsf{cell}_v\big)\,;\; [\![\Gamma, x : \mathsf{var}D \vdash M]\!] : [\![\Gamma]\!] \Rightarrow [\![B]\!]$$

Erratic choice operator is defined by:

$$[\![\Gamma \vdash M \,\mathsf{or}\, N]\!] = \big([\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!]\big) \mathbin{⨾} \sigma_{\mathsf{choice}} : [\![\Gamma]\!] \Rightarrow [\![\mathsf{exp}D]\!]$$

where $\sigma_{\mathsf{choice}} : [\![\mathsf{exp}D]\!] \times [\![\mathsf{exp}D]\!] \Rightarrow [\![\mathsf{exp}D]\!]$ is the only nondeterministic strategy in the model given by:

$$\begin{array}{ccccc} [\![\mathsf{exp}D]\!] & \times & [\![\mathsf{exp}D]\!] & \Rightarrow & [\![\mathsf{exp}D]\!] \\ & & & & q \\ q & & & & \\ v & & & & \\ & & & & v \end{array} \qquad \begin{array}{ccccc} [\![\mathsf{exp}D]\!] & \times & [\![\mathsf{exp}D]\!] & \Rightarrow & [\![\mathsf{exp}D]\!] \\ & & & & q \\ & & q & & \\ & & v & & \\ & & & & v \end{array}$$

The semantics of the raise operator is defined by:

$$[\![\Gamma \vdash \mathsf{raise}\, M]\!] \;=\; [\![\Gamma \vdash M]\!] \mathbin{⨾} \sigma_{\mathsf{raise}} : [\![\Gamma]\!] \Rightarrow [\![B]\!]$$

where $\sigma_{\mathsf{raise}} : [\![\mathbf{0}]\!] \Rightarrow [\![B]\!]$ is the strategy which responds to the initial question in $[\![B]\!]$ with the unique question in $[\![\mathbf{0}]\!] (= o)$:

$$\begin{array}{ccc} o & \Rightarrow & [\![B]\!] \\ & & m_{\mathsf{Q}} \\ abort & & \end{array}$$

Iteration can be represented using recursion:

$$\mathsf{while}\, M \,\mathsf{do}\, N \;=\; Y\big(\lambda\, \mathsf{c} : \mathsf{com}\,.\, \mathsf{if}\, M \,\mathsf{then}\, \{N\,;\; \mathsf{c}\} \,\mathsf{else}\, \mathsf{skip}\big).$$

3.5 Soundness

We now show the soundness of this model with respect to safe-equivalence, i.e.

$$[\![\Gamma \vdash M]\!] \leq [\![\Gamma \vdash N]\!] \ \Rightarrow \ \Gamma \vdash M \sqsubset\!\!\sim N$$

First, we show the *consistency* result that any term which may converge has an appropriate play in its semantics.

Let $L = x_1 : \mathsf{var}D_1, \ldots, x_k : \mathsf{var}D_k$ be a var-context and s be a L-state. The state s can be modelled by a strategy $\sigma_{\mathrm{s}} : I \Rightarrow !([\![\mathsf{var}D_1]\!] \times \ldots \times [\![\mathsf{var}D_k]\!])$ consisting of a tuple of suitably initialised cell strategies, i.e. $\sigma_{\mathrm{s}} = \mathsf{cell}_{\mathrm{s}(x_1)} \otimes \ldots \otimes \mathsf{cell}_{\mathrm{s}(x_k)}$. We define a strategy $[\![\mathrm{s}]\!] : [\![E]\!] \Rightarrow !([\![E]\!] \times [\![L]\!])$ as $[\![\mathrm{s}]\!] = id^{\dagger}_{[\![E]\!]} \otimes \sigma_{\mathrm{s}}$. The interpretation of a term $E, L \vdash M : T$ in a L-state s is defined by the linear composition $[\![\mathrm{s}]\!]\,;\ [\![E, L \vdash M : T]\!] : [\![E]\!] \Rightarrow [\![T]\!]$. We will also consider the interaction sequences $[\![\mathrm{s}]\!] \parallel [\![E, L \vdash M : T]\!]$. For any sequence $it't \in [\![\mathrm{s}]\!] \parallel [\![E, L \vdash M : T]\!]$, where i is an initial question, t' is an even-length sequence of moves in $![\![L]\!]$, and t is a sequence of moves, we say that t' *leaves state* s' if for each x_i in L, the last write move of t' in $\mathsf{var}D_i$ sets x_i to $\mathrm{s}'(x_i)$, and if there is no write in $\mathsf{var}D_i$ then $\mathrm{s}'(x_i) = \mathrm{s}(x_i)$.

Proposition 3.5.1 *Let $E, L \vdash M : T$ be a term and* s *be a L-state. If $E, L \vdash M, \mathrm{s} \Longrightarrow V, \mathrm{s}'$ then for each $i \cdot t \in [\![\mathrm{s}']\!] \parallel [\![V]\!]$, there exists some $i \cdot t' \cdot t \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$ such that t' plays solely in $[\![L]\!]$ and leaves state s'. If $E, L \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}$ then there exists some $i \cdot t' \cdot abort \cdot abort \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$ such that t' plays solely in $[\![L]\!]$.*

Proof Suppose $E, L \vdash M, \mathrm{s} \Longrightarrow V, \mathrm{s}'$. Then the proof is by induction on the derivation of $E, L \vdash M, \mathrm{s} \Longrightarrow V, \mathrm{s}'$, and it is handled in the same way as the proof in [64, pp. 90–91].

Suppose $E, L \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}$. The proof is a straightforward induction on the derivation of $E, L \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}$. We consider the case of any arithmetic-logic operator op. The first rule ending in $E, L \vdash M \,\mathsf{op}\, N, \mathrm{s} \Longrightarrow \mathcal{E}$ is:

$$\frac{E, L \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}}{E, L \vdash M \,\mathsf{op}\, N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

By the inductive hypothesis, we have $q \cdot t' \cdot abort \cdot abort \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$, where t' is a play in $[\![L]\!]$. By the definition of $[\![M \,\mathsf{op}\, N]\!]$, we must have a play of the following form:

$$\begin{array}{ccccccccccc}
[\![E]\!] & \Rightarrow & \big([\![E]\!] & \times & [\![L]\!]\big) & \Rightarrow & \big([\![\mathsf{exp}D]\!] & \times & [\![\mathsf{exp}D]\!]\big) & \Rightarrow & [\![\mathsf{exp}D]\!] \\
 & & & & & & & & & & q \\
 & & & & & & q & & & & \\
 & & & & t' & & & & & & \\
 & & abort & & & & & & & & \\
abort & & & & & & & & & &
\end{array}$$

So, we have that $q \cdot t' \cdot abort \cdot abort \in [\![\mathrm{s}]\!] \parallel [\![M \,\mathsf{op}\, N]\!]$ where t' plays solely in $[\![L]\!]$.

The second rule is:

$$\frac{E, L \vdash M, \mathrm{s} \Longrightarrow v_1, \mathrm{s}' \qquad E, L \vdash N, \mathrm{s}' \Longrightarrow \mathcal{E}}{E, L \vdash M \,\mathsf{op}\, N, \mathrm{s} \Longrightarrow \mathcal{E}}$$

By the inductive hypothesis, we have $q \cdot t' \cdot v_1 \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$, where t' leaves state s' and $q \cdot t'' \cdot abort \cdot abort \in [\![\mathrm{s}']\!] \parallel [\![N]\!]$. Then, by the definition of $[\![M \,\mathsf{op}\, N]\!]$, there is a play:

$$\begin{array}{ccccccccccc}
[\![E]\!] & \Rightarrow & \big([\![E]\!] & \times & [\![L]\!]\big) & \Rightarrow & \big([\![\mathsf{exp}D]\!] & \times & [\![\mathsf{exp}D]\!]\big) & \Rightarrow & [\![\mathsf{exp}D]\!] \\
 & & & & & & & & & & q \\
 & & & & & & q & & & & \\
 & & & & t' & & & & & & \\
 & & & & & & v_1 & & & & \\
 & & & & & & & & q & & \\
 & & & & t'' & & & & & & \\
 & & abort & & & & & & & & \\
abort & & & & & & & & & &
\end{array}$$

So, we have $q \cdot t' \cdot t'' \cdot abort \cdot abort \in [\![\mathrm{s}]\!] \parallel [\![M \,\mathsf{op}\, N]\!]$ where t' and t'' play solely in $[\![L]\!]$. The other cases in the inductive proof are similar. ∎

Corollary 3.5.2 (Consistency) *Let $E \vdash M : \mathsf{com}$ be a term. If $E \vdash M \Longrightarrow \mathsf{skip}$ then $run \cdot done \in [\![M]\!]$, and if $E \vdash M \Longrightarrow \mathcal{E}$ then $run \cdot abort \in [\![M]\!]$.*

Next, we prove that the converse of this result is also true, i.e. if a term's game semantics contains an appropriate play then the term may converge. This result is known as *computational adequacy*. The proof of this result makes use of logical relations [92].

Let $E, L, \Delta \vdash M : T$ be a term where E is a exn-context, L is a var-context, and Δ is arbitrary. We say that $E, L \mid \Delta \vdash M : T$ is a *split term*. Split terms with empty Δ are called *semi-closed*, and we denote them as $E, L \mid \vdash M : T$. We now define a predicate of computability on split terms.

Computability

- $E, L \mid \vdash M : \mathsf{com}$ is computable iff, for any L-state s, if $run \cdot t \cdot done \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$ where t leaves s$'$ then $E, L \vdash M, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'$, and if $run \cdot t \cdot abort \cdot abort \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$ then $E, L \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}$.

- $E, L \mid \vdash M : \mathsf{exp}D$ is computable iff, for any L-state s, if $q \cdot t \cdot v \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$ where t leaves s$'$ then $E, L \vdash M, \mathrm{s} \Longrightarrow v, \mathrm{s}'$, and if $q \cdot t \cdot abort \cdot abort \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$ then $E, L \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}$.

- $E, L \mid \vdash M : \mathbf{0}$ is computable iff, for any L-state s, if $abort \cdot abort \cdot abort \in [\![\mathrm{s}]\!] \parallel [\![M]\!]$ then $E, L \vdash M, \mathrm{s} \Longrightarrow \mathcal{E}$.

- $E, L \mid \vdash M : \mathsf{var}D$ is computable iff, $E, L \mid \vdash !M : \mathsf{exp}D$ is computable, and $E, L \mid \vdash M := v : \mathsf{com}$ for all $v \in D$ is computable.

- $E, L \mid \vdash M : T \to T'$ is computable iff, $E, L \mid \vdash M\,N : T'$ is computable for all computable $E, L \mid \vdash N : T$.

- $E, L \mid x_1 : T_1, \ldots, x_k : T_k \vdash M : T$ is computable iff, for all computable $E, L \mid \vdash N_i : T_i$, $E, L \mid \vdash M[N_1/x_1, \ldots, N_k/x_k] : T$ is computable.

Proposition 3.5.3 *All split terms are computable.*

Proof The proof is by induction on the structure of M.

The cases when a play corresponds to successful termination of a term are handled in the same way as in the proof in [64, pp. 92–93].

Suppose a play corresponds to abnormal termination. Consider the case of sequencing. Let $E, L \mid \Delta \vdash M \,;\, N : \mathsf{com}$ be a term, and let M' and N' be M and N with the free identifiers from Δ instantiated by semi-closed, computable terms. By the inductive hypothesis, M and N are both computable, and so are M' and N'. If $run \cdot t \cdot abort \cdot abort \in \llbracket \mathrm{s} \rrbracket \parallel \llbracket M' \,;\, N' \rrbracket$ then either $t = t_1 t_2$ where $run \cdot t_1 \cdot done \in \llbracket \mathrm{s} \rrbracket \parallel \llbracket M' \rrbracket$ and $run \cdot t_2 \cdot abort \cdot abort \in \llbracket \mathrm{s}' \rrbracket \parallel \llbracket N' \rrbracket$, or $run \cdot t \cdot abort \cdot abort \in \llbracket \mathrm{s} \rrbracket \parallel \llbracket M' \rrbracket$, i.e. we have two plays of the following form:

$$
\begin{array}{ccccccccccc}
\llbracket E \rrbracket & \Rightarrow & (\llbracket E \rrbracket & \times & \llbracket L \rrbracket) & \Rightarrow & (\llbracket \mathsf{com} \rrbracket & \times & \llbracket \mathsf{com} \rrbracket) & \Rightarrow & \llbracket \mathsf{com} \rrbracket \\
 & & & & & & & & & & run \\
 & & & & & & run & & & & \\
 & & & & t_1 & & & & & & \\
 & & & & & & done & & & & \\
 & & & & & & & & run & & \\
 & & & & t_2 & & & & & & \\
 & & abort & & & & & & & & \\
abort & & & & & & & & & & \\
\hline
 & & & & & & & & & & run \\
 & & & & & & run & & & & \\
 & & & & t & & & & & & \\
 & & abort & & & & & & & & \\
abort & & & & & & & & & &
\end{array}
$$

In the first case, by the inductive hypothesis, $E, L \vdash M', \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'$ for the s' left by t_1 and $E, L \vdash N', \mathrm{s}' \Longrightarrow \mathcal{E}$. So $E, L \vdash M' \,;\, N', \mathrm{s} \Longrightarrow \mathcal{E}$. In the second

case, $E, L \vdash M', \mathrm{s} \Longrightarrow \mathcal{E}$, and so $E, L \vdash M' \ ; \ N', \mathrm{s} \Longrightarrow \mathcal{E}$. It follows that $M' \, ; \, N'$ and $M \, ; \, N$ are computable. The other cases are similar. ∎

Corollary 3.5.4 (Adequacy) *Let $E \vdash M : \mathsf{com}$ be a term. If $run \cdot done \in [\![M]\!]$ then $E \vdash M \Longrightarrow \mathsf{skip}$, and if $run \cdot abort \in [\![M]\!]$ then $E \vdash M \Longrightarrow \mathcal{E}$.*

The consistency and adequacy results give us a soundness theorem.

Theorem 3.5.5 (Soundness) *For any terms $\Gamma \vdash M, N : T$, if $[\![\Gamma \vdash M]\!] \leq [\![\Gamma \vdash N]\!]$ then $\Gamma \vdash M \lesssim N$.*

Proof Let $C[-]$ be any program context and suppose that $C[M]$ may terminate successfully (resp., abnormally), so that by Corollary 3.5.2, $run \cdot done \in [\![C[M]]\!]$ (resp., $run \cdot abort \in [\![C[M]]\!]$). Since $[\![\Gamma \vdash M]\!] \leq [\![\Gamma \vdash N]\!]$, we have that $[\![C[M]]\!] \leq [\![C[N]]\!]$ by the compositionality of game semantics (i.e. monotonicity of composition of strategies), and so $run \cdot done \in [\![C[N]]\!]$ (resp., $run \cdot abort \in [\![C[N]]\!]$). By Corollary 3.5.4, $C[N]$ may terminate successfully (resp., abnormally). Therefore, $\Gamma \vdash M \lesssim N$. ∎

3.6 Definability

Next, we want to show that every element of this model is the interpretation of some term of EIAA. We assume that all strategies mentioned below satisfy both the visibility and bracketing conditions. First, let us introduce some definitions.

Definition A strategy $\sigma : A$ is *deterministic* iff, if $smn, smn' \in \sigma$ then $n = n'$ and the justifier of n is the same as that of n'.

Definition A deterministic strategy $\sigma : A$ is *innocent* iff

$$smn \in \sigma \ \wedge \ t \in \sigma \ \wedge tm \in P_A \ \wedge \ulcorner tm \urcorner = \ulcorner sm \urcorner \ \Rightarrow \ tmn \in \sigma$$

In other words, how σ plays depends only on the current Player view.

Definition For any deterministic, innocent strategy $\sigma : A$, we define the *view function* of σ to be the partial function f from Player views to Player moves defined by:

$$f(v) = n \ \Leftrightarrow \ \exists\, sm.smn \in \sigma \wedge \ulcorner sm \urcorner = v$$

The *compact* strategies are those with finite view functions.

The definability proof relies on three things: two factorization theorems, which reduce the question of definability for nondeterministic strategies to that for innocent strategies, and the innocent definability result, which shows that any innocent strategy with finite view function is definable in IAA$-$new. We refer to the sublanguage of EIAA obtained by omitting the or and new constructs as IAA$-$new.

Proposition 3.6.1 *If $\sigma : A$ is a nondeterministic strategy then there exists a deterministic strategy $det(\sigma) : [\![\mathsf{expint}]\!] \Rightarrow A$ and a strategy $oracle : I \Rightarrow [\![\mathsf{expint}]\!]$ such that $\sigma = oracle \mathbin{;} det(\sigma)$. Furthermore, if σ is compact then so is $det(\sigma)$.*

Proof The proof is given in [64, pp. 73–76]. ∎

Proposition 3.6.2 *If $\sigma : A$ is a deterministic strategy then there exists an innocent strategy $inn(\sigma) : [\![\mathsf{var}D]\!] \Rightarrow A$ and a strategy $\mathsf{cell} : I \Rightarrow ![\![\mathsf{var}D]\!]$ such that $\sigma = \mathsf{cell};\ inn(\sigma)$. Furthermore, if σ is compact then so is $inn(\sigma)$.*

Proof The proof is given in [64, pp. 82–84]. ∎

Proposition 3.6.3 *Suppose that $T_1, \ldots, T_n$ are IAA types interpreted respectively by games $A_1, \ldots, A_n$ and let $\sigma : (A_1 \times \ldots \times A_i) \Rightarrow \big((A_{i+1} \Rightarrow \ldots \Rightarrow A_n) \Rightarrow B\big)$ be a compact, innocent strategy where B is o, $[\![\mathsf{exp}D]\!]$, $[\![\mathsf{com}]\!]$, or $[\![\mathsf{var}D]\!]$. Then there exists an IAA$-$*new* term $x_1 : T_1, \ldots, x_i : T_i \vdash M : T_{i+1} \to \ldots \to T_n \to T$ such that $\sigma = [\![M]\!]$.*

Proof The proof is by induction on the view function of σ. In the base case, $\sigma = \{\varepsilon\}$ so we can set $M = Y(\lambda\, x : T.x)$. Next, we analyse the answer of σ to the initial question in B.

- The cases when B is $[\![\mathsf{exp}D]\!]$, $[\![\mathsf{com}]\!]$, or $[\![\mathsf{var}D]\!]$ and σ replies with an answer move in B are handled as in [64, pp. 94–95].
- If B is o and there is no response from σ, then $M = Y(\lambda\, x : \mathbf{0}.x)$.

Let σ responds to the initial question i by playing a question i_j in some A_j. We uncurry σ, obtaining a strategy $\sigma' : (A_1 \times \ldots \times A_n) \Rightarrow B$. We separate out all moves in A_j which will be hereditarily justified by i_j into an extra copy of A_j, obtaining a strategy $\sigma'' : (A_1 \times \ldots \times A_n \times A_j) \Rightarrow B$, which responds to the initial question i with i_j in this new copy of A_j.

Consider the view $\ulcorner smn \urcorner$ for some $smn \in \sigma''$. It either contains an immediate answer to i_j, or no answer to i_j at all. In the latter case, $\ulcorner smn \urcorner = i i_j s'$ in which i_j is not answered. Since σ satisfies bracketing condition, i is not answered either. Therefore the sequence s' consists entirely of moves in $A_1, \ldots, A_n$ and the extra copy of A_j. Suppose that A_j is of the form $A'_1 \Rightarrow \ldots \Rightarrow A'_l \Rightarrow B_j$. Then s' in the extra copy A_j is restricted to $A'_1, \ldots, A'_l$. So we consider s' as a play in $(A_1 \times \ldots \times A_n) \Rightarrow (A'_1 \times \ldots \times A'_l)$. The set of all Player

views of this form induces an innocent strategy for this game, $(\sigma'_1, \ldots, \sigma'_l)$. The view function of each σ'_i is smaller than that of σ, so $\sigma'_i = [\![M'_i]\!]$ for some term M'_i by the inductive hypothesis. We analyse the structure of B_j.

- The cases when B_j is $[\![\mathsf{exp}D]\!]$, $[\![\mathsf{com}]\!]$, or $[\![\mathsf{var}D]\!]$ are handled as in [64, pp. 94–95], obtaining a term M such that $\sigma' = [\![M]\!]$.
- If B_j is o, then $\sigma' = [\![\mathsf{raise}\, x_j M'_1 \ldots M'_l : B]\!]$.

We obtain the term defining σ by λ abstracting $x_{i+1}, \ldots, x_n$. ∎

Proposition 3.6.4 *If $\sigma : [\![\Gamma]\!] \Rightarrow [\![T]\!]$ is a compact, deterministic strategy satisfying visibility and bracketing, then σ is definable in IAA.*

Proof By Proposition 3.6.2, σ can be decomposed as the cell strategy composed with a compact, innocent strategy σ' for $([\![\Gamma]\!] \times [\![\mathsf{var}D]\!]) \Rightarrow [\![T]\!]$, definable in IAA$-$new by some $\Gamma, x : \mathsf{var}D \vdash M : T$. Then σ is definable by $\Gamma \vdash \mathsf{new}D\ x := 0\ \mathsf{in}\ M$. ∎

Theorem 3.6.5 (Definability) *If $\sigma : [\![\Gamma]\!] \Rightarrow [\![T]\!]$ is a compact strategy satisfying visibility and bracketing, then σ is definable in EIAA.*

Proof By Proposition 3.6.1, σ can be decomposed as the *oracle* strategy composed with a compact, deterministic strategy σ' for $([\![\Gamma]\!] \times [\![\mathsf{expint}]\!]) \Rightarrow [\![T]\!]$, definable in IAA by some $\Gamma, x : \mathsf{expint} \vdash M : T$. Then σ is definable by $\Gamma \vdash \lambda\, x : \mathsf{expint}.M\Theta$, where $\Theta = Y(\lambda\, y : \mathsf{expint}.0\ \mathsf{or}\ 1 + y)$. ∎

3.7 Full Abstraction

The full abstraction result will in fact hold not in the category **C**, but in the quotient of the game model described so far with respect to the *intrinsic*

preorder $\precsim$.

Let $\alpha : o \Rightarrow (A \Rightarrow [\![\mathsf{com}]\!])$ be tests on strategies for $o \Rightarrow A$. Given a strategy $\sigma : o \Rightarrow A$, we define

$$\sigma \bullet \alpha = \big(id_o, \sigma\big) \mathbin{;} \Lambda^{-1}(\alpha) : o \Rightarrow [\![com]\!]$$

A strategy $\sigma : o \Rightarrow A$ passes the test if $\sigma \bullet \alpha = \top^{done}$ or $\sigma \bullet \alpha = \top^{abort}$ where $\top^{done} = \{\varepsilon, run \cdot done\}$ and $\top^{abort} = \{\varepsilon, run \cdot abort\}$. The *intrinsic preorder* for strategies on $o \Rightarrow A$ is then defined as follows:

$$\begin{array}{l} \sigma \precsim \tau \text{ iff } \forall \alpha : o \Rightarrow (A \Rightarrow [\![\mathsf{com}]\!])\,. \\ \text{if } \sigma \bullet \alpha = \top^{done} \text{ then } \tau \bullet \alpha = \top^{done} \;\wedge\; \text{if } \sigma \bullet \alpha = \top^{abort} \text{ then } \tau \bullet \alpha = \top^{abort} \end{array}$$

So $\sigma \precsim \tau$ iff τ passes every test passed by σ.

Theorem 3.7.1 *[Full abstraction of EIAA] For any EIAA terms* $\Gamma \vdash M, N : T$*,*

$$[\![\Gamma \vdash M]\!] \precsim [\![\Gamma \vdash N]\!] \quad \text{iff} \quad \Gamma \vdash \mathrm{M} \sqsubsim \mathrm{N}$$

Proof Without loss of generality we can assume that M and N are terms of type T with only one global identifier $e : \mathbf{0}$.

Suppose $[\![M]\!] \precsim [\![N]\!]$ and $C[M]$ may terminate successfully (resp., abnormally) for some context $C[-] : \mathsf{com}$ where the hole is of type T. This context corresponds to some test $\alpha : o \Rightarrow ([\![T]\!] \Rightarrow [\![\mathsf{com}]\!])$ such that $[\![C[P]]\!] = [\![P]\!] \bullet \alpha$ for all suitably typed terms $e : \mathbf{0} \vdash P : T$. We have $[\![C[M]]\!] = [\![M]\!] \bullet \alpha = \top^{done}$ (resp., $\top^{abort}$), so since $[\![M]\!] \precsim [\![N]\!]$, we also have $[\![C[N]]\!] = [\![N]\!] \bullet \alpha = \top^{done}$ (resp., $\top^{abort}$). But $C[N]$ is computable, by Corollary 3.5.4 $C[N]$ may terminate successfully (resp., abnormally).

For the converse, we prove the contrapositive. Suppose $[\![M]\!] \not\precsim [\![N]\!]$. Then for some $\alpha : o \Rightarrow ([\![T]\!] \Rightarrow [\![\mathsf{com}]\!])$, $[\![M]\!] \bullet \alpha = \top^{done}$ (resp., $\top^{abort}$) and

$[\![N]\!] \bullet \alpha = \bot = \{\varepsilon\}$. By Theorem 3.6.5, $\alpha = [\![e : \mathbf{0}, x : T \vdash C[x] : \mathsf{com}]\!]$ for some term $C[x]$. We therefore have a context $C[-]$, such that $[\![C[M]]\!] = \top^{done}$ (resp., $\top^{abort}$) and $[\![C[N]]\!] = \bot$. By Corollary 3.5.4, $C[M]$ may terminate successfully (resp., abnormally) but not $C[N]$. Therefore, $M \not\sqsubseteq N$, as required. ∎

As a corollary we obtain the full abstraction result for AIA.

Corollary 3.7.2 (Full abstraction of AIA) *For any AIA terms $\Gamma \vdash M, N : T$,*

$$[\![\Gamma \vdash M]\!] \precsim [\![\Gamma \vdash N]\!] \quad \text{iff} \quad \Gamma \vdash \mathrm{M} \sqsubseteq \mathrm{N}$$

Proof It follows from Theorem 3.7.1, Proposition 3.2.1, and Proposition 3.2.2. ∎

Let us call a play *safe* if it does not terminate in *abort*, and a strategy *safe* if it consists only of safe plays. Otherwise, we will call plays and strategies *unsafe*. From the full abstraction result, it follows that:

Corollary 3.7.3 (Safety) *An AIA term $\Gamma \vdash M : T$ is safe iff $[\![\Gamma \vdash M : T]\!]$ is safe.*

Proof Without loss of generality we can assume that M is a closed term of type T.

Suppose $[\![M]\!]$ is a safe strategy and $C_{\text{safe}}[-] : \mathsf{com}$ is a safe context. This context corresponds to some safe strategy $\sigma : [\![T]\!] \Rightarrow [\![\mathsf{com}]\!]$, such that $[\![C_{\text{safe}}[M]]\!] = [\![M]\!] \bullet \sigma$. By Corollary 3.5.2, if $C_{\text{safe}}[M]$ may terminate abnormally then $run \cdot abort \in [\![C_{\text{safe}}[M]]\!]$. But σ and $[\![M]\!]$ are safe. Therefore, $C_{\text{safe}}[M]$ cannot terminate abnormally and $\Gamma \vdash M : T$ is safe.

The proof for the converse is similar. ∎

This result ensures that model checking a term's strategy for safety is equivalent to proving safety of the term.

3.8 Quotient Game Semantics

Given a base type expint_π or varint_π of AIA, we can quotient the arena and game for $[\![\mathsf{expint}]\!]$ or $[\![\mathsf{varint}]\!]$ (respectively) in a standard way, by replacing any integer n with its partition $\{m \mid m \approx_\pi n\}$. This extends compositionally to any type T of AIA: we can quotient the arena and game for $[\![\widetilde{T}]\!]$ by the abstractions in T. For any play t of the game for $[\![\widetilde{T}]\!]$, let $\overline{t}$ denote the image play of the quotient game, obtained by replacing each integer in t by its partition in the corresponding abstraction in T.

It is straightforward to check that, for any term $\Gamma \vdash M : T$ of AIA, and plays t and t' of the game for $[\![\widetilde{\Gamma} \vdash \widetilde{T}]\!]$, such that $\overline{t} = \overline{t'}$, we have

$$t \in [\![\Gamma \vdash M : T]\!] \Leftrightarrow t' \in [\![\Gamma \vdash M : T]\!]$$

Therefore, the quotient of the strategy $[\![\Gamma \vdash M : T]\!]$ by the abstractions in Γ and T loses no information.

Example Consider the term [3]

$$\vdash 1 +_{[0,1]} 1 : \mathsf{expint}_{[0,1]}$$

Its strategy and the quotient of its strategy are shown in Fig. 3.1 (a) and (b), respectively. ∎

Moreover, the quotient strategies can be defined compositionally. The strategy $\sigma \,;\, \tau : A \Rightarrow C$ is obtained by composing the strategies $\sigma : A \Rightarrow B_1$

[3]For simplicity, we sometimes write int_π as simply π.

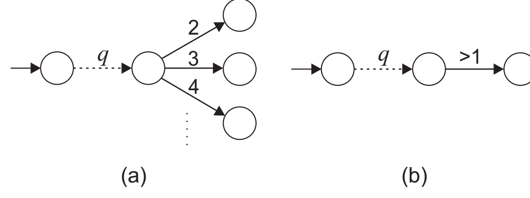


Figure 3.1: Strategies

and $\tau : B_2 \Rightarrow C$, where B_1 and B_2 are games for types which have equal concretisations. Since the abstractions in the types interpreted by B_1 and B_2 may be different, we need to allow a move which contains a partition c to interact with any move obtained by replacing c with some c' such that $c \cap c' \neq \emptyset$. Hence, even if the quotient strategies for σ and τ are deterministic, the one for $\sigma \fatsemi \tau$ may be nondeterministic.

In the rest of the thesis, for any term $\Gamma \vdash M : T$ of AIA, $[\![\Gamma \vdash M : T]\!]$ will denote its quotient game strategy.

Example Consider the quotient strategy

$$[\![x : \mathsf{varint}_{[0,4]} \vdash x := !x +_{[0,3]} 1_{[0,1]} : \mathsf{com}]\!]$$

If the value 3 is read from the variable x, the result of the addition is >3, because it belongs to the abstraction $[0,3]$. When >3 is assigned to x which is abstracted by $[0,4]$, it is nondeterministically converted to either 4 or >4. Thus, the following are two possible complete plays:

$$run\, read^x\, 3^x\, write(4)^x\, ok^x\, done, \quad run\, read^x\, 3^x\, write(>4)^x\, ok^x\, done \qquad \blacksquare$$

Chapter 4

The CSP Game Semantics Model

We have seen so far that game semantics gives fully abstract models for AIA (see Chapter 3). The full abstraction result means that the model validates all observable program properties, but unfortunately the model as described involves complex technical constructions, and calculating and reasoning within it is difficult. However, most of the complexity in the model is used to handle arbitrary higher-order functions. If the attention is restricted to the *second-order recursion-free fragment of AIA*, then the game semantics model can be significantly simplified. For this language fragment, programs define strategies for which justification pointers are uniquely determined by plays, and they can be disregarded. Thus, it has been shown in [48, 51] that the strategy of any second-order recursion-free IA term with finite data types can be represented by a *regular language*. This gives a decision procedure for a range of verification problems to be solved algorithmically, such as: program equivalences ($\cong$) and inequivalences, approximations ($\precsim$), assertions, invariants and other safety

properties.

In this chapter, we start by describing the second-order recursion-free fragment of AIA we are addressing. Then, we review the regular-language representation of the game model for the AIA fragment. We illustrate this model with several examples and show how it can be used for automatic verification. Next, we describe the representation of the game model using the CSP process algebra. For any term, we compositionally define a CSP process whose traces are exactly all the plays of the strategy for the term. A range of properties can then be decided by checking traces refinements between CSP processes.

Compared with the representation by regular languages [51, 10], the CSP representation brings several benefits:

- CSP operators preserve traces refinement (e.g. [98]), which means that a CSP process representing a term can be optimised and abstracted compositionally at the syntactic level (e.g. using process algebraic laws), and its set of traces will be preserved or enlarged.

- The ProBE and FDR tools [46] can be used to explore CSP processes visually, to check refinements automatically, and to debug interactively when a refinement does not hold.

- Compositional state-space reduction algorithms in FDR [97] enable smaller models to be generated before or during refinement checking.

- Composition of strategies, which is used in game semantics to obtain the strategy for a term from strategies for its subterms, is represented in CSP by renaming, parallel composition and hiding operators, and FDR is highly optimised for verification of such networks of processes.

- FDR has a special debugging feature which allows identification of hidden events only in counterexample traces rather than in full models. This "uncovering" feature of FDR is essential in efficiently implementing the abstraction refinement procedure (see Chapter 5).

We also present a prototype compiler which, given any term, outputs a CSP process representing its game semantics. The effectiveness of this approach is evaluated on two examples: a sorting algorithm, and an abstract data type implementation. The experimental results show that, for model generation, the CSP approach can outperform the approach which uses regular languages [10].

4.1 The Second-Order Language Fragment

In the rest of the thesis, we proceed to work with the *second-order recursion-free fragment of AIA*.

The *order* of types is defined by:

$$\mathsf{ord}(B) = 0 \qquad \mathsf{ord}(T \to T') = max\{\mathsf{ord}(T) + 1, \mathsf{ord}(T')\}$$

We say that a term is of *i-th order* if in its typing derivation the types of all free identifiers are of order less than i and the type of the term is of order at most i. The set of all i-th order AIA terms will be denoted by AIA_i. Hence, the second-order restriction means that the function types are restricted to

$$T ::= B \mid B \to T$$

We also add a new binding construct let to the language fragment. Without this construct, the fragment is only sufficient for reasoning about

program fragments, but not whole programs. Namely the fragment contains identifiers of function types, but no mechanism to bind them to actual functions.

The typing rule for let is:

$$\frac{\Gamma \vdash N : T \qquad \Gamma, x : T \vdash M : T'}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M : T'}$$

The operational semantics is given by:

$$\frac{\Gamma \vdash M[N/x], \mathrm{s} \Longrightarrow \mathcal{E}}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M, \mathrm{s} \Longrightarrow \mathcal{E}} \qquad \frac{\Gamma \vdash M[N/x], \mathrm{s} \Longrightarrow V, \mathrm{s}'}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M, \mathrm{s} \Longrightarrow V, \mathrm{s}'}$$

Programs involving the let construct can be reduced to programs without let by beta reduction. For the majority of the rest of this thesis, we will consider this construct. But if the use of the let construct causes some additional difficulties in the presentation, which cannot be handled in an uniform and consistent way, we will restrict ourselves to terms in β-normal form, i.e. to the let-free fragment.

In addition to the AIA_2 fragment, we will also consider IA_2 with bounded integers (or finitary IA_2). The reason we consider this variant of IA is that the regular-language representation [51] was originally developed for it and subsequently a model-checking tool using finite-state machines [10] was implemented for it. So, by working with terms of this language fragment we can compare the efficiency of tools based on CSP and regular-languages (see Section 4.3.5).

The finitary IA_2 has bounded integers and booleans as basic data types,

$$D ::= \mathsf{int}_n \mid \mathsf{bool}$$

where $\mathsf{int}_n = \{0, \ldots, n-1\}$. In order to implement arithmetic-logic operations, we use congruence arithmetic modulo n. For example, the expression $1+2$ evaluates to 3 if its type is int_4, or to 0 for int_3. All integer free identifiers and local

variables in a term are annotated by an abstraction, and the type annotations of every arithmetic operation and constant are inferred using the following rules. An integer constant n is implicitly of type int_{n+1}. An operation between values of types int_{n_1} and int_{n_2} produces a value of type $\mathsf{int}_{max\{n_1,n_2\}}$. The operation is performed modulo $max\{n_1, n_2\}$. This data-abstraction scheme is very informal and the obtained abstracted programs are not always conservative over-approximations of their concrete counterparts, i.e. safety of an abstracted program does not imply safety of the concrete program. However, the abstractions can be useful in detecting many errors.

Actually, AIA can be considered as an extension of IA with bounded integers. In Chapter 5, we show that data-abstractions used in AIA are conservative over-approximations and safety of any finitely abstracted program implies safety of the corresponding concrete program.

4.1.1 Syntactic Sugar

Arrays of length $k > 0$ can be introduced in the language by using the existing term formers. They do not contribute semantically, being only what is called syntactic sugar. They can be expressed by the following abbreviations:

$$\mathsf{new}_D\ x[k] := v\ \mathsf{in}\ M \ \equiv$$
$$\mathsf{new}_D\ x[0] := v\ \mathsf{in}$$
$$\ldots$$
$$\mathsf{new}_D\ x[k-1] := v\ \mathsf{in}\ M$$
$$x[E] \ \equiv$$
$$\mathsf{if}\ E = 0\ \mathsf{then}\ x[0]\ \mathsf{else}$$
$$\ldots$$
$$\mathsf{if}\ E = k-1\ \mathsf{then}\ x[k-1]\ \mathsf{else\ abort}$$

The command abort is executed whenever an array out-of-bounds error occurs, i.e. there is an attempt to access elements out of the bounds of an array.

4.2 Regular-language Representation

We now give an overview of the regular-language representation of the game semantics model for recursion-free AIA_2. More technical details of the representation can be found in [48, 51].

In this setting, types are represented as *alphabets* of moves, plays of a game as *words* (sequences) over an alphabet, and strategies as *regular-languages* over an alphabet. The languages, denoted by $\mathcal{L}(R)$, are specified using *extended regular expressions* R. They are defined inductively over finite alphabets $\mathcal{A}$ using the following operations:

$$\emptyset \quad \varepsilon \quad a \quad R \cdot R' \quad R^* \quad R + R' \quad R \cap R'$$
$$R \mid_{\mathcal{A}'} \quad R[R'/w] \quad R^{\langle a \rangle} \quad R' \,{}^{\circ}_{9}{}_{\mathcal{B}}\, R \quad R \bowtie R' \quad \widetilde{R}$$

where R, R' ranges over extended regular expressions, $\mathcal{A}, \mathcal{B}$ over finite alphabets, $a \in \mathcal{A}$, $\mathcal{A}' \subseteq \mathcal{A}$ and $w \in \mathcal{A}^*$.

Constants $\emptyset$, ε and a denote the languages $\emptyset$, $\{\varepsilon\}$ and $\{a\}$, respectively. Concatenation $R \cdot R'$, Kleene star R^*, union $R + R'$ and intersection $R \cap R'$ are the standard operations. Restriction $R \mid_{\mathcal{A}'}$ removes all symbols in $\mathcal{A}'$ from all sequences in the language of R. Substitution $R[R'/w]$ is the language of R where all occurrences of the subword w have been replaced by the sequences of R'. Given two symbols $a,b \in \mathcal{A}$, $b^{\langle a \rangle}$ is a new symbol obtained by tagging the latter with the former. If a symbol is tagged more than once, we write: $(b^{\langle a_1 \rangle})^{\langle a_2 \rangle} = b^{\langle a_1, a_2 \rangle}$. We define the alphabet $\mathcal{A}^{\langle a \rangle} = \{b^{\langle a \rangle} \mid b \in \mathcal{A}\}$ and the regular expression $R^{\langle a \rangle}$ as the language obtained by tagging of all symbols in

the sequences of R with a. Composition $R' \mathbin{;}_{\mathcal{B}} R$ is defined as follows:

$$R' \mathbin{;}_{\mathcal{B}} R = \begin{cases} \{w[s/a \cdot b] \mid w \in R\} & \text{if } a \cdot s \cdot b \in R' \\ \{w[s \cdot abort/w_{\geq a}] \mid w \in R\} & \text{if } a \cdot s \cdot abort \in R' \end{cases}$$

where R' is a set of words of form $a \cdot s \cdot b$ or $a \cdot s \cdot abort$, $a, b \in \mathcal{B}$, s does not contain symbols from $\mathcal{B}$, and $w_{\geq a}$ is the suffix of w starting from and including a. The shuffle operation of two regular languages is defined as $\mathcal{L}(R) \bowtie \mathcal{L}(R') = \bigcup_{w_1 \in \mathcal{L}(R), w_2 \in \mathcal{L}(R')} w_1 \bowtie w_2$, where $w \bowtie \varepsilon = \varepsilon \bowtie w = w$ and $a \cdot w_1 \bowtie b \cdot w_2 = a \cdot (w_1 \bowtie b \cdot w_2) + b \cdot (a \cdot w_1 \bowtie w_2)$. The effective alphabet $\lfloor R \rfloor$ of a regular expression R is the set of all symbols appearing in the language denoted by that regular expression, i.e. $\lfloor R \rfloor = \{a \in \mathcal{A} \mid a \in w \text{ for some } w \in \mathcal{L}(R)\}$. Then, the broadening operation $\widetilde{R}$ shuffles the regular expression R with all sequences not in its effective alphabet, i.e. $\widetilde{R} = R \bowtie (\mathcal{A} \setminus \lfloor R \rfloor)^*$. It is a standard result that any extended regular expression obtained from the operations above denotes a regular language [51, pp. 11–12], which can be recognised by a finite automaton [73].

Each type T is interpreted by an alphabet $\mathcal{A}_{[\![T]\!]}$ of moves. For each type we also define alphabets of questions $\mathrm{Q}_{[\![T]\!]}$ and answers $\mathrm{A}^{\mathsf{q}}_{[\![T]\!]}$ for each $\mathsf{q} \in \mathrm{Q}_{[\![T]\!]}$.

$$\mathcal{A}_{[\![\mathsf{int}_{[n,m]}]\!]} = \{< n, n, n+1, \ldots, 0, \ldots, m-1, m, > m\} \qquad \mathcal{A}_{[\![\mathsf{bool}]\!]} = \{tt, f\!f\}$$

$$\mathrm{Q}_{[\![\mathsf{exp}D]\!]} = \{q\} \qquad \mathrm{A}^{q}_{[\![\mathsf{exp}D]\!]} = \mathcal{A}_{[\![D]\!]}$$

$$\mathrm{Q}_{[\![\mathsf{com}]\!]} = \{run\} \qquad \mathrm{A}^{run}_{[\![\mathsf{com}]\!]} = \{done\}$$

$$\mathrm{Q}_{[\![\mathsf{var}D]\!]} = \{read, write(a) \mid a \in \mathcal{A}_{[\![D]\!]}\} \qquad \mathrm{A}^{read}_{[\![\mathsf{var}D]\!]} = \mathcal{A}_{[\![D]\!]} \qquad \mathrm{A}^{write(a)}_{[\![\mathsf{var}D]\!]} = \{ok\}$$

$$\mathrm{Q}_{[\![B_1\times\ldots\times B_k\to B]\!]} = \sum_{1\le i\le k} \mathrm{Q}^{\langle i\rangle}_{[\![B_i]\!]} \cup \mathrm{Q}_{[\![B]\!]}$$

$$\mathrm{A}^{\mathsf{q}^{\langle i\rangle}}_{[\![B_1\times\ldots\times B_k\to B]\!]} = \big(\mathrm{A}^{\mathsf{q}}_{[\![B_i]\!]}\big)^{\langle i\rangle},\ \mathsf{q}\in \mathrm{Q}_{[\![B_i]\!]},\ 1\le i\le k$$

$$\mathrm{A}^{\mathsf{q}}_{[\![B_1\times\ldots\times B_k\to B]\!]} = \mathrm{A}^{\mathsf{q}}_{[\![B]\!]},\ \mathsf{q}\in \mathrm{Q}_{[\![B]\!]}$$

$$\mathcal{A}_{[\![T]\!]} = \mathrm{Q}_{[\![T]\!]} \cup \bigcup_{\mathsf{q}\in \mathrm{Q}_{[\![T]\!]}} \mathrm{A}^{\mathsf{q}}_{[\![T]\!]}$$

We use meta-variables q to range over symbols which are questions and a over symbols which are answers.

Each term is interpreted using an evaluation function $[\![-]\!]^R$ mapping a term $\Gamma \vdash M : T$ and an environment u into a regular language defined over the alphabet:

$$\mathcal{A}_{[\![\Gamma\vdash T]\!]} = \Big(\sum_{x_i:T_i\in\Gamma} \mathcal{A}^{\langle x_i\rangle}_{[\![T_i]\!]}\Big) + \mathcal{A}_{[\![T]\!]} + \{abort\}$$

The environment u maps free identifiers $x : T \in \Gamma$ to the copy-cat regular languages K^x_T defined as:

$$K^x_{B_1\times\ldots\times B_k\to B} = \sum_{\mathsf{q}\in \mathsf{Q}_{[\![B]\!]}} \mathsf{q}\cdot\mathsf{q}^{\langle x\rangle}\cdot\Big(\sum_{1\le i\le k} R^{x,i}_{B_i}\Big)^*\cdot \sum_{\mathsf{a}\in \mathsf{A}^{\mathsf{q}}_{[\![B]\!]}} \mathsf{a}^{\langle x\rangle}\cdot\mathsf{a}$$

where $R^{x,i}_B = \sum_{\mathsf{q}\in\mathsf{Q}_{[\![B]\!]}} \mathsf{q}^{\langle x,i\rangle}\cdot\mathsf{q}^{\langle i\rangle}\cdot\sum_{\mathsf{a}\in\mathsf{A}^{\mathsf{q}}_{[\![B]\!]}} \mathsf{a}^{\langle i\rangle}\cdot\mathsf{a}^{\langle x,i\rangle}$.

In Table 4.1 AIA_2 terms are interpreted by regular expressions describing their sets of complete plays. All other plays are even-length prefixes of the complete ones. The representation of language constructs ‘c’ is given in Table 4.2.

Functions $\mathsf{op}_{D_1\times D_2\to D}$, which implement arithmetic-logic operations, and the function $\mathsf{cast}_{B',B}$, which converts a value of type B' into type B, are given in Section 4.3, which describes CSP representation for AIA_2. Since they are equivalent in both representations, we do not reproduce them here.

$$\llbracket \Gamma \vdash v : \mathsf{exp}D \rrbracket^R u = q \cdot v$$
$$\llbracket \Gamma \vdash \mathsf{skip} : \mathsf{com} \rrbracket^R u = run \cdot done$$
$$\llbracket \Gamma \vdash \mathsf{abort} : \mathsf{com} \rrbracket^R u = run \cdot abort$$
$$\llbracket \Gamma \vdash \mathsf{c}(M_1, \ldots, M_k) : B' \rrbracket^R u = \llbracket \Gamma \vdash M_1 : B_1^{\langle 1 \rangle} \rrbracket^R u \mathbin{;}_{\mathcal{A}^{\langle 1 \rangle}_{\llbracket B_1 \rrbracket}} \cdots$$
$$\cdots \llbracket \Gamma \vdash M_k : B_k^{\langle k \rangle} \rrbracket^R u \mathbin{;}_{\mathcal{A}^{\langle k \rangle}_{\llbracket B_k \rrbracket}} \llbracket \mathsf{c} : B_1^{\langle 1 \rangle} \times \ldots B_k^{\langle k \rangle} \to B' \rrbracket^R u$$
$$\llbracket \Gamma \vdash MN : T \rrbracket^R u = \llbracket \Gamma \vdash N : B \rrbracket^R u \mathbin{;}_{\mathcal{A}_{\llbracket B \rrbracket}} \llbracket \Gamma \vdash M : B \to T \rrbracket^R u$$
$$\llbracket \Gamma, x : T \vdash x : T \rrbracket^R u = u(x)$$
$$\llbracket \mathsf{new}_D\, x := v \,\mathsf{in}\, M : B \rrbracket^R u = \big(\widetilde{\gamma}^x_v \cap \llbracket \Gamma, x : \mathsf{var}D \vdash M \rrbracket^R (u \mid x \mapsto K^x_{\mathsf{var}D}) \big)\big|_{\mathcal{A}^{\langle x \rangle}_{\llbracket \mathsf{var}D \rrbracket}}$$
$$\gamma^x_v = (read^{\langle x \rangle} \cdot v^{\langle x \rangle})^* \cdot \Big(\sum_{a \in \mathcal{A}_{\llbracket D \rrbracket}} write(a)^{\langle x \rangle} \cdot ok^{\langle x \rangle} \cdot (read^{\langle x \rangle} \cdot a^{\langle x \rangle})^* \Big)^*$$
$$\llbracket \Gamma \vdash \mathsf{let}\, x \,\mathsf{be}\, N \,\mathsf{in}\, M : T' \rrbracket^R u = \llbracket \Gamma, x : T \vdash M : T' \rrbracket^R (u \mid x \mapsto \llbracket \Gamma \vdash N : T \rrbracket^R u)$$

Table 4.1: Regular-language representation

In [51, pp. 28–32], it was shown the correctness of regular-language representation for finitary IA_2. We now prove a similar result for AIA_2 by showing that the regular-language representation is isomorphic to the game semantics model (see Chapter 3).

We start by using the fact that for the types considered in a second-order language fragment, the justification pointers in the game models can be ignored [51, pp. 27–28]. Then, for any term $x_1 : T_1, \ldots, x_k : T_k \vdash M : B_1 \times \ldots \times B_l \to B$ of AIA_2, we define an *isomorphism* ρ as:

- the tagging of all moves in the game model of T_j with $\langle x_j \rangle$ and the tagging with $\langle x_j, i \rangle$ of all moves in $B_{j,i}$ where $T_j = B_{j,1} \times \ldots \times B_{j,k_j} \to B'_j$.

- the tagging with $\langle j \rangle$ of all moves in B_j.

We now show that the regular-language representation of a term is ρ-isomorphic to the strategy for that term where all justification pointers are deleted.

$$[\![\mathsf{op}_D : \mathsf{exp}D_1^{\langle 1\rangle} \times \mathsf{exp}D_2^{\langle 2\rangle} \to \mathsf{exp}D]\!]^R u = \sum_{\substack{a_1 \in \mathcal{A}_{[\![D_1]\!]}, a_2 \in \mathcal{A}_{[\![D_2]\!]} \\ a = \mathsf{op}_{D_1 \times D_2 \to D}(a_1, a_2)}} q \cdot q^{\langle 1\rangle} \cdot a_1^{\langle 1\rangle} \cdot q^{\langle 2\rangle} \cdot a_2^{\langle 2\rangle} \cdot a$$

$$[\![; : \mathsf{com}^{\langle 1\rangle} \times B^{\langle 2\rangle} \to B]\!]^R u = \sum_{\mathsf{q} \in \mathrm{Q}_{[\![B]\!]}, \mathsf{a} \in \mathrm{A}_{[\![B]\!]}} \mathsf{q} \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot \mathsf{q}^{\langle 2\rangle} \cdot \mathsf{a}^{\langle 2\rangle} \cdot \mathsf{a}$$

$$[\![\mathsf{if}_B : \mathsf{expbool}^{\langle 1\rangle} \times B_1^{\langle 2\rangle} \times B_2^{\langle 3\rangle} \to B]\!]^R u = \sum_{\substack{\mathsf{q} \in \mathrm{Q}_{[\![B]\!]}, \mathsf{a} \in \mathrm{A}^{\mathsf{q}}_{[\![B_1]\!]} \\ \mathsf{b} \in \mathsf{cast}_{B_1, B}(\mathsf{a})}} \mathsf{q} \cdot q^{\langle 1\rangle} \cdot tt^{\langle 1\rangle} \cdot \mathsf{q}^{\langle 2\rangle} \cdot \mathsf{a}^{\langle 2\rangle} \cdot \mathsf{b} + \sum_{\substack{\mathsf{q} \in \mathrm{Q}_{[\![B]\!]}, \mathsf{a} \in \mathrm{A}^{\mathsf{q}}_{[\![B_2]\!]} \\ \mathsf{b} \in \mathsf{cast}_{B_2, B}(\mathsf{a})}} \mathsf{q} \cdot q^{\langle 1\rangle} \cdot f\!f^{\langle 1\rangle} \cdot \mathsf{q}^{\langle 3\rangle} \cdot \mathsf{a}^{\langle 3\rangle} \cdot \mathsf{b}$$

$$[\![\mathsf{while} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \to \mathsf{com}]\!]^R u = run \cdot \left(q^{\langle 1\rangle} \cdot tt^{\langle 1\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle}\right)^* \cdot q^{\langle 1\rangle} \cdot f\!f^{\langle 1\rangle} \cdot done$$

$$[\![:= : \mathsf{var}D_1^{\langle 1\rangle} \times \mathsf{exp}D_2^{\langle 2\rangle} \to \mathsf{com}]\!]^R u = \sum_{\substack{a \in \mathcal{A}_{[\![D_2]\!]} \\ b \in \mathsf{cast}_{\mathsf{exp}D_2, \mathsf{exp}D_1}(a)}} run \cdot q^{\langle 2\rangle} \cdot a^{\langle 2\rangle} \cdot write(b)^{\langle 2\rangle} \cdot ok^{\langle 2\rangle} \cdot done$$

$$[\![! : \mathsf{var}D^{\langle 1\rangle} \to \mathsf{exp}D]\!]^R u = \sum_{a \in \mathcal{A}_{[\![D]\!]}} q \cdot read^{\langle 1\rangle} \cdot a^{\langle 1\rangle} \cdot a$$

Table 4.2: Representation of language constructs

Lemma 4.2.1 *For any AIA_2 term*

$$\mathsf{closure}([\![\Gamma \vdash M : T]\!]^R u) \stackrel{\rho}{\equiv} [\![\Gamma \vdash M : T]\!]$$

where u is an environment mapping all free identifiers $x : T' \in \Gamma$ to copy-cat regular languages $K^x_{T'}$, $\mathsf{closure}(R)$ denotes the regular language containing even-length prefixes of all sequences in R, and $[\![\Gamma \vdash M : T]\!]$ denotes the set of all plays in the strategy for M obtained as in Chapter 3 where justification pointers are deleted.

Proof The proof is by induction on the derivation of $\Gamma \vdash M : T$ and it is a

straightforward extension of the similar proof for finitary IA_2 [51, pp. 28–32].

For language constants and constructs, their sets of complete plays are finite so we can check by inspection that their regular-language and games interpretations are ρ-isomorphic. The correctness of all other term formers was proved in [51].

So, the proof follows by inductive hypothesis and the correctness of the definition of composition. ∎

This Lemma shows that the regular-language semantics is a correct representation of the game semantics, and as a corollary we obtain the full abstraction theorem for this representation.

Theorem 4.2.2 *For any two terms of AIA_2*

$$\Gamma \vdash M \cong N \quad \textit{iff} \quad \mathsf{closure}([\![\Gamma \vdash M : T]\!]^R u) = \mathsf{closure}([\![\Gamma \vdash N : T]\!]^R u)$$

Since equality of regular languages is decidable, it follows from Lemma 4.2.1 and Theorem 4.2.2 that:

Corollary 4.2.3 (Decidability) *Observational safe-equivalence of finitely abstracted terms of AIA_2 is decidable by equality checks of regular languages.*

Example Consider the term

$$f : \mathsf{com} \times \mathsf{com} \to \mathsf{com} \vdash \mathsf{new}_{\mathsf{int}[0,1]}\, x := 0 \,\mathsf{in}$$
$$f\big(x := !x +_{[0,2]} 1,\ \mathsf{if}\ (!x >_{\mathsf{bool}} 1)\ \mathsf{then\ abort}\big)$$

in which f is a non-local procedure.

The strategy for this term represented as a finite automaton is shown in Figure 4.1. The dashed edges indicate moves of the Opponent and solid edges moves of the Player. They serve only as a visual aid to the reader. Accepting

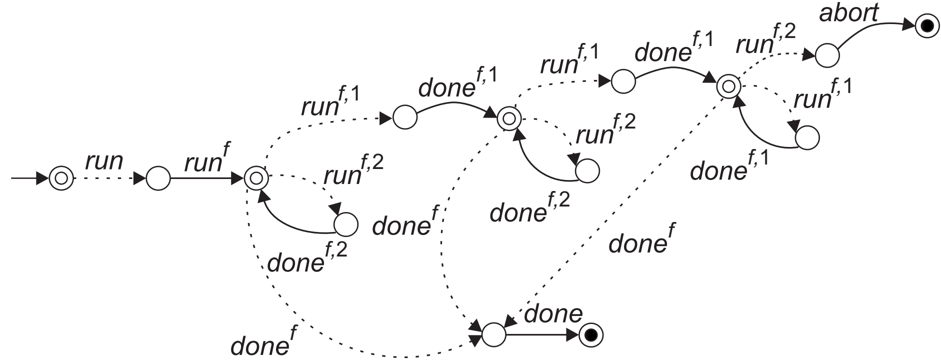


Figure 4.1: A strategy as a finite automaton

states are designated by an interior circle. The states whose interior circles are filled in, correspond to complete plays in the strategy. Justification pointers are ignored since they can be uniquely determined by plays. The model illustrates only the possible behaviors of this term: if the non-local procedure f calls its first argument, two or more times, and afterwards its second argument then the term terminates abnormally; otherwise the term terminates successfully. The model does not assume that f uses its arguments, or how many times or in what order. Notice that no references to the variable x appear in the model because it is locally defined and so not visible from the outside of the term. ∎

4.3 CSP Representation

The main contribution of this chapter is presented here. We show how the game semantics model of AIA_2 can be represented using the CSP process algebra. The section starts by giving a brief introduction to CSP.

4.3.1 Background: CSP

CSP (Communicating Sequential Processes) [70] is a language for modelling interacting components. Each component is specified through its behaviour which is given as a process. Here we introduce only the CSP notation and the ideas used in this thesis. For a fuller introduction to the language the reader is referred to [98].

CSP processes are defined in terms of the *events* that they can perform. The choice of alphabets, i.e. sets of events that processes might use, is one of the most important modelling decision that is made when representing a real system in CSP. The set of all possible events is denoted Σ. Events may be atomic in structure, such as c and *run*, or may consist of a channel name plus a finite number of 'data' components, such as $c.q$ and *write*.1, and are of the form

$$c.x_1. \dots .x_n$$

where c is an identifier (channel name), $T_1, \dots, T_n$ is a finite sequence of types, and $x_i \in T_i$ for each $1 \leq i \leq n$. We can define the set of all events that can arise on any set of channels and partially defined events as follows. If c is a channel with type $T_1. \dots .T_n$, $0 \leq k \leq n$ and $a_i \in T_i$ for $1 \leq i \leq k$, then

$$\{\mid c.a_1. \dots .a_k \mid\} = \{c.a_1. \dots .a_k.b_{k+1}. \dots .b_n \mid b_{k+1} \in T_{k+1}, \dots, b_n \in T_n\}$$

is the set of events which can be formed as extensions of $c.a_1. \dots .a_k$.

Then, we can define the set of all events that can arise on channels $c_1, \dots, c_k$ as:

$$\{\mid c_1, \dots, c_l \mid\} = \{\mid c_1 \mid\} \cup \dots \cup \{\mid c_l \mid\}$$

We will also use the following notation: $c.T_1. \dots .T_n = \{\mid c \mid\}$

The following collection of process operators will be used:

$$\begin{aligned} P \;\; ::= \;\; & p \;\mid\; STOP \;\mid\; SKIP \;\mid\; RUN_A \;\mid ?x : A \to P(x) \\ & \mid\; \mu\, p \centerdot P \;\mid\; P_1 \,\Box\, P_2 \;\mid\; \textsf{if } b \textsf{ then } P_1 \textsf{ else } P_2 \;\mid\; P_1 \underset{A}{\parallel} P_2 \\ & \mid\; P \setminus A \;\mid\; P\,[a/b] \;\mid\; P_1 \,;\, P_2 \end{aligned}$$

where $A \subseteq \Sigma$ represents a set of events, P a process expression and p is a process name (or identifier).

The simplest process is $STOP$ which performs no events. $SKIP$ is a process that successfully terminates causing the special event ✓ ('tick'). The event ✓ is not a member of Σ, emphasising that it is special. $\Sigma^{\checkmark}$ will denote the extended alphabet $\Sigma \cup \{\checkmark\}$. RUN_A can always perform any event from A. If for each $x \in A$, there exists a process $P(x)$, then the prefix choice process, $?x : A \to P(x)$, can perform any event a from the set A and then behaves as the appropriate $P(a)$. For example, $?x : \{\} \to P(x)$ is equivalent to $STOP$. We can write $?x : \{a\} \to P(x)$ as $a \to P(a)$.

To define a process recursively by $p = P$, where P is any CSP process involving p, we write $\mu\, p \centerdot P$. We consider only guarded recursions, where each recursive call p is prefixed by an event in P. For example, RUN_A and $\mu\, p \centerdot ?x : A \to p$ are equivalent. The external choice operator defines a process $P_1 \,\Box\, P_2$ which can behave either as P_1 or as P_2. Conditional $\textsf{if } b \textsf{ then } P_1 \textsf{ else } P_2$, where b is a boolean test, behaves as P_1 if b is true, or as P_2 if b is false. A parallel composition $P_1 \underset{A}{\parallel} P_2$ runs P_1 and P_2 in parallel, making them synchronise on events in A and allowing interleaving of all other events. It terminates successfully if and only if both component processes do so. A process $P \setminus A$ behaves as P except that all events from A become hidden, i.e. they are transformed into invisible or internal events τ ('tau'). The event τ is also a special event and it is not in Σ. To rename an event or channel b to a in

a process P, we write $P[a/b]$. A sequential composition $P_1 \mathbin{;} P_2$ runs P_1, and if it terminates successfully, runs P_2. For example, for all P, $SKIP \mathbin{;} P = P$ but $STOP \mathbin{;} P = STOP$.

CSP processes can be given denotational semantics by their sets of traces. A trace is a finite sequence of events. A sequence t is a *trace* of a process P if there is some execution of P in which exactly that sequence of events is performed. Invisible events τ are not recorded in traces. For a process P, we define $\mathsf{traces}(P)$ to be the set of all its traces. The rules for calculating the traces sets of all CSP constructs are given below.

$$
\begin{array}{lcl}
\mathsf{traces}(STOP) & = & \{\varepsilon\} \\
\mathsf{traces}(SKIP) & = & \{\varepsilon, \checkmark\} \\
\mathsf{traces}(?x : A \rightarrow P(x)) & = & \{\varepsilon\} \cup \{a \cdot s \mid a \in A, s \in \mathsf{traces}(P(a))\} \\
\mathsf{traces}(\mu\, p \bullet P) & = & \bigsqcup_{i \in \mathbb{N}} \mathsf{traces}(F_i), \text{for } F_0 = STOP, F_{i+1} = P[F_i/p] \\
\mathsf{traces}(P_1 \;\Box\; P_2) & = & \mathsf{traces}(P_1) \cup \mathsf{traces}(P_2) \\
\mathsf{traces}(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2) & = & \mathsf{traces}(P_1) \text{ if } b \text{ evaluates to } tt; \\
 & & \text{ and } \mathsf{traces}(P_2) \text{ if } b \text{ evaluates to } f\!f \\
\mathsf{traces}(P_1 \underset{A}{\parallel} P_2) & = & \bigcup\{s \underset{A}{\parallel} t \mid s \in \mathsf{traces}(P_1), t \in \mathsf{traces}(P_2)\} \\
\mathsf{traces}(P \backslash A) & = & \{s \upharpoonright (\Sigma \backslash A) \mid s \in \mathsf{traces}(P)\} \\
\mathsf{traces}(P_1 \mathbin{;} P_2) & = & (\mathsf{traces}(P_1) \cap \Sigma^*) \\
 & & \cup\ \{s \cdot t \mid s \cdot \checkmark \in \mathsf{traces}(P_1), t \in \mathsf{traces}(P_2)\}
\end{array}
$$

where the notation $P[F_i/p]$ means the substitution of the process F_i for all free occurrences of the process identifier p in P. For $a, a' \in A$ and $b, b' \in \Sigma \backslash A$,

we define a set of traces $s \underset{A}{\parallel} t$ for all $s, t \in \Sigma^*$ as follows:

$$s \underset{A}{\parallel} t = t \underset{A}{\parallel} s \qquad s \cdot \checkmark \underset{A}{\parallel} t \cdot \checkmark = \{u \cdot \checkmark \mid u \in s \underset{A}{\parallel} t\}$$
$$\varepsilon \underset{A}{\parallel} \checkmark = \{\} \qquad \varepsilon \underset{A}{\parallel} \varepsilon = \{\varepsilon\} \qquad \varepsilon \underset{A}{\parallel} a = \{\} \qquad \varepsilon \underset{A}{\parallel} b = \{b\}$$
$$a \cdot s \underset{A}{\parallel} b \cdot t = \{b \cdot u \mid u \in a \cdot s \underset{A}{\parallel} t\} \qquad a \cdot s \underset{A}{\parallel} a \cdot t = \{a \cdot u \mid u \in s \underset{A}{\parallel} t\}$$
$$a \cdot s \underset{A}{\parallel} a' \cdot t = \{\}, \text{ if } a \neq a'$$
$$b \cdot s \underset{A}{\parallel} b' \cdot t = \{b \cdot u \mid u \in s \underset{A}{\parallel} b' \cdot t\} \cup \{b' \cdot u \mid u \in b \cdot s \underset{A}{\parallel} t\}$$

Note that for any process P, $\mathsf{traces}(P)$ is a non-empty and prefix-closed set.

Let $\mathsf{traces}^{\mathsf{ev}}(P)$ be the set of all even-length traces of P:

$$\mathsf{traces}^{\mathsf{ev}}(P) = \{t \mid \mathit{even_length}(t) \text{ and } t \in \mathsf{traces}(P)\}$$

A process P_2 is a *traces refinement* of P_1 if and only if any trace of P_2 is also a trace of P_1:

$$P_1 \sqsubseteq_T P_2 \;\Leftrightarrow\; \mathsf{traces}(P_2) \subseteq \mathsf{traces}(P_1)$$

Example Consider the process

$$P = \mu\, p \centerdot (a \to p) \;\Box\; (b \to \mathit{SKIP})$$

Its traces set is $\mathsf{traces}(P) = \{a^n, a^n b, a^n b \checkmark \mid n \in \mathbb{N}\}$. Then, we have that $\mathit{RUN}^{\checkmark}_{\{a,b\}} \sqsubseteq_T P$, where $\mathit{RUN}^{\checkmark}_A = \mu\, p \centerdot \mathit{SKIP} \;\Box\; (?x : A \to p)$. ∎

CSP processes can also be given operational semantics using *labelled transition systems* (LTS). The LTS of a process is a directed graph whose nodes represent process states and whose edges are labelled by events representing what happens when the given event is performed. LTSs have a distinguished start state, and any edge whose label is $\checkmark$ leads to a special terminated state Ω.

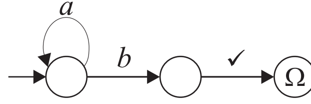


Figure 4.2: A labelled transition system

Example The LTS of the process P from the previous example is shown in Figure 4.2. ▮

The FDR (stands for Failures/Divergences Refinement) tool [46] is a refinement checker for CSP processes. It supports *hierarchical compression* and contains several procedures for compositional state-space reduction. FDR builds up a system gradually, at each stage compressing the subsystems to find an equivalent process with many fewer states. The main result which enables this technique is that the traces (denotational) semantics of CSP is a congruence with respect to the operational semantics [98, Chapter 9]. Hence, before generating a LTS for a context $C[P_1, P_2, \ldots, P_n]$, LTSs of its component processes P_i can be reduced, while preserving semantics of the complete context [97].

FDR is also highly optimised for checking refinements by processes which consist of a number of component processes composed by operators such as renaming, parallel composition and hiding. Namely, FDR uses a two-level approach to calculate LTSs of processes. The low-level is fully general but relatively inefficient, while the high-level is restricted but much more efficient. Low-level processes are fully evaluated using the low-level compiler, which turns them into explicit LTSs (i.e. a list of states and transitions). High-level processes, which are combinations of low-level component processes using ‘high-level’ operators (renaming, parallel composition and hiding), are compiled into implicit (symbolic) LTSs, which consist of efficient rules for com-

puting the initial transitions (events) and next states of any combination of states of low-level components that might arise. Thus a process, which is a combination of low-level components using high-level operators, will be explored much more efficiently by FDR than an equivalent one entirely compiled at the low-level.

4.3.2 Representation

With each type T, we associate a set of possible events: an alphabet $\mathcal{A}_{[\![T]\!]}$. It contains events $\mathsf{q} \in \mathrm{Q}_{[\![T]\!]}$ called questions, which are appended to a channel with name Q, and for each question q, there is a set of events $\mathsf{a} \in \mathrm{A}^{\mathsf{q}}_{[\![T]\!]}$ called answers, which are appended to a channel with name A.

$$\mathcal{A}_{[\![\mathsf{int}_{[n,m]}]\!]} = \{< n, n, n+1, \ldots, 0, \ldots, m-1, m, > m\} \qquad \mathcal{A}_{[\![\mathsf{bool}]\!]} = \{tt, f\!f\}$$

$$\mathrm{Q}_{[\![\mathsf{exp}D]\!]} = \{q\} \qquad \mathrm{A}^{q}_{[\![\mathsf{exp}D]\!]} = \mathcal{A}_{[\![D]\!]}$$

$$\mathrm{Q}_{[\![\mathsf{com}]\!]} = \{run\} \qquad \mathrm{A}^{run}_{[\![\mathsf{com}]\!]} = \{done\}$$

$$\mathrm{Q}_{[\![\mathsf{var}D]\!]} = \{read, write.v \mid v \in \mathcal{A}_{[\![D]\!]}\} \qquad \mathrm{A}^{read}_{[\![\mathsf{var}D]\!]} = \mathcal{A}_{[\![D]\!]} \qquad \mathrm{A}^{write.v}_{[\![\mathsf{var}D]\!]} = \{ok\}$$

$$\mathrm{Q}_{[\![B_1 \times \ldots \times B_k \to B]\!]} = \bigcup_{1 \le i \le k} \{i.\mathsf{q} \mid \mathsf{q} \in \mathrm{Q}_{[\![B_i]\!]}\} \cup \mathrm{Q}_{[\![B]\!]}$$

$$\mathrm{A}^{i.\mathsf{q}}_{[\![B_1 \times \ldots \times B_k \to B]\!]} = \{i.\mathsf{a} \mid \mathsf{a} \in \mathrm{A}^{\mathsf{q}}_{[\![B_i]\!]}\},\ \mathsf{q} \in \mathrm{Q}_{[\![B_i]\!]},\ 1 \le i \le k$$

$$\mathrm{A}^{\mathsf{q}}_{[\![B_1 \times \ldots \times B_k \to B]\!]} = \mathrm{A}^{\mathsf{q}}_{[\![B]\!]},\ \mathsf{q} \in \mathrm{Q}_{[\![B]\!]}$$

$$\mathcal{A}_{[\![T]\!]} = Q.\mathrm{Q}_{[\![T]\!]} \cup A. \bigcup_{\mathsf{q} \in \mathrm{Q}_{[\![T]\!]}} \mathrm{A}^{\mathsf{q}}_{[\![T]\!]}$$

We shall define, for any term $\Gamma \vdash M : T$ and environment u for Γ, a CSP process which represents the game semantics of $\Gamma \vdash M : T$ with respect to u. This process is denoted $[\![\Gamma \vdash M : T]\!]^{CSP} u$, and it is over the alphabet

$$\begin{array}{l}
K^{x}_{\mathsf{exp}D} = Q.q \rightarrow x.Q.q \rightarrow x.A?\mathsf{a} : \mathrm{A}^{q}_{[\![\mathsf{exp}D]\!]} \rightarrow A.\mathsf{a} \rightarrow \mathit{SKIP} \\
K^{x}_{\mathsf{com}} = Q.run \rightarrow x.Q.run \rightarrow x.A?\mathsf{a} : \mathrm{A}^{run}_{[\![\mathsf{com}]\!]} \rightarrow A.\mathsf{a} \rightarrow \mathit{SKIP} \\
K^{x}_{\mathsf{var}D} = (Q.read \rightarrow x.Q.read \rightarrow x.A?\mathsf{a} : \mathrm{A}^{read}_{[\![\mathsf{var}D]\!]} \rightarrow A.\mathsf{a} \rightarrow \mathit{SKIP}) \\
\quad \Box\, (Q.write?v : \mathcal{A}_{[\![D]\!]} \rightarrow x.Q.write.v \rightarrow x.A?\mathsf{a} : \mathrm{A}^{write.v}_{[\![\mathsf{var}D]\!]} \rightarrow A.\mathsf{a} \rightarrow \mathit{SKIP}) \\
K^{x}_{B_1 \times \ldots \times B_k \rightarrow B} = Q?\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \rightarrow x.Q.\mathsf{q} \\
\quad \rightarrow \mu\, L \,\bullet \Big(\mathit{SKIP} \,\Box \Big(\Box_{j=1}^{k} \big(x.Q.j?\mathsf{q}_j : \mathrm{Q}_{[\![B_j]\!]} \rightarrow Q.j.\mathsf{q}_j \rightarrow A.j?\mathsf{a}_j : \mathrm{A}^{\mathsf{q}_j}_{[\![B_j]\!]} \\
\quad \rightarrow x.A.j.\mathsf{a}_j \rightarrow \mathit{SKIP}\big) \,\text{;}\, L \Big) \Big) \,\text{;}\, x.A?\mathsf{a} : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \rightarrow A.\mathsf{a} \rightarrow \mathit{SKIP}
\end{array}$$

Table 4.3: Copy-cat processes for free identifiers

$\mathcal{A}_{[\![\Gamma \vdash T]\!]}$ defined as follows:

$$\mathcal{A}_{[\![x:T]\!]} = x.\mathcal{A}_{[\![T]\!]} \quad \mathcal{A}_{[\![\Gamma]\!]} = \bigcup_{x:T \in \Gamma} \mathcal{A}_{[\![x:T]\!]}$$
$$\mathcal{A}_{[\![\Gamma \vdash T]\!]} = \mathcal{A}_{[\![\Gamma]\!]} \cup \mathcal{A}_{[\![T]\!]} \cup \{abort\}$$

The standard game semantics of $\Gamma \vdash M : T$ is obtained by using the environment u which is a mapping such that, for any $x : T \in \Gamma$, $u(x)$ is the copy-cat process K^x_T, defined in Table 4.3. The process K^x_T represents the copy-cat strategy of game semantics, and it describes the "most general" behaviour of a sequential procedure.

Apart from defining a process $[\![\Gamma \vdash M : T]\!]^{CSP} u$ for any term $\Gamma \vdash M : T$, we also define a process $[\![\Gamma \vdash M : T]\!]^{*CSP} u$ which can repeat the strategy of $\Gamma \vdash M : T$ arbitrarily many times. This is done as follows:

(i) for each language construct 'c', we define a process for its strategy P_{c}.

(ii) for each process P_{c}, we define a process P^*_{c} which performs the strategy of 'c' arbitrarily many times:

$$P^*_{\mathsf{c}} = \mathit{SKIP} \,\Box\, (P_{\mathsf{c}} \,\text{;}\, P^*_{\mathsf{c}}).$$

(iii) for each composite term $\mathsf{c}(M_1, \ldots, M_n)$ consisting of a language construct 'c' and subterms $M_1, \ldots, M_n$, we define $[\![\mathsf{c}(M_1, \ldots, M_n)]\!]^{*CSP} u$ from the process P^*_{c} and processes $[\![M_i]\!]^{*CSP} u$, using only renaming, parallel composition and hiding.

(iv) each process $[\![\mathsf{c}(M_1, \ldots, M_n)]\!]^{CSP} u$ is defined as $[\![\mathsf{c}(M_1, \ldots, M_n)]\!]^{*CSP} u$, except that the process P_{c} is used instead of P^*_{c}, and for some immediate subterms, $[\![M_i]\!]^{CSP} u$ can be used instead of $[\![M_i]\!]^{*CSP} u$.

For any term $\Gamma \vdash M : T$, we could have defined $[\![\Gamma \vdash M : T]\!]^{*CSP} u$ as

$$[\![\Gamma \vdash M : T]\!]^{*CSP} u = \mu\, p \centerdot SKIP \,\Box\, \left([\![\Gamma \vdash M : T]\!]^{CSP} u \,;\, p\right)$$

Indeed, these processes have the same finite traces. However, our definitions of $[\![\Gamma \vdash M : T]\!]^{CSP} u$ and $[\![\Gamma \vdash M : T]\!]^{*CSP} u$ use the CSP operators of hiding, parallel composition and renaming outside any uses of recursion and prefixing. This means that an FDR debugging feature is applicable, which enables us, for any $t \in \mathsf{traces}([\![\Gamma \vdash M : T]\!]^{CSP} u)$, to identify all its hidden events without further model checking. This property will simplify the implementation of the abstraction refinement procedure (see Chapter 5).

Expressions The representation of expressions is given in Tables 4.4 and 4.5.

$$[\![\Gamma \vdash v : \mathsf{exp}D]\!]^{CSP} u = Q.q \rightarrow A.v \rightarrow SKIP,\ v \in \mathcal{A}_{[\![D]\!]}$$

$$[\![\Gamma \vdash E_1\, \mathsf{op}_D E_2 : \mathsf{exp}D]\!]^{CSP} u = [\![\Gamma \vdash E_1 : \mathsf{exp}D_1]\!]^{CSP} u[Q_1/Q, A_1/A] \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \left([\![\Gamma \vdash E_2 : \mathsf{exp}D_2]\!]^{CSP} u[Q_2/Q, A_2/A] \underset{\{\mid Q_2, A_2 \mid\}}{\parallel} P_{\mathsf{op}:D_1 \times D_2 \rightarrow D} \setminus \{\mid Q_2, A_2 \mid\}\right) \setminus \{\mid Q_1, A_1 \mid\}$$

Table 4.4: Processes for expressions

$$
\begin{aligned}
& P_{\mathsf{op}:\widetilde{D_1}\times\widetilde{D_2}\to\widetilde{D}} = Q.q \to Q_1.q \to A_1?\mathsf{a}_1 : \mathrm{A}_{[\![\mathsf{exp}\widetilde{D_1}]\!]} \to \\
& \qquad Q_2.q \to A_2?\mathsf{a}_2 : \mathrm{A}_{[\![\mathsf{exp}\widetilde{D_2}]\!]} \to A.\mathsf{a}_1\,\mathsf{op}\,\mathsf{a}_2 \to \mathit{SKIP} \\
& \qquad\qquad \text{for } D_1 = \widetilde{D_1} \text{ and } D_2 = \widetilde{D_2} \text{ and } D = \widetilde{D} \\
& P_{\mathsf{op}:D_1\times D_2\to D} = Q.q \to Q_1.q \to A_1?\mathsf{a}_1 : \mathrm{A}_{[\![\mathsf{exp}D_1]\!]} \to \\
& \qquad Q_2.q \to A_2?\mathsf{a}_2 : \mathrm{A}_{[\![\mathsf{exp}D_2]\!]} \to \big(\mathsf{let}\ S = \mathsf{op}_{D_1\times D_2\to D}(\mathsf{a}_1,\mathsf{a}_2)\ \mathsf{within} \\
& \qquad \mathsf{if}\ (\mid S \mid = 1)\ \mathsf{then}\ \big(A?v : S \to \mathit{SKIP}\big)\ \mathsf{else}\ \big(\mathsf{if}\ (\mathsf{a}_1\ \mathsf{is\ not\ singleton})\ \mathsf{then} \\
& \qquad (A?v : S \to nd.\mathsf{a}_1 \to \mathit{SKIP})\ \mathsf{else}\ (A?v : S \to nd.\mathsf{a}_2 \to \mathit{SKIP})\big)\big) \\
& \qquad\qquad \text{for } D_1 \neq \widetilde{D_1} \text{ or } D_2 \neq \widetilde{D_2} \text{ or } D \neq \widetilde{D}
\end{aligned}
$$

Table 4.5: Processes for op construct

The CSP process for an integer or boolean constant replies to the question q by the value of that constant, and then terminates. For an operator application $E_1\ \mathsf{op}_D\ E_2$, we compose the processes for E_1 and E_2, and a process for op. As with all processes which represent strategies here, the composition is performed by the CSP operators of renaming, parallel and hiding. The process for op asks for values of the arguments, and after obtaining them responds by performing the operation. For each operation of type $\mathsf{exp}D_1 \times \mathsf{exp}D_2 \to \mathsf{exp}D$ which contains some abstractions, there exists a function $\mathsf{op}_{D_1\times D_2\to D}$ implementing it. If the function of an operation returns more than one result for the operands, then a special marker move $nd.a$ is performed. In such an instance, the operation necessarily has at least one abstracted integer operand which is not a singleton, i.e. which abstracts more than one integer. The process for the operation then performs the marker move $nd.a$, where a is such an operand. This move is neither Opponent nor Player, but only marks that nondeterminism has occurred. It is used for implementing efficiently the abstraction refinement procedure (see Chapter 5). In Tables 4.6 and 4.7 are shown the functions implementing the equality and addition operation. The other operations are implemented similarly and the CSP script containing functions

$=_{\mathsf{int}_{[n_1,m_1]}\times\mathsf{int}_{[n_2,m_2]}\to\mathsf{bool}}(v_1, v_2) :=$
  if $(v_1 = \mathbb{Z})$ or $(v_2 = \mathbb{Z})$ then $\{tt, \mathit{ff}\}$ else
  if $(v_1 = > m_1)$ then
    if $(v_2 \neq > m_2)$ and $(v_2 \leq m_1)$ then $\{\mathit{ff}\}$
      else $\{tt, \mathit{ff}\}$ else
  if $(v_1 = < n_1)$ then
    if $(v_2 \neq < n_2)$ and $(v_2 \geq n_1)$ then $\{\mathit{ff}\}$
      else $\{tt, \mathit{ff}\}$ else
  if $(v_2 = > m_2)$ then
    if $(v_1 \neq > m_1)$ and $(v_1 \leq m_2)$ then $\{\mathit{ff}\}$
      else $\{tt, \mathit{ff}\}$ else
  if $(v_2 = < n_2)$ then
    if $(v_1 \neq < n_1)$ and $(v_1 \geq n_2)$ then $\{\mathit{ff}\}$
      else $\{tt, \mathit{ff}\}$ else
  $\{v_1 = v_2)$

Table 4.6: An implementation of the equality operation

which correspond to all operations is given in Appendix A.

Commands Tables 4.8 and 4.9 show processes for the commands. For sequential composition, conditional and iteration, processes for the components are composed with a process for the construct itself, similarly to how the process for $E_1 \,\mathsf{op}_D\, E_2$ was defined above. However, in the case of the conditional, one of the processes for M_{tt} and M_{ff} will not be run, so *SKIP* is used for such empty termination. For iteration, the processes for E and C may be run arbitrarily many times, so we use the corresponding $[\![-]\!]^{*CSP}$ processes. The function $\mathsf{cast}_{B',B}$ converts a value a of type B' into type B such that if more than one value of type B is obtained the special marker move $nd.a$ is performed. If $B' = B$, then $\mathsf{cast}_{B',B}$ is the identity function, otherwise its

$+_{\mathsf{int}_{[n_1,m_1]} \times \mathsf{int}_{[n_2,m_2]} \rightarrow \mathsf{int}_{[n,m]}}(v_1, v_2) =$
    if $(v_1 = \mathbb{Z})$ or $(v_2 = \mathbb{Z})$ then $\{\mathbb{Z}\}$ else
    if $(v_1 = > m_1)$ then
        if $(v_2 \geq m_2)$ and $(m_1 + m_2 \geq m)$ then $\{> m\}$ else
        if $(v_2 \neq < n_2)$ and $(v_2 + m_1 + 1 \geq n)$ then $\{v_2 + m_1 + 1, \ldots, m, > m\}$
            else $\{< n, n, \ldots, m, > m\}$ else
    if $(v_1 = < n_1)$ then
        if $(v_2 \leq n_2)$ and $(n_1 + n_2 \leq n)$ then $\{< n\}$ else
        if $(v_2 \neq > m_2)$ and $(v_2 + n_1 - 1 \leq m)$ then $\{< n, n, \ldots, v_2 + n_1 - 1\}$
            else $\{< n, n, \ldots, m, > m\}$ else
    if $(v_2 = > m_2)$ then
        if $(v_1 \geq m_1)$ and $(m_1 + m_2 \geq m)$ then $\{> m\}$ else
        if $(v_1 \neq < n_1)$ and $(v_1 + m_2 + 1 \geq n)$ then $\{v_1 + m_2 + 1, \ldots, m, > m\}$
            else $\{< n, n, \ldots, m, > m\}$ else
    if $(v_2 = < n_2)$ then
        if $(v_1 \leq n_1)$ and $(n_1 + n_2 \leq n)$ then $\{< n\}$ else
        if $(v_1 \neq > m_1)$ and $(v_1 + n_2 - 1 \leq m)$ then $\{< n, n, \ldots, v_1 + n_2 - 1\}$
            else $\{< n, n, \ldots, m, > m\}$ else
    if $(v_1 + v_2 > m)$ then $\{> m\}$ else
    if $(v_1 + v_2 < n)$ then $\{< n\}$ else
    $\{v_1 + v_2)$

Table 4.7: An implementation of the addition operation

implementation is given below.

$\mathsf{cast}_{\mathsf{expint}_{[n',m']},\mathsf{expint}_{[n,m]}}(v') :=$
    if $([n, m] = [\,])$ then $\{\mathbb{Z}\}$ else
    if $(v' = \mathbb{Z})$ then $\{< n, n, \ldots, m, > m\}$ else
    if $(v' = > m')$ then
        if $(m' \geq m)$ then $\{> m\}$ else $\{m' + 1, \ldots, m, > m\}$ else
    if $(v' = < n')$ then
        if $(n' \leq n)$ then $\{< n\}$ else $\{< n, n, \ldots, n' - 1\}$ else
    if $(v' > m)$ then $\{> m\}$ else
    if $(v' < n)$ then $\{< n\}$ else
    $\{v'\}$

$$
\begin{array}{l}
[\![\Gamma \vdash \mathsf{skip} : \mathsf{com}]\!]^{CSP} u = Q.run \rightarrow A.done \rightarrow SKIP \\
[\![\Gamma \vdash \mathsf{abort} : \mathsf{com}]\!]^{CSP} u = Q.run \rightarrow abort \rightarrow STOP \\
[\![\Gamma \vdash C \ ; \ M : B]\!]^{CSP} u = \\
\quad [\![\Gamma \vdash C : \mathsf{com}]\!]^{CSP} u[Q_1/Q, A_1/A] \underset{\{|Q_1,A_1|\}}{\parallel} \\
\quad \big([\![\Gamma \vdash M : B]\!]^{CSP} u[Q_2/Q, A_2/A] \underset{\{|Q_2,A_2|\}}{\parallel} \\
\quad P_{;:B} \setminus \{| \ Q_2, A_2 \ |\}\big) \setminus \{| \ Q_1, A_1 \ |\} \\
[\![\Gamma \vdash \mathsf{if}_B \ E \ \mathsf{then} \ M_{tt} \ \mathsf{else} \ M_{ff} : B]\!]^{CSP} u = \\
\quad [\![\Gamma \vdash E : \mathsf{expbool}]\!]^{CSP} u[Q_0/Q, A_0/A] \underset{\{|Q_0,A_0|\}}{\parallel} \\
\quad \Big(([\![\Gamma \vdash M_{tt} : B_1]\!]^{CSP} u[Q_1/Q, A_1/A] \,\square\, SKIP) \underset{\{|Q_1,A_1|\}}{\parallel} \\
\quad \big(([\![\Gamma \vdash M_{ff} : B_2]\!]^{CSP} u[Q_2/Q, A_2/A] \,\square\, SKIP) \underset{\{|Q_2,A_2|\}}{\parallel} \\
\quad P_{\mathsf{if}:\mathsf{expbool} \times B_1 \times B_2 \rightarrow B} \setminus \{| \ Q_2, A_2 \ |\}\big) \setminus \{| \ Q_1, A_1 \ |\}\Big) \setminus \{| \ Q_0, A_0 \ |\} \\
[\![\Gamma \vdash \mathsf{while} \ E \ \mathsf{do} \ C : \mathsf{com}]\!]^{CSP} u = \\
\quad [\![E : \mathsf{expbool}]\!]^{*CSP} u[Q_1/Q, A_1/A] \underset{\{|Q_1,A_1|\}}{\parallel} \\
\quad \big([\![C : \mathsf{com}]\!]^{*CSP} u[Q_2/Q, A_2/A] \underset{\{|Q_2,A_2|\}}{\parallel} \\
\quad P_{\mathsf{while}:\mathsf{com}} \setminus \{| \ Q_2, A_2 \ |\}\big) \setminus \{| \ Q_1, A_1 \ |\}
\end{array}
$$

Table 4.8: Processes for commands

Variables The processes for assignment and de-referencing are straightforward. In the definition for local-variable declarations, a 'cell' process $U_D(x, v)$ is used for remembering the initial or the most-recently written value into the variable x. It is composed with the process for the scope of the declaration, ensuring 'good variable' behaviour. Tables 4.10 and 4.11 contain the details.

Functionals Table 4.12 contains the remaining process definitions: for free identifier, function application and function declaration terms. In each case, environments are used to access or record processes associated with free identifiers. For function application, the processes for the arguments may be run arbitrarily many times.

$$
\begin{array}{l}
P_{;:B} = Q?\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to Q_1.run \to A_1?\mathsf{a}_1 : \mathrm{A}^{run}_{[\![\mathsf{com}]\!]} \to \\
\quad Q_2.\mathsf{q} \to A_2?\mathsf{a} : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \to A.\mathsf{a} \to SKIP \\
P_{\mathsf{if}:\mathsf{expbool}\times B_1\times B_2\to B} = Q.\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to Q_0.q \to A_0?\mathsf{a}_0 : \mathrm{A}^{q}_{[\![\mathsf{expbool}]\!]} \to \\
\quad \mathsf{if}\ (\mathsf{a}_0)\ \mathsf{then}\Big(Q_1.\mathsf{q} \to A_1?\mathsf{a}_1 : \mathrm{A}^{\mathsf{q}}_{[\![B_1]\!]} \to \big(\mathsf{let}\ S = \mathsf{cast}_{B_1,B}(\mathsf{a}_1)\ \mathsf{within} \\
\quad \mathsf{if}\ (\mid S \mid = 1)\ \mathsf{then}\ (A?v : S \to SKIP)\ \mathsf{else}(A?v : S \to nd.\mathsf{a}_1 \to SKIP)\big)\Big) \\
\quad \mathsf{else}\ \Big(Q_2.\mathsf{q} \to A_2?\mathsf{a}_2 : \mathrm{A}^{\mathsf{q}}_{[\![B_2]\!]} \to \big(\mathsf{let}\ S = \mathsf{cast}_{B_2,B}(\mathsf{a}_2)\ \mathsf{within} \\
\quad \mathsf{if}\ (\mid S \mid = 1)\ \mathsf{then}\ (A?v : S \to SKIP)\ \mathsf{else}\ (A?v : S \to nd.\mathsf{a}_2 \to SKIP)\big)\Big) \\
P_{\mathsf{while}:\mathsf{com}} = Q.run \to \mu\, p \centerdot Q_1.q \to A_1?\mathsf{a}_1 : \mathrm{A}^{q}_{[\![\mathsf{expbool}]\!]} \to \\
\quad \Big(\mathsf{if}\ (\mathsf{a}_1)\ \mathsf{then}\ \big(Q_2.run \to A_2?\mathsf{a}_2 : \mathrm{A}^{run}_{[\![\mathsf{com}]\!]} \to p\big)\ \mathsf{else}\ \big(A.done \to SKIP\big)\Big)
\end{array}
$$

Table 4.9: Processes for command constructs

4.3.3 Correctness and Property Verification

For any term from AIA_2, the set of all even-length traces of its CSP interpretation is isomorphic to its regular language interpretation $[\![-]\!]^R$, as defined in Section 4.2.

Theorem 4.3.1 *For any term $\Gamma \vdash M : T$, we have:*

$$
\mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash M : T]\!]^{CSP} u) \ \stackrel{\phi}{\equiv}\ \mathsf{closure}([\![\Gamma \vdash M : T]\!]^{R} u') \tag{4.1}
$$

where u and u' are the environments that map free identifiers of the term to copy-cat processes and regular languages respectively, and ϕ is an isomorphism defined by:

$$
\begin{array}{rll}
\phi(Q.a) = a & \phi(A.a) = a & \mathit{if}\, a \in \mathcal{A}_{[\![B]\!]} \\
\phi(Q.i.a) = a^{\langle i \rangle} & \phi(A.i.a) = a^{\langle i \rangle} & \mathit{if}\, a \in \mathcal{A}_{[\![B_i]\!]}, 1 \le i \le l \\
\phi(x_j.Q.a) = a^{\langle x_j \rangle} & \phi(x_j.A.a) = a^{\langle x_j \rangle} & \mathit{if}\, a \in \mathcal{A}_{[\![B'_j]\!]}, 1 \le j \le k \\
\phi(x_j.Q.i.a) = a^{\langle x_j, i \rangle} & \phi(x_j.A.i.a) = a^{\langle x_j, i \rangle} & \mathit{if}\, a \in \mathcal{A}_{[\![B_{j,i}]\!]}, 1 \le j \le k, 1 \le i \le l_j
\end{array}
$$

$$
\begin{array}{l}
[\![\Gamma \vdash V := M : \mathsf{com}]\!]^{CSP} u = \\
\quad [\![\Gamma \vdash M : \mathsf{exp} D_2]\!]^{CSP} u[Q_1/Q, A_1/A] \underset{\{| Q_1, A_1 |\}}{\|} \\
\quad \big([\![\Gamma \vdash V : \mathsf{var} D_1]\!]^{CSP} u[Q_2/Q, A_2/A] \underset{\{| Q_2, A_2 |\}}{\|} \\
\quad P_{:=:\mathsf{var}D_1 \times \mathsf{exp}D_2 \to \mathsf{com}} \setminus \{| \; Q_2, A_2 \; |\}\big) \setminus \{| \; Q_1, A_1 \; |\} \\
[\![\Gamma \vdash ! V : \mathsf{exp} D]\!]^{CSP} u = \\
\quad [\![\Gamma \vdash V : \mathsf{var} D]\!]^{CSP} u[Q_1/Q, A_1/A] \underset{\{| Q_1, A_1 |\}}{\|} P_{!:\mathsf{var}D \to \mathsf{exp}D} \setminus \{| \; Q_1, A_1 \; |\} \\
[\![\Gamma \vdash \mathsf{new}_D \, x := v \, \mathsf{in} \, C : \mathsf{com}]\!]^{CSP} u = \\
\quad [\![\Gamma, x : \mathsf{var} D \vdash C : \mathsf{com}]\!]^{CSP} (u \mid x \mapsto K^x_{\mathsf{var}D})[Q_1/Q, A_1/A] \underset{\{| Q_1, A_1, x |\}}{\|} \\
\quad P_{\mathsf{new}:D}(x, v) \setminus \{| \; Q_1, A_1 \; |\}
\end{array}
$$

Table 4.10: Processes for variables

$$
\begin{array}{l}
P_{:=:\mathsf{var}D_1 \times \mathsf{exp}D_2 \to \mathsf{com}} = Q.run \to Q_1.q \to A_1?\mathsf{a}_1 : \mathrm{A}^q_{[\![\mathsf{exp}D_2]\!]} \to \\
\qquad \mathsf{let} \; S = \mathsf{cast}_{\mathsf{exp}D_1, \mathsf{exp}D}(\mathsf{a}_1) \; \mathsf{within} \; \mathsf{if} \; (| \; S \; | = 1) \\
\qquad \mathsf{then} \; \big(Q_2.write?v : S \to A_2.ok \to A.done \to SKIP\big) \\
\qquad \mathsf{else} \; \big(Q_2.write?v : S \to A_2.ok \to A.done \to nd.\mathsf{a}_1 \to SKIP\big) \\
P_{!:\mathsf{var}D \to \mathsf{exp}D} = Q.q \to Q_1.read \to A_1?\mathsf{a} : \mathrm{A}^{read}_{[\![\mathsf{var}D]\!]} \to A.\mathsf{a} \to SKIP \\
P_{\mathsf{new}:D}(x, v) = Q.run \to Q_1.run \to U_D(x, v) \\
U_D(x, v) = \big(x.Q.read \to x.A.v \to U_D(x, v)\big) \\
\qquad \Box \; \big(x.Q.write?v' : \mathcal{A}_{[\![D]\!]} \to x.A.ok \to U_D(x, v')\big) \\
\qquad \Box \big(A_1.done \to A.done \to SKIP\big)
\end{array}
$$

Table 4.11: Processes for variable constructs

and $\phi(abort) = abort$ *for* $\Gamma = x_1 : T_1, \ldots, x_k : T_k, \; T_j = B_{j,1} \times \ldots \times B_{j,l_j} \to B'_j$ *and* $T = B_1 \times \ldots \times B_l \to B$.

Proof The proof is by a routine induction on the typing rules, by showing that the definitions of CSP processes in Section 4.3 correspond to the definitions of regular languages in Section 4.2.

Language constants and constructs. For any constant v,

$$\mathsf{traces}([\![\Gamma \vdash v : \mathsf{exp}D]\!]^{CSP} u) = \{\varepsilon, Q.q, Q.q \cdot A.v, Q.q \cdot A.v \cdot \checkmark\}$$

$$
\begin{array}{l}
[\![\Gamma \vdash x : T]\!]^{CSP} u = u(x) \\
[\![\Gamma \vdash x(M_1 \ldots M_k) : B]\!]^{CSP} u = \\
\quad [\![\Gamma \vdash M_1 : B_1]\!]^{*CSP} u[Q.1/Q, A.1/A] \underset{\{|Q.1, A.1|\}}{\|} \\
\quad \ldots\ldots\ldots \\
\quad \Big([\![\Gamma \vdash M_k : B_k]\!]^{*CSP} u[Q.k/Q, A.k/A] \underset{\{|Q.k, A.k|\}}{\|} \\
\quad [\![\Gamma \vdash x : B_1 \times \ldots B_k \to B]\!]^{CSP} u \setminus \{|\ Q.k, A.k\ |\} \Big) \cdots \setminus \{|\ Q.1, A.1\ |\} \\
[\![\Gamma \vdash \mathsf{let}\, x\, \mathsf{be}\, N\, \mathsf{in}\, M : T']\!]^{CSP} u = \\
\quad [\![\Gamma, x : T \vdash M : T']\!]^{CSP} \big(u \mid x \mapsto [\![\Gamma \vdash N : T]\!]^{CSP} u\big)
\end{array}
$$

Table 4.12: Processes for functionals

$$
\mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash v]\!]^{CSP} u) = \{\varepsilon, Q.q \cdot A.v\} \overset{\phi}{\equiv} \{\varepsilon, q \cdot v\} = \mathsf{closure}([\![\Gamma \vdash v]\!]^{R} u')
$$

The proofs for the other constants and constructs are similar.

Arithmetic-logic operations. By inductive hypothesis, we have that $[\![\Gamma \vdash E_1]\!]^{CSP} u$ and $[\![\Gamma \vdash E_1]\!]^{R} u'$, $[\![\Gamma \vdash E_2]\!]^{CSP} u$ and $[\![\Gamma \vdash E_2]\!]^{R} u'$, and $P_{\mathsf{op}:D_1 \times D_2 \to D}$ and $[\![\mathsf{op}_D]\!]^{R} u'$ are ϕ-isomorphic. Then, by the isomorphism between two definitions of composition, we have that $\mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash E_1\, \mathsf{op}_D E_2]\!]^{CSP} u) \overset{\phi}{\equiv} \mathsf{closure}([\![\Gamma \vdash E_1\, \mathsf{op}_D E_2]\!]^{R} u')$.

Free identifiers. It can be verified by inspection that copy-cat processes and copy-cat regular-languages are ϕ-isomorphic.

Local variables. The semantics of a local variable is achieved by imposing the good variable behavior on the local variable and removing all moves involving the variable. The first condition is imposed by intersection of regular languages and parallel composition of processes, while the second condition is imposed by restriction and hiding in the regular-language and CSP representation, respectively. Hence, the isomorphism between two

representations follows from $\mathsf{traces}^{\mathsf{ev}}(U_D(x, v)) \stackrel{\phi}{\equiv} \mathsf{closure}(\gamma_v^x)$. The latter can be seen by inspection.

Representations of function application and function declaration are isomorphic by inductive hypothesis and the isomorphism between two definitions of composition. ∎

As a corollary we obtain the full abstraction result for CSP representation.

Theorem 4.3.2 *For any two AIA_2 terms*

$$\Gamma \vdash M \cong N \quad \textit{iff} \quad \mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash M : T]\!]^{CSP} u) = \mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash N : T]\!]^{CSP} u)$$

Proof It follows from Theorem 4.3.1 and Theorem 4.2.2. ∎

By Theorem 4.3.2, we have that observational safe-equivalence is captured by two traces refinements:

Corollary 4.3.3 (Observational safe-equivalence)

$$\begin{aligned} \Gamma \vdash M \cong N \ \Leftrightarrow \ & [\![\Gamma \vdash M : T]\!]^{CSP} u \ \sqsubseteq_T \ [\![\Gamma \vdash N : T]\!]^{CSP} u \\ & [\![\Gamma \vdash N : T]\!]^{CSP} u \ \sqsubseteq_T \ [\![\Gamma \vdash M : T]\!]^{CSP} u \end{aligned}$$

Refinement checking in FDR terminates for finite-state processes, i.e. those whose labelled transition systems are finite. Our next result confirms that this is the case for the processes interpreting finitely abstracted terms. As a corollary, we have that observational safe-equivalence is decidable using FDR.

Theorem 4.3.4 *For any finitely abstracted term $\Gamma \vdash M : T$, the CSP process $[\![\Gamma \vdash M : T]\!]^{CSP} u$ is finite state.*

Proof Since the copy-cat processes K_T^x are finite state, the theorem is implied by the following claim: for any term $\Gamma \vdash M : T$ and any environment u which maps each identifier in Γ to a finite-state process, $[\![\Gamma \vdash M : T]\!]^{CSP} u$ is finite state.

In the fragment of CSP we are using, the only operators which can result in infinite-state transition systems are the infinite choice operator $?x : A \rightarrow P(x)$ with an infinite set A, and recursion. The claim therefore follows by induction on the typing rules of AIA, and these observations:

- each alphabet $\mathcal{A}_{[\![\Gamma \vdash T]\!]}$ is finite.
- each use of the choice operator is over a finite set.
- the recursive process in the definition of $K^x_{B_1 \times \ldots \times B_k \rightarrow B}$ is finite state by inspection.
- the recursive processes $U_D(x : \mathsf{var}D, v : \mathcal{A}_{[\![D]\!]})$ are finite state because $\mathcal{A}_{[\![D]\!]}$ is a finite set.
- the recursive processes in the definitions for iteration and function application are finite state by the inductive hypothesis. ∎

Corollary 4.3.5 (Decidability) Observational safe-equivalence *of finitely abstracted terms of* AIA_2 *is decidable by two traces refinements between finite-state CSP processes.*

We now consider an example equivalence and prove it using the CSP model.

Example Consider the process for the term

$$c : \mathsf{com} \vdash \mathsf{new}_{\mathsf{int}}\, x := 0 \,\mathsf{in}\, c : \mathsf{com}$$

It has the same traces as

$$(Q_1.run \to c.Q.run \to c.A.done \to A_1.done \to SKIP)$$
$$\underset{|A_1,Q_1,x|}{\|} (Q.run \to Q_1.run \to U_{\mathsf{int}}(x,0)) \backslash \{| \, Q_1, A_1, x \, |\}$$

Simplifying further yields

$$Q.run \to c.Q.run \to c.A.done \to A.done \to SKIP$$

which is the process for the term $c : \mathsf{com} \vdash c : \mathsf{com}$.

By Corollary 4.3.5, we conclude that

$$c : \mathsf{com} \vdash \mathsf{new}_{\mathsf{int}}\, x := 0 \,\mathsf{in}\, c : \mathsf{com} \;\cong\; c : \mathsf{com}$$

This simple equivalence reflects the fact that a non-locally defined command cannot modify a local variable [51]. ∎

In addition to checking observational safe-equivalence of two program terms, it is desirable to be able to check properties of terms. We focus on verifying safety properties, and take the view that a term $\Gamma \vdash M : T$ is safe if and only if all traces in $\mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash M : T]\!]^{CSP} u)$ are safe. Recall from Section 3.7 that a trace is unsafe if it terminates in *abort*; otherwise it is safe.

Corollary 4.3.6 (Decidability) Safety *of a finitely abstracted term* $\Gamma \vdash M : T$ *is decidable by one traces refinement between finite-state CSP processes:*

$$\mu\, p \,\centerdot\, SKIP \,\Box \left(?x : \mathcal{A}_{[\![\Gamma \vdash T]\!]} \backslash \{abort\} \to p\right) \;\sqsubseteq_T\; [\![\Gamma \vdash M : T]\!]^{CSP} u$$

Example Consider the term

$$f : \mathsf{com} \times \mathsf{com} \to \mathsf{com} \vdash \mathsf{new}_{\mathsf{int}[0,1]}\, x := 0 \,\mathsf{in}$$
$$f\left(x := !x +_{[0,2]} 1,\ \mathsf{if}\ (!x >_{\mathsf{bool}} 1)\ \mathsf{then\ abort}\right)$$

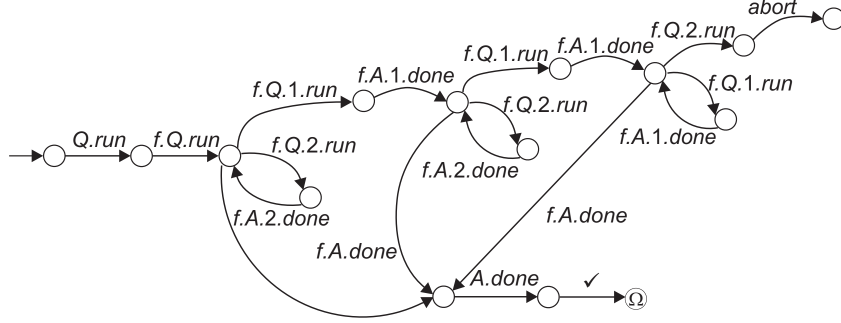


Figure 4.3: A strategy as a labelled transition system

in which f is a non-local procedure.

The LTS of the CSP process representing this term is shown in Figure 4.3. It contains an unsafe play: $Q.run \cdot f.Q.run \cdot f.Q.1.run \cdot f.A.1.done \cdot f.Q.1.run \cdot f.A.1.done \cdot f.Q.2.run \cdot abort$. So, the term is unsafe. ∎

4.3.4 Type Inference System

We want to simplify writing of abstracted terms in AIA by using a *type inference system*. Given abstractions for all free identifiers and local variables, the type inference system determines all other abstractions from the former by inference. The judgments of the type inference system have the form:

$$\Gamma \vdash_{\mathsf{TIS}} M : T$$

and are defined by the axioms and rules given in Table 4.13.

Abstractions $[0, n]$ are given to integer constants $n \geq 0$, and $[n, 0]$ to integer constants $n < 0$. For operations op whose concrete type is $\mathsf{expint} \times \mathsf{expint} \rightarrow \mathsf{expint}$, the abstraction of the result type will be the most refined abstraction such that the operation between any concrete values of the operands will evaluate to a concrete value as well. The rule for conditionals and mkvar are similar. All other rules are equivalent to the typing rules of AIA. We show

$$x_1 : T_1, \ldots, x_k : T_k \vdash_{\mathsf{TIS}} x_i : T_i,\ i \in \{1, \ldots, k\}$$

$$\Gamma \vdash_{\mathsf{TIS}} n : \mathsf{expint}_{[0,n]},\ n \geq 0 \qquad \Gamma \vdash_{\mathsf{TIS}} n : \mathsf{expint}_{[n,0]},\ n < 0$$

$$\Gamma \vdash_{\mathsf{TIS}} b : \mathsf{expbool},\ b \in \{tt, ff\}$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{exp}D_1 \qquad \Gamma \vdash_{\mathsf{TIS}} N : \mathsf{exp}D_2}{\Gamma \vdash_{\mathsf{TIS}} M \,\mathsf{op}\, N : \mathsf{expbool}}\ \mathsf{op} : \mathsf{exp}\widetilde{D_1} \times \mathsf{exp}\widetilde{D_2} \rightarrow \mathsf{expbool}$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{expint}_{[n_1,m_1]} \qquad \Gamma \vdash_{\mathsf{TIS}} N : \mathsf{expint}_{[n_2,m_2]}}{\Gamma \vdash_{\mathsf{TIS}} M \,\mathsf{op}\, N : \mathsf{expint}_{[min\{x_1 \mathsf{op} x_2\}, max\{x_1 \mathsf{op} x_2\}]}}\ \begin{array}{l}\mathsf{op} : \mathsf{exp}\mathsf{int} \times \mathsf{exp}\mathsf{int} \rightarrow \mathsf{exp}\mathsf{int} \\ n_1 \leq x_1 \leq m_1,\, n_2 \leq x_2 \leq m_2\end{array}$$

$$\Gamma \vdash_{\mathsf{TIS}} \mathsf{skip} : \mathsf{com} \quad \Gamma \vdash_{\mathsf{TIS}} \mathsf{abort} : \mathsf{com} \qquad \frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{com} \qquad \Gamma \vdash_{\mathsf{TIS}} N : B}{\Gamma \vdash_{\mathsf{TIS}} M\, ;\ N : B}$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{expbool} \qquad \Gamma \vdash_{\mathsf{TIS}} N_{tt} : B_1 \qquad \Gamma \vdash_{\mathsf{TIS}} N_{ff} : B_2}{\Gamma \vdash_{\mathsf{TIS}} \mathsf{if}\ M\ \mathsf{then}\ N_{tt}\ \mathsf{else}\ N_{ff} : B_1}\ B_1 = B_2$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{expbool} \qquad \Gamma \vdash_{\mathsf{TIS}} N_{tt} : \mathsf{expint}_{[n_1,m_1]} \qquad \Gamma \vdash_{\mathsf{TIS}} N_{ff} : \mathsf{expint}_{[n_2,m_2]}}{\Gamma \vdash_{\mathsf{TIS}} \mathsf{if}\ M\ \mathsf{then}\ N_{tt}\ \mathsf{else}\ N_{ff} : \mathsf{expint}_{[min\{n_1,n_2\}, max\{m_1,m_2\}]}}$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{expbool} \qquad \Gamma \vdash_{\mathsf{TIS}} N : \mathsf{com}}{\Gamma \vdash_{\mathsf{TIS}} \mathsf{while}\ M\ \mathsf{do}\ N : \mathsf{com}}$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{var}D_1 \qquad \Gamma \vdash_{\mathsf{TIS}} N : \mathsf{exp}D_2}{\Gamma \vdash_{\mathsf{TIS}} M := N : \mathsf{com}}\ \widetilde{D_1} = \widetilde{D_2} \qquad \frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{var}D}{\Gamma \vdash_{\mathsf{TIS}} !M : \mathsf{exp}D}$$

$$\frac{\Gamma, x : \mathsf{var}D \vdash_{\mathsf{TIS}} M : B \qquad \Gamma \vdash_{\mathsf{TIS}} v : \mathsf{exp}D_1}{\Gamma \vdash_{\mathsf{TIS}} \mathsf{new}_D\ x := v\ \mathsf{in}\ M : B}\ \widetilde{D} = \widetilde{D_1}$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} N : B_1 \times \ldots \times B_k \rightarrow B \qquad \Gamma \vdash_{\mathsf{TIS}} M_i : B_i}{\Gamma \vdash_{\mathsf{TIS}} N(M_1, \ldots, M_k) : B}\ 1 \leq i \leq k$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} N : T \qquad \Gamma, x : T \vdash_{\mathsf{TIS}} M : T'}{\Gamma \vdash_{\mathsf{TIS}} \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M : T'}$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{exp}D_1 \rightarrow \mathsf{com} \qquad \Gamma \vdash_{\mathsf{TIS}} N : \mathsf{exp}D_2}{\Gamma \vdash_{\mathsf{TIS}} \mathsf{mkvar}\ MN : \mathsf{var}D_1}\ D_1 = D_2$$

$$\frac{\Gamma \vdash_{\mathsf{TIS}} M : \mathsf{expint}_{[n_1,m_1]} \rightarrow \mathsf{com} \qquad \Gamma \vdash_{\mathsf{TIS}} N : \mathsf{expint}_{[n_2,m_2]}}{\Gamma \vdash_{\mathsf{TIS}} \mathsf{mkvar}\ MN : \mathsf{varint}_{[min\{n_1,n_2\}, max\{m_1,m_2\}]}}$$

Table 4.13: Type inference system

that the type inference system is an extension of the underlying type system given in Table 2.1.

Lemma 4.3.7 *If* $\Gamma \vdash_{\mathsf{TIS}} M : T$ *then there exists an unique term* $\Gamma \vdash M : T$.

Proof We explicitly annotate all types of the subterms with the abstractions inferred using the rules of the type inference system in Table 4.13. ∎

Example The term

$$f : \mathsf{com} \times \mathsf{com} \rightarrow \mathsf{com} \vdash \mathsf{new}_{\mathsf{int}[0,1]}\, x := 0 \,\mathsf{in}$$
$$f\big(x := !x + 1,\ \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort}\big)$$

produces a term with the same abstractions as the term from the previous example. ∎

From now on, we proceed to work with the judgments of the form $\Gamma \vdash_{\mathsf{TIS}} M : T$ where the abstractions of subterms are inferred. We often abbreviate the above notation to $\Gamma \vdash M : T$.

4.3.5 Compiler and Applications

We have implemented a compiler in Java [15], which automatically converts a finitely abstracted AIA_2 term into a CSP process which represents its game semantics. The resulting CSP process is defined by a script in machine readable CSP [98] which the compiler outputs.

The scripts output by the compiler can be loaded into the tools ProBE for interactive exploration of labelled transition systems, and FDR for automatic analysis and interactive debugging. One of the functions of FDR is to check traces refinement between two finite-state processes. As we have seen

before, this can be used to decide observational safe-equivalence between two terms, and safety of a term.

FDR offers a number of hierarchical compression algorithms [97], which can be applied during either model generation or refinement checking. The scripts which our compiler produces normally contain instructions to apply diamond elimination (which eliminates all τ events from a labelled transition system) and strong bisimulation quotienting to subprocesses which model local variable declaration subterms. This exploits the fact that game semantics hides interactions between a local variable and its scope. The interaction events become τ events, enabling the model to be reduced.

We now consider applications of the approach proposed here for two kinds of example: a sorting algorithm, and an abstract data type implementation.

A sorting algorithm We first analyse the bubble-sort algorithm. The input to the compiler is in Figure 4.4, where the array size is a meta variable $k > 0$.

The program first copies the input array x into a local array a, which is then sorted and copied back into x. The local array a is not visible from the outside of the program, so only reads and writes of the non-local array x are seen in the model. A labelled transition system for $k = 2$ is shown in Figure 4.5. The left-hand half represents reads of all possible combinations of values from x, while the right-hand half represents writes of the same values in sorted order.

In order to compare efficiency of the tools based on CSP and regular-language representation [10], we have also implemented a compiler for finitary IA_2. We slightly change the bubble-sort term in Figure 4.4. The type of array

1 $x[k]$: $\mathsf{varint}_{[0,0]}$ $\vdash$
2 $\mathsf{new}_{\mathsf{int}[0,0]}$ $a[k] := 0$ in
3 $\mathsf{new}_{\mathsf{int}[0,k]}$ $i := 0$ in
4 while $(i < k)$ do $\{ a[i] :=!x[i]; \quad i :=!i + 1; \}$
5 $\mathsf{new}_{\mathsf{bool}}$ $\mathit{flag} := \mathit{true}$ in
6 while (flag) do $\{$
7 $\quad i := 0;$
8 $\quad \mathit{flag} := \mathit{false};$
9 $\quad$ while $(i < k - 1)$ do $\{$
10 $\quad\quad$ if $(a[i] > a[i + 1])$ then $\{$
11 $\quad\quad\quad \mathit{flag} := \mathit{true};$
12 $\quad\quad\quad$ $\mathsf{new}_{\mathsf{int}[0,0]}$ $\mathtt{temp} :=!a[i]$ in
13 $\quad\quad\quad a[i] :=!a[i + 1];$
14 $\quad\quad\quad a[i + 1] :=!\mathtt{temp}; \quad \}$
15 $\quad\quad i :=!i + 1; \quad \} \}$
16 $i := 0;$
17 while $(i < k)$ do $\{ x[i] :=!a[i]; \quad i :=!i + 1; \}$
18 : com

Figure 4.4: Source code of AIA_2 bubble sort

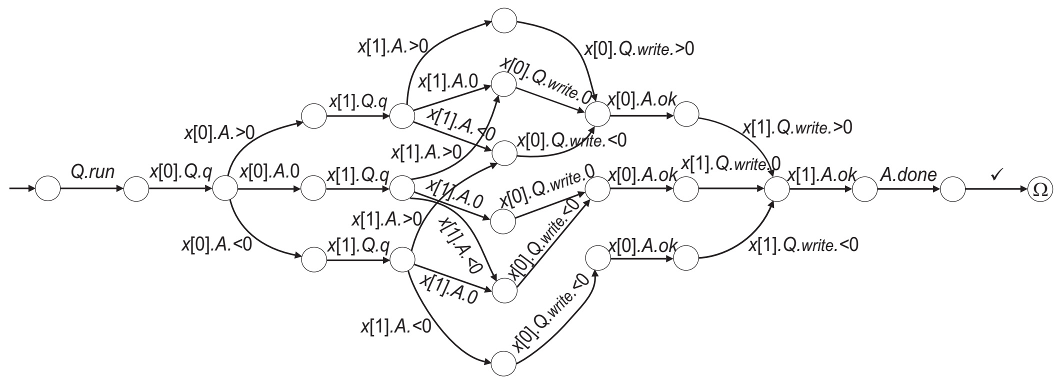


Figure 4.5: LTS for AIA_2 bubble sort with $k = 2$

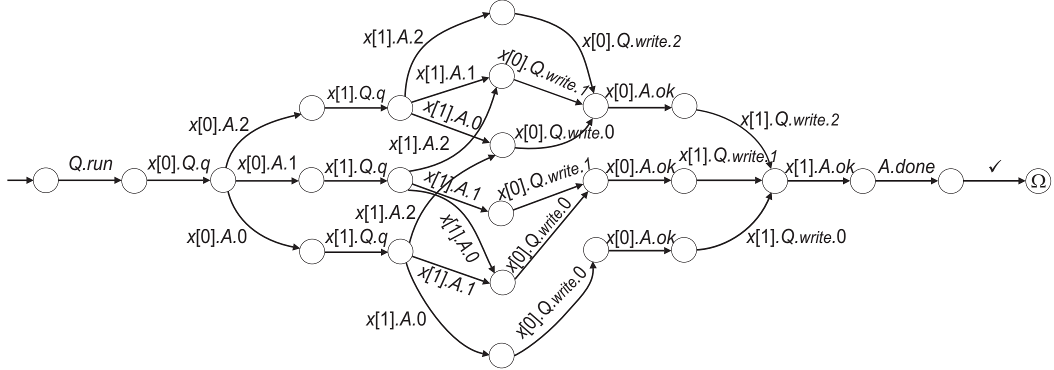


Figure 4.6: LTS for IA_2 bubble sort with $k = 2$

elements is int_3, i.e. it contains 3 distinct values, and the type of the index i becomes int_{k+1}. The model of the IA_2 bubble-sort is shown in Figure 4.6.

Table 4.14 contains the experimental results for model generation. The experiment consisted of running the compiler on the bubble-sort implementation, and then letting FDR generate a transition system (model) for the resulting process. The latter stage involved a number of hierarchical compressions, as outlined above. The transition system with maximum number of states involved in the generation of the final model is referred to as the largest generated transition system. We list the execution time in minutes, the size of the largest generated transition system, and the size of the final transition system. We ran FDR on a Research Machine AMD Athlon 64(tm) Processor 3500^+ with 2GB RAM. The results from the tool based on regular languages were obtained on a SunBlade 100 with 2GB RAM [10]. The one extra state in the CSP models is the special terminated state Ω. The CSP approach yields better results in time and space. This is firstly due to composition of strategies being represented in CSP using the renaming, parallel composition and hiding operators, and FDR being highly optimised for verification of such networks of processes. Secondly, FDR builds the models gradually, at each stage

k	CSP			Regular expressions		
	T(min)	Max	Model	T(min)	Max	Model
5	4	1 775	164	5	3 376	163
10	13	21 015	949	10	64 776	948
15	35	115 125	2 859	120	352 448	2 858
20	70	378 099	6 394	240	1 153 240	6 393
30	390	5 204 232	20 339	failed		

Table 4.14: Model generation of IA_2 bubble sort

compressing the subterm models.

Further information about model generation for $k = 20$ is shown in Figure 4.7. FDR first produces a transition system for the subprogram which is the scope of the declaration of the local array a. Each component of a, which is indexed from 0 to 19, is represented by the process $U_{\mathsf{int}}(a.i, 0)$ (see Table 4.10). FDR obtains the final model by taking the transition system for the scope of a and composing it with transition systems for the components of a in turn. At each step, compression algorithms are applied. In the figure, we show numbers of states before and after compression, after every two steps. The largest generated transition system in this case is obtained after composing the transition system for the component $a.13$ (or $a[13]$) with the compressed transition system for its scope.

We now turn to verifying absence of out-of-bounds errors. Let us modify the IA_2 bubble sort by replacing $k - 1$ in line 9 in Figure 4.4 by k, which introduces an out-of-bounds error. Table 4.15 shows some experimental results for checking the safety of this term. We did not apply compressions after composing the last component of a with the rest of the program. Instead, a composite model is generated on-the-fly during refinement checking. This

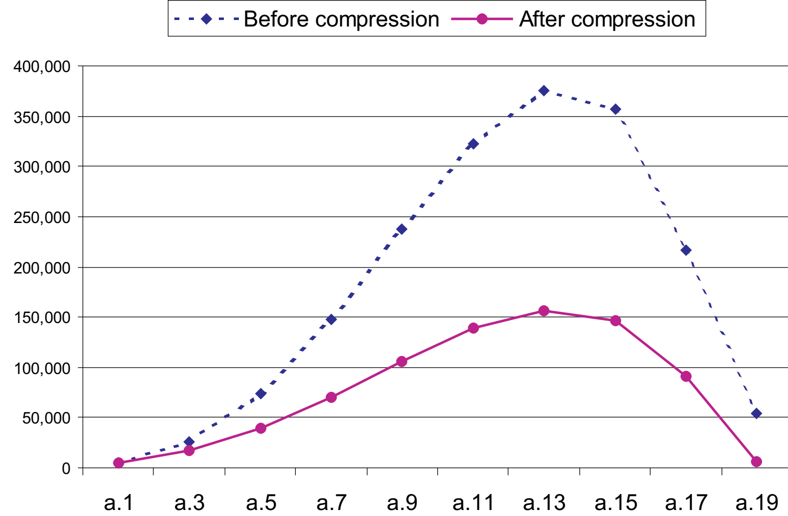


Figure 4.7: Effects of compressions for IA_2 bubble sort with $k = 20$

k	Total(min)	Spec	Impl	Check
29	250.5	10	240	0.5
30	317.5	12	305	0.5
31	494.2	12.5	391	0.7

Table 4.15: Checking safety for an erroneous IA_2 bubble sort

enabled us to check the property for array size 31, although the model generation did not succeed for this size. The times shown in Table 4.15 are: total execution time needed for this check, time to process the specification, time to process the implementation, and time to check refinement. They are all in minutes.

An abstract data type implementation Figure 4.8 contains an implementation of a stack of maximum size k (a meta variable). There are four free iden-

```
1   empty : com, overflow : com, p : exp int[],
2   ANALYSE(com, exp int[]) : com ⊢
3   new_int[] buffer[k] := 0 in
4   new_int[0,k] top := 0 in
5   let com push(exp int[] x) {
6     if (!top = k) then overflow
7       else {buffer[!top] := !x;  top := !top + 1}
8   } in
9   let exp int[] pop {
10    if (!top = 0) then empty
11      else {top := !top − 1;  return !buffer[!top + 1]}
12  } in
13  ANALYSE(push(p), pop)
14  : com
```

Figure 4.8: A stack implementation

tifiers: commands empty and overflow, expression p, and command ANALYSE which takes two arguments. After implementing the stack by a sequence of local declarations, we export the functions push(x) and pop by calling ANALYSE with arguments push(p) and pop. In effect, the model contains all interleavings of calls to push(p) and pop, corresponding to all possible behaviours of the non-local expression p and non-local function ANALYSE. The CSP script produced by our compiler for this example is provided in Appendix A.

A transition system for $k = 2$ is shown in Figure 4.9. For clarity, labels push and pop are used instead of ANALYSE.1 and ANALYSE.2.

By replacing empty with abort command (resp. overflow with abort), we can check separately for 'empty', i.e. reads from empty stacks, (resp. 'overflow', i.e. writes to full stacks) errors, both of which are present for any k. For 'empty' (resp. 'overflow') error, the counterexample traces which the FDR de-

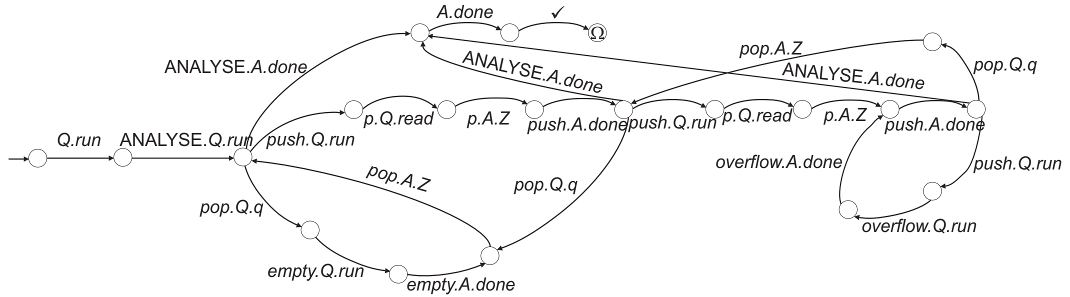


Figure 4.9: LTS for the stack with $k = 2$

bugger gives correspond to: a single call of `pop` method (resp. $k+1$ consecutive calls of `push` method) after which `abort` is executed.

In addition to checking properties of external behaviors of given terms, we can also check assertions which refer to local data. Assertions can be added to a term using a local function `assert` whose argument is a boolean expression. If the argument is true, the `assert` function does nothing, but otherwise it calls `abort`. For example, we can check whether, the last value pushed onto the stack is the value at the top of the stack. We replace all $\mathsf{int}_{[]}$ abstractions in Figure 4.8 by the more refined one $\mathsf{int}_{[0,0]}$, and the call to `ANALYSE` in line 13 by the following code:

```
let com assert(expbool b) {
    if b then skip else abort;
} in
let expbool validate() {
    new int[0,0] y := p in
    push(y);
    return (pop = y);
} in
ANALYSE(push(p), pop, assert(validate()))
: com
```

The assertion check fails and the generated counterexample reports that an error is caused by pushing a value onto a full stack. Indeed, if the stack is already full, pushing a new element will be ignored, and the overflow command is called. In that case, the top value in the stack will be different from the last pushed one.

Chapter 5

Abstraction Refinement

Abstraction refinement has proved to be one of the most effective methods of automatic verification of systems with very large state spaces, especially software systems. Current state-of-the art tools implementing abstraction refinement algorithms [17, 66, 24] combine model checking and theorem proving: model checking is used to verify whether an abstracted program satisfies a property, while theorem proving is used to refine the abstraction using the counterexamples discovered by model checking. Since abstractions are conservative over-approximations the safety of any abstracted program implies the safety of the concrete program. The converse is not true, and the refinement process may not terminate if the concrete program has an infinite state space.

This chapter introduces a purely semantic (syntax-independent) approach to (data) abstraction refinement, based on game semantics. The verification procedure, which applies to programs which can contain infinite integer types, is illustrated in Figure 5.1. It checks whether a program fragment is unsafe, i.e. it may execute the designated unsafe command abort.

The procedure starts by transforming the concrete (input) program into

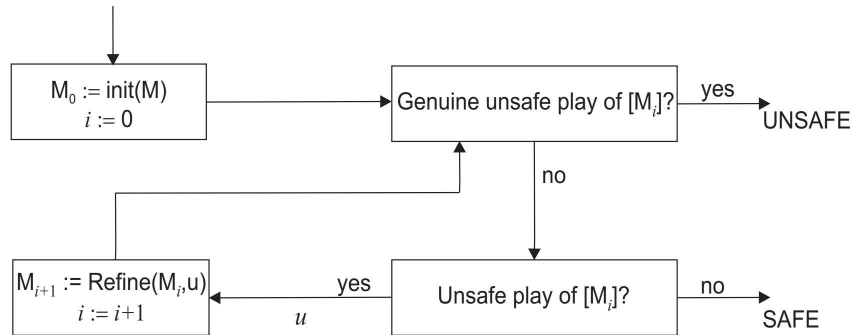


Figure 5.1: Verification procedure

the most abstract version of it, where all infinite integer types are abstracted to the coarsest abstraction, and the game-semantic model of the finitely abstracted program is fed to a model checker. Since our abstractions are safe, any abstracted program is an over-approximation of the concrete program. If no counterexample is found, the procedure terminates with answer SAFE. Otherwise, the counterexamples are analysed and classified as either *genuine*, which correspond to execution traces in the concrete program, or *potentially spurious*, which can be introduced due to abstraction. If genuine counterexamples exist the program is deemed UNSAFE, otherwise the spurious counterexamples are used to refine the abstractions. The procedure is then repeated on the refined abstracted program. The abstraction refinement procedure is a semi-algorithm: it terminates and reports a genuine counterexample for unsafe programs, but it may diverge for safe programs.

The following is a simple example illustrating this procedure. Consider the (concrete) program fragment below, which uses a local variable x and a non-local function f. Is this program safe for all safe instantiations of f, or is it possible for its execution to terminate abnormally?

$$\mathsf{new}_{\mathsf{int}}\, x := 0 \,\mathsf{in}\, f\big(x := !x + 1, \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort}\big)$$

The program is not safe if function f uses its first argument two or more times, then its second argument.

We approximate the set of integers by a finite set of partitioning intervals. Let the initial abstraction have only one partition. The initial abstracted program is:

$$\mathsf{new}_{\mathsf{int}[]}\ x := 0\ \mathsf{in}\ f\big(x := !x + 1, \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort}\big)$$

A counterexample execution trace exists, corresponding to the function evaluating its second argument. During the execution of this argument, the value of x is not 0 but, because of the abstraction, possibly any integer, chosen nondeterministically. If the chosen value is greater than 1 then abort occurs. Of course, this counterexample is spurious because it is made possible only by the nondeterminism caused by over-abstraction. However, the counterexample informs the refinement procedure that the abstraction of x needs to be improved. Iterations like this one are performed until we obtain

$$\mathsf{new}_{\mathsf{int}[0,1]}\ x := 0\ \mathsf{in}\ f\big(x := !x + 1, \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort}\big)$$

at which point a genuine counterexample is discovered, corresponding to the behaviour resulting in abnormal termination.

The abstraction refinement procedure we described uses *interaction plays*, where all internal moves are not hidden, for interpreting potentially spurious counterexamples and computing refined abstractions for the next iteration. This makes the procedure highly inefficient for implementation since it is necessary to use game models where all internal moves are exposed. Here, we describe a tool, called GameChecker, which implements efficiently the procedure by first identifying counterexample *standard* plays, where all internal moves are hidden, and then obtaining corresponding *interaction* plays by

"uncovering" the hidden moves. The tool is based on representing game models in the CSP process algebra, which can be verified for safety using the FDR refinement checker [1]. We can exploit an FDR debugging feature which allows identification of hidden events only in counterexample traces rather than in full models, in order to implement the "uncovering" operation necessary to compute interaction plays efficiently.

5.1 Interaction Game Semantics

In *standard* game semantics, which is presented in Chapter 3, to obtain the strategy $\sigma \fatsemi \tau : A \Rightarrow C$, the strategies $\sigma : A \Rightarrow B_1$ and $\tau : B_2 \Rightarrow C$ are composed, and moves which interact are hidden. (Here B_1 and B_2 are games for types which have equal concretisations.)

Let us define an alternative semantics, where moves which interact are not hidden. Consider composing $\sigma : A \Rightarrow B_1$ and $\tau : B_2 \Rightarrow C$ to obtain $\sigma \fatsemi_{int} \tau : A \Rightarrow C$. Let $r_1 \in B_1$ and $r_2 \in B_2$ be two interacting moves, then they are both recorded in $\sigma \fatsemi_{int} \tau$. Indeed, since we only have $\widetilde{B_1} = \widetilde{B_2}$, r_1 and r_2 may be different. However, if types interpreted by B_1 and B_2 do not contain abstractions, i.e. they are not types of integer expressions or integer variables, then $B_1 = B_2$ and $r_1 = r_2$. In such cases, we may record r_1 and r_2 only once, for readability.

We call this the *interaction game semantics*, and its building blocks interaction plays and interaction strategies.

For any term $\Gamma \vdash M : T$ of AIA, its interaction semantics is denoted $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$, and it can be easily reconciled with its standard game semantics,

[1] FDR is a commercial product of Formal Systems (Europe) Ltd. It is available free of charge for academic use. See `http://www.fsel.com`.

by performing all the hiding at once:

$$[\![\Gamma \vdash M : T]\!] = \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle \upharpoonright [\![\Gamma]\!], [\![T]\!] \tag{5.1}$$

where $- \upharpoonright [\![\Gamma]\!], [\![T]\!]$ indicates restriction to the games corresponding to base types occurring in Γ and T.

Standard plays are alternating sequences of Opponent and Player moves. Interaction plays in addition contain internal moves, which do not interact in subsequent compositions, but which record all intermediate steps taken during the computation.

Example Consider the interaction strategy of the term given in the Example on page 66:

$$\langle\!\langle x : \mathsf{varint}_{[0,4]} \vdash x := !x +_{[0,3]} 1_{[0,1]} : \mathsf{com} \rangle\!\rangle$$

One of its complete interaction plays, corresponding to the second standard play in the Example on page 66, is:

$$run\; q_2\; q_{2,1}\; q_{2,1,1}\; read^x\; 3^x\; 3_{2,1,1}\; 3_{2,1,1}\; 3_{2,1}\; 3_{2,1}\; q_{2,2}\; 1_{2,2}\; 1_{2,2}$$
$$(>3)_2\; (>3)_2\; write(>4)_1\; write(>4)^x\; ok^x\; ok_1\; done$$

We use tags on internal moves to precisely identify the coordinates of the subterm that corresponds to each move. For instance, $q_{2,1}$ is the question to the subterm $!x$, which is the 1st immediate subterm of $!x + 1$, which in turn is the 2nd immediate subterm of $x := !x + 1$. Observe also the double occurrences of integer internal moves, in line with how interaction plays are composed. In this example, those pairs are equal because, in any composition, any two corresponding abstractions are equal. An abstract value needs to be converted to another abstraction only within the strategy for assignment,

where a value with abstraction $[0, 3]$ is assigned to a variable with abstraction $[0, 4]$. ∎

The interaction game semantics, rather than the standard semantics, will be used for the purpose of abstraction refinement. The reason is that, given an unsafe standard play of an abstracted term, it does not in general contain sufficient information to decide that it can be produced by the concrete version of the term (i.e. that it is a genuine counterexample), or to choose one or more abstractions to be refined for the next iteration.

In traditional, state-based abstraction-refinement an abstract counterexample to a safety property is guaranteed to be genuine if the computation was deterministic (or, at least, the nondeterminism was not caused by over-abstraction). In standard game semantics, however, all internal steps within a computation are hidden. This results in standard strategies of abstracted terms in general not containing all information about sources of their nondeterminism.

Example Consider the following abstracted term:

$$\vdash \mathsf{new}_{\mathsf{int}[]}\, x := 0\, \mathsf{in}\, \mathsf{if}\, (!x \neq 0)\, \mathsf{then}\, \mathsf{abort} : \mathsf{com}$$

Its complete standard plays are $run \cdot abort$ and $run \cdot done$. In fact, its strategy is the same as the strategy of the EIAA term abort or skip. However, the counterexample $run \cdot abort$ is spurious, and the abstraction of x needs to be refined, but internal moves which point to this abstraction as the source of nondeterminism have been hidden. ∎

5.2 Conservativity of Abstraction

For safety, we want to show that a finitely abstracted program is a conservative over-approximation of the concrete program, i.e. if the abstracted program is safe then the concrete program is also safe.

For abstractions π and π', we say that π' *refines* π if, for any partition (i.e. abstracted value) c' of π', there exists a unique partition c of π such that $c' \subseteq c$. We say that c is the *corresponding abstracted value* of c' in π. When π' refines π, and c is a partition of π, we say that π' *splits* c if c is not a partition of π'. We extend the *refine* relation to data types as follows: bool *refines* bool, and $\mathsf{int}_{\pi'}$ *refines* int_{π} if π' refines π.

Definition Let types D'_1, D'_2 and D' refine D_1, D_2 and D respectively. We say that an abstracted operation $\mathsf{op} : \mathsf{exp}D'_1 \times \mathsf{exp}D'_2 \rightarrow \mathsf{exp}D'$ is *safely approximated* by abstracted operation $\mathsf{op} : \mathsf{exp}D_1 \times \mathsf{exp}D_2 \rightarrow \mathsf{exp}D$ iff for every c'_1, c'_2, c' of type D'_1, D'_2, D' and c_1, c_2 of type D_1, D_2 respectively, if $c'_1 \subseteq c_1$, $c'_2 \subseteq c_2$ and $c' \in \mathsf{op}_{D'_1 \times D'_2 \rightarrow D'}(c'_1, c'_2)$, then there exists a unique c of type D such that $c \in \mathsf{op}_{D_1 \times D_2 \rightarrow D}(c_1, c_2)$ and $c' \subseteq c$.

Example Abstraction $[0, 1]$ refines $[0, 0]$ such that $[0, 1]$ splits >0 and >0 is the corresponding abstracted value of 1 and >1 in $[0, 0]$. ▮

We note that abstracted programs may contain nondeterministic branching, because the outcome of integer conversions and arithmetic-logic operations might not be a unique value. As interaction plays contain internal moves, we can distinguish those whose underlying computation did not pass through any nondeterministic branching that is due to abstraction.

Definition (a) Given integer abstractions π and π', and an abstracted value (i.e. partition) c of π, we say that converting c to π' is *deterministic* if there exists an abstracted value c' of π' such that $c \subseteq c'$.

(b) Given an abstracted operation $\mathsf{op} : \mathsf{exp}D_1 \times \mathsf{exp}D_2 \rightarrow \mathsf{exp}D$ and abstracted values c_1 and c_2 of type D_1 and D_2 respectively, we say that the application of op to c_1 and c_2 is *deterministic* if there exists an abstracted value c of type D such that $\forall\, v_1 \in c_1, v_2 \in c_2,\ v_1 \,\mathsf{op}\, v_2 \ \in\ c$[2].

(c) An interaction play $u \in \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$ is *deterministic* if each conversion of an abstracted integer value in u is deterministic, and each application of an arithmetic-logic operator in u is deterministic.

We say that a term $\Gamma' \vdash M' : T'$ *refines* a term $\Gamma \vdash M : T$ if $\widetilde{\Gamma'} = \widetilde{\Gamma}$, $\widetilde{M'} = \widetilde{M}$, $\widetilde{T'} = \widetilde{T}$, and each abstraction in $\Gamma' \vdash M' : T'$ refines the corresponding abstraction in $\Gamma \vdash M : T$. For any play t of $\Gamma' \vdash M' : T'$, let $\overline{t}$ denote the image play of $\Gamma \vdash M : T$ (see Section 3.8), obtained by replacing each abstracted integer c' in t by its partition c (such that $c' \subseteq c$) in the corresponding abstraction in $\Gamma \vdash M : T$.

Theorem 5.2.1 *Suppose $\Gamma' \vdash M' : T'$ refines $\Gamma \vdash M : T$.*

(i) For any $t \in [\![\Gamma' \vdash M' : T']\!]$, we have $\overline{t} \in [\![\Gamma \vdash M : T]\!]$. The same is true for the $\langle\!\langle - \rangle\!\rangle$ semantics.

(ii) For any deterministic $u \in \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$, there exists $t \in \langle\!\langle \Gamma' \vdash M' : T' \rangle\!\rangle$ such that $u = \overline{t}$ [3].

[2] Here we regard the abstracted values tt and ff as singleton sets $\{tt\}$ and $\{ff\}$.

[3] *This can be strengthened to apply to interaction plays which are deterministic with respect to the abstractions in $\Gamma' \vdash M' : T'$. The latter notion allows nondeterministic conversions of, and operator applications to, abstracted values which are not split by the corresponding abstractions in $\Gamma' \vdash M' : T'$.*

Proof By induction on the typing rules of AIA. We only consider the most interesting cases which involve integer abstractions. Proofs for other cases are similar. We prove (i) for the $\langle\!\langle - \rangle\!\rangle$ semantics since the proof for the $[\![-]\!]$ semantics follows from the former.

Consider the case of any constant v. We have that $q \cdot c' \in \langle\!\langle \Gamma' \vdash v : \mathsf{exp}D' \rangle\!\rangle$ and $q \cdot c \in \langle\!\langle \Gamma \vdash v : \mathsf{exp}D \rangle\!\rangle$, such that $v \in c'$, $v \in c$ and $c' \subseteq c$. Moreover, $q \cdot c$ is deterministic and $q \cdot c = \overline{q \cdot c'}$.

Consider the case of any arithmetic-logic operator op. Let $t \in \langle\!\langle \Gamma' \vdash M' \,\mathsf{op}_{D'}\, N' : \mathsf{exp}D' \rangle\!\rangle$ be of the form $q \cdot q_1 \cdot c_1' \cdot q_2 \cdot d_2' \cdot e'$ and let $e' \in \mathsf{op}_{D_1' \times D_2' \to D'}(c', d')$. From the induction hypothesis, there exist partitions c of type D_1 and d of type D_2 such that $q \cdot c \in \langle\!\langle \Gamma \vdash M : \mathsf{exp}D_1 \rangle\!\rangle$, $q \cdot d \in \langle\!\langle \Gamma \vdash N : \mathsf{exp}D_2 \rangle\!\rangle$, and $c' \subseteq c$, $d' \subseteq d$. From the safety of $\mathsf{op} : \mathsf{exp}D_1 \times \mathsf{exp}D_2 \to \mathsf{exp}D$, there exists e of type D such that $e \in \mathsf{op}_{D_1 \times D_2 \to D}(c, d)$ and $e' \subseteq e$. Then, $\overline{t} = q \cdot q_1 \cdot c_1 \cdot q_2 \cdot d_2 \cdot e \in \langle\!\langle \Gamma \vdash M \,\mathsf{op}_D\, N : \mathsf{exp}D \rangle\!\rangle$.

Let $u = q \cdot q_1 \cdot c_1 \cdot q_2 \cdot d_2 \cdot e$ be a deterministic play in $\langle\!\langle \Gamma \vdash M \,\mathsf{op}_D\, N : \mathsf{exp}D \rangle\!\rangle$, where $e = \mathsf{op}_{D_1 \times D_2 \to D}(c, d)$ (i.e. $\forall\, v_1 \in c, v_2 \in d,\ v_1 \,\mathsf{op}\, v_2 \in e$). From the induction hypothesis and the safety of $\mathsf{op} : \mathsf{exp}D_1 \times \mathsf{exp}D_2 \to \mathsf{exp}D$, there must be partitions $c' \subseteq c$, $d' \subseteq d$ and $e' \subseteq e$ of type D_1',D_2' and D' respectively, such that $q \cdot c' \in \langle\!\langle \Gamma' \vdash M' \rangle\!\rangle$, $q \cdot d' \in \langle\!\langle \Gamma' \vdash N' \rangle\!\rangle$, and $e' \in \mathsf{op}_{D_1' \times D_2' \to D'}(c', d')$. Then, $t = q \cdot q_1 \cdot c_1' \cdot q_2 \cdot d_2' \cdot e' \in \langle\!\langle \Gamma' \vdash M' \,\mathsf{op}_{D'}\, N' : \mathsf{exp}D' \rangle\!\rangle$ and $u = \overline{t}$.

Consider the case of assignment. Let $t \in \langle\!\langle \Gamma' \vdash M' := N' : \mathsf{com} \rangle\!\rangle$ be of the form $run \cdot q_2 \cdot d_2' \cdot write(c')_1 \cdot ok_1 \cdot done$, where c', d' are of type D_1', D_2' respectively and $d' \cap c' \neq \emptyset$. From the induction hypothesis, there exist partitions d of type D_2 and c of type D_1 such that $q \cdot d \in \langle\!\langle \Gamma \vdash N : \mathsf{exp}D_2 \rangle\!\rangle$, $write(c) \cdot ok \in \langle\!\langle \Gamma \vdash M : \mathsf{var}D_1 \rangle\!\rangle$, and $c' \subseteq c$, $d' \subseteq d$. Then, $\overline{t} = run \cdot q_2 \cdot d_2 \cdot write(c)_1 \cdot ok_1 \cdot done \in \langle\!\langle \Gamma \vdash M := N : \mathsf{com} \rangle\!\rangle$ and $d \cap c \neq \emptyset$.

Let $u = run \cdot q_2 \cdot d_2 \cdot write(c)_1 \cdot ok_1 \cdot done$ be a deterministic play in $\langle\langle \Gamma \vdash M := N : \mathsf{com} \rangle\rangle$, where $d \subseteq c$. From the induction hypothesis, there must be partitions $c' \subseteq c$ and $d' \subseteq d$ of type D_1' and D_2' respectively, such that $q \cdot d' \in \langle\langle \Gamma' \vdash N' \rangle\rangle$, $write(c') \cdot ok \in \langle\langle \Gamma' \vdash M' \rangle\rangle$, and $c' \cap d' \neq \emptyset$. Then, $t = run \cdot q_2 \cdot d_2' \cdot write(c')_1 \cdot ok_1 \cdot done \in \langle\langle \Gamma' \vdash M' := N' : \mathsf{com} \rangle\rangle$ and $u = \overline{t}$. ∎

The following consequence of Corollary 3.7.3, Theorem 5.2.1 and the correspondence between standard and interaction game semantics (5.1) will justify the correctness of the abstraction refinement procedure.

Corollary 5.2.2 *Suppose* $\Gamma' \vdash M' : T'$ *refines* $\Gamma \vdash M : T$*.*

(i) If $[\![\Gamma \vdash M : T]\!]$ *is safe, then* $\Gamma' \vdash M' : T'$ *is safe.*

(ii) If $\langle\langle \Gamma \vdash M : T \rangle\rangle$ *has a deterministic unsafe interaction play, then* $\Gamma' \vdash M' : T'$ *is unsafe.*

5.3 Abstraction Refinement

Recall from Section 2.1 that an abstraction π is *finitary* if it has finitely many partitions, and a term is *finitely abstracted* if it contains only finitary abstractions. Apart from the identity abstraction κ, observe that the abstractions we work with, $[]$ and $[n, m]$ where $n \leq 0 \leq m$, are finitary.

We have shown in Section 4.2 that for any finitely abstracted term $\Gamma \vdash M : T$ of AIA_2, the set $[\![\Gamma \vdash M : T]\!]$ is a regular language and an automaton which recognises it is effectively constructible. We can show that the same is true for the $\langle\langle - \rangle\rangle$ semantics. To obtain an automaton for $\langle\langle \Gamma \vdash M : T \rangle\rangle$, the construction is the same as for $[\![\Gamma \vdash M : T]\!]$ except that interacting moves are tagged with subterm coordinates rather than hidden.

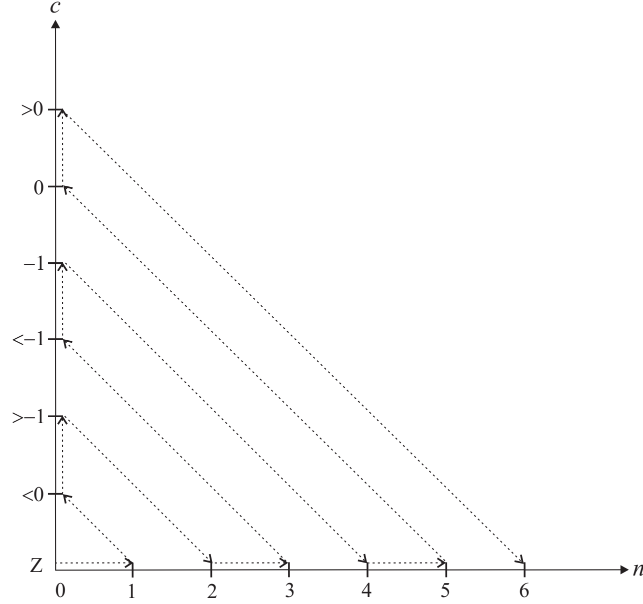


Figure 5.2: A possible definition of $\sqsubseteq$

Let $A[\![\Gamma \vdash M : T]\!]$ and $A\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$ denote the automata which recognise $[\![\Gamma \vdash M : T]\!]$ and $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$ respectively. Since there is no hiding in the construction of $A\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$ (see the Example on page 112), this automaton is deterministic.

Given a finite word u and a deterministic automaton A which accepts u, we call u *cycle-free* if the accepting run visits any state of A at most once.

Let $\prec$ denote the following computable linear ordering between abstracted values:

$$\mathbb{Z} \prec (<0) \prec (>-1) \prec (<-1) \prec -1 \prec 0 \prec (>0) \prec \cdots$$
$$(<-(n+1)) \prec -(n+1) \prec n \prec (>n) \prec \cdots$$

This ordering has the property that $c \prec c'$ whenever $c' \subset c$. For two moves (possibly tagged with subterm coordinates) r and r' which are equal except

for containing different abstracted integer values c and c', let $r \prec r'$ if $c \prec c'$, and $r' \prec r$ if $c' \prec c$. Now, we extend this ordering to a computable linear ordering on all moves (in an arbitrary but fixed way), and denote it by $\prec$. Let $\prec$ also denote the linear orderings on plays obtained by lifting the linear ordering on moves lexicographically.

Let $(n, c) \sqsubseteq (n', c')$ be any computable linear ordering between pairs of nonnegative integers and abstracted integer values which is obtained by extending the partial ordering defined by $n \leq n'$ and $c \preceq c'$, and which admits no infinite strictly decreasing sequences, and no infinite strictly increasing sequences bounded above (e.g., see Figure 5.2). For any play u, let $|u|$ denote its length, and $\max(u)$ denote the $\prec$-maximal abstracted integer value in u (or $\mathbb{Z}$ if there is no such value). Let $u \sqsubseteq u'$ mean $(|u|, \max(u)) \sqsubseteq (|u'|, \max(u'))$. Now, let $\trianglelefteq$ be the linear ordering between plays such that $u \trianglelefteq u'$ if and only if either $u \sqsubset u'$, or $|u| = |u'|$, $\max(u) = \max(u')$ and $u \preceq u'$.

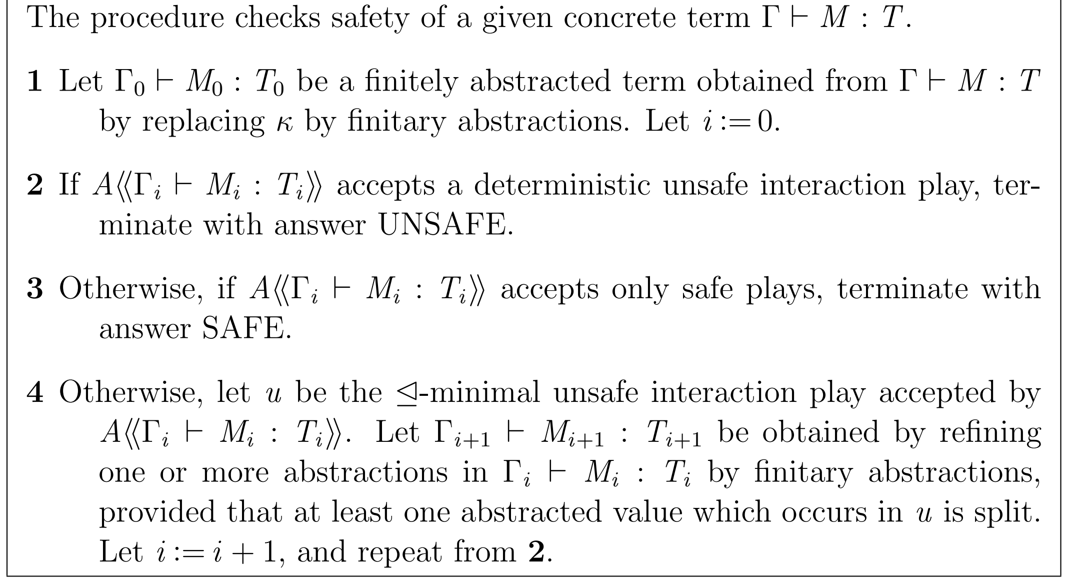
Lemma 5.3.1 *In the linear order of all plays with respect to* $\trianglelefteq$*:*

(i) there is no infinite strictly decreasing sequence;

(ii) there is no infinite strictly increasing sequence which is bounded above.

Proof This is due to the following two facts. Firstly, the $\sqsubseteq$ ordering between pairs of nonnegative integers and abstracted integer values has the properties (i) and (ii). Secondly, for any such pair (n, c), there are only finitely many plays u such that $|u| = n$ and $\max(u) = c$. ∎

The abstraction refinement procedure (ARP) is given in Figure 5.3. Note that, in Step **1**, the initial abstractions can be chosen arbitrarily; and in Step **4**, arbitrary abstractions can be refined in arbitrary ways, as long as the



The procedure checks safety of a given concrete term $\Gamma \vdash M : T$.

1 Let $\Gamma_0 \vdash M_0 : T_0$ be a finitely abstracted term obtained from $\Gamma \vdash M : T$ by replacing κ by finitary abstractions. Let $i := 0$.

2 If $A\langle\langle \Gamma_i \vdash M_i : T_i \rangle\rangle$ accepts a deterministic unsafe interaction play, terminate with answer UNSAFE.

3 Otherwise, if $A\langle\langle \Gamma_i \vdash M_i : T_i \rangle\rangle$ accepts only safe plays, terminate with answer SAFE.

4 Otherwise, let u be the $\trianglelefteq$-minimal unsafe interaction play accepted by $A\langle\langle \Gamma_i \vdash M_i : T_i \rangle\rangle$. Let $\Gamma_{i+1} \vdash M_{i+1} : T_{i+1}$ be obtained by refining one or more abstractions in $\Gamma_i \vdash M_i : T_i$ by finitary abstractions, provided that at least one abstracted value which occurs in u is split. Let $i := i + 1$, and repeat from **2**.

Figure 5.3: Abstraction refinement procedure

refinement splits at least one abstracted value in u. These two choices do not affect correctness and semi-termination, but they allow experimentation with different heuristics in concrete implementations. Also, in Step **4**, arbitrary unsafe plays can be chosen. In this case, the procedure is still correct, but we cannot guarantee the semi-termination.

Theorem 5.3.2 *ARP is well-defined and effective. If it terminates with SAFE (UNSAFE, respectively), then* $\Gamma \vdash M : T$ *is safe (unsafe, respectively).*

Proof For well-defined-ness, Lemma 5.3.1 (i) ensures that the $\trianglelefteq$-minimal unsafe interaction play u accepted by $A\langle\langle \Gamma_i \vdash M_i : T_i \rangle\rangle$ always exists. Since the condition in Step **2** was not satisfied, u is not deterministic. Therefore, u cannot contain only singleton abstracted values, so there is at least one abstracted value in u which can be split.

Effectiveness follows from the fact that it suffices to consider cycle-free plays in Step **4**, and from the computability of $\trianglelefteq$.

If ARP terminates with SAFE (UNSAFE, respectively), then $\Gamma \vdash M : T$ is safe (unsafe, respectively) by Corollary 5.2.2, since any abstraction is refined by the identity abstraction κ. ∎

Theorem 5.3.3 *If $\Gamma \vdash M : T$ is unsafe then ARP will terminate with UNSAFE.*

Proof By Corollary 3.7.3 and the correspondence (5.1), there exists an unsafe interaction play $t \in \langle\langle \Gamma \vdash M : T \rangle\rangle$.

For each i, let U_i be the set of all unsafe $u \in \langle\langle \Gamma_i \vdash M_i : T_i \rangle\rangle$, and let $u_i^\dagger$ be the $\trianglelefteq$-minimal element of U_i.

It follows by Theorem 5.2.1 that, for any $u \in \langle\langle \Gamma_{i+1} \vdash M_{i+1} : T_{i+1} \rangle\rangle$, $\overline{u} \in \langle\langle \Gamma_i \vdash M_i : T_i \rangle\rangle$. Also, we have $\overline{u} \trianglelefteq u$, since they have the same length and $\overline{c} \preceq c$ for any c. Now, Step **4** ensures that, for any i, $u_i^\dagger \notin \langle\langle \Gamma_{i+1} \vdash M_{i+1} : T_{i+1} \rangle\rangle$.

Therefore, $u_0^\dagger \lhd u_1^\dagger \lhd \cdots u_i^\dagger \lhd \cdots$. But, for each i, $u_i^\dagger \trianglelefteq \overline{t}^i \trianglelefteq t$. By Lemma 5.3.1 (ii), ARP must terminate for $\Gamma \vdash M : T$! ∎

ARP may diverge for safe terms. This is generally the case with abstraction refinement methods since the underlying problem is undecidable. A simple example is the term

$$e : \mathsf{expint} \vdash \mathsf{new}_{\mathsf{int}}\, x := e \,\mathsf{in}\, \mathsf{if}\, (!x = !x + 1)\, \mathsf{then}\, \mathsf{abort} : \mathsf{com}$$

This term is safe, but any finitely abstracted term obtained from it is unsafe.

5.4 Implementation

We have seen so far how counterexample guided abstraction refinement ideas can be adapted to the setting of game semantics models (see Section 5.3).

However, implementing the procedure above is non-trivial because the semi-algorithm, as described, is highly inefficient. In this section, we describe GameChecker, a model checking tool which implements efficiently an abstraction refinement procedure for verifying safety properties of software, such as assertion violations, buffer overruns or array-out-of-bounds errors. The procedure is guaranteed to terminate for unsafe inputs.

GameChecker is available from:
`http://www.dcs.warwick.ac.uk/~aleks/gamechecker.htm`.

5.4.1 Representing Game Models in CSP

GameChecker includes a compiler which translates any finitely abstracted term $\Gamma \vdash M : T$ into a CSP process $[\![\Gamma \vdash M : T]\!]^{CSP} u_\Gamma$, where u_Γ maps free identifiers to copy-cat processes, whose set of finite even-length traces $\mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash M : T]\!]^{CSP} u_\Gamma)$ is the set of all plays of the game strategy for the term. Those processes are defined compositionally, by induction on the structure of terms (see Section 4.3).

The abstraction refinement procedure described in Section 5.3 requires models consisting of *fully revealed plays*, i.e., models in which semantic composition of strategies does not involve *hiding* of the moves involved in composition. The fully revealed plays allow us to discern between genuine and spurious counterexamples by identifying the precise subterms that produce abstracted moves. However, fully revealed models are much larger and therefore impractical. In GameChecker, this is overcome as follows: first we use special marker moves to identify points in plays at which abstraction gives rise to nondeterminism, then we use a special debugging feature of FDR that lets us reveal only those plays which are counterexamples rather than full models.

Nondeterminism due to abstraction happens when an arithmetic-logic operation or a conversion produces more than one result. In such an instance, the operation necessarily has at least one abstracted integer operand which is not a singleton, i.e. which abstracts more than one integer. The game strategy for the operation then performs a special marker move $nd.c$, where c is such an operand. Those moves are propagated through strategy compositions, so for any term $\Gamma \vdash M : T$, they appear in $\mathsf{traces}^{\mathsf{ev}}([\![\Gamma \vdash M : T]\!]^{CSP} u_\Gamma)$ at the points where nondeterminism due to abstraction occurs.

Example Consider $[\![x : \mathsf{var\,int}_{[0,4]} \vdash x := !x{+}1 : \mathsf{com}]\!]$. If the abstract value <0 is read from x, $!x + 1$ can evaluate to both 0 and <0. The following complete play corresponds to choosing the result 0:

$$run\ read^x\ {<}0^x\ nd.({<}0)\ write(0)^x\ ok^x\ done$$

The move $nd.({<}0)$ records the non-singleton abstracted integer operand <0. ∎

FDR offers a number of state-space reduction algorithms which preserve finite-trace sets, and which are thus compositional. The processes representing the game strategies are particularly amenable to such reductions, because moves which are hidden through composition of strategies become internal (τ) process transitions. The compiler within GameChecker inserts calls to FDR's state-space reduction algorithms within the process scripts it outputs.

5.4.2 Implementing Abstraction Refinement Procedure

GameChecker checks safety of a given concrete term $\Gamma \vdash M : T$ (with infinite integer data types) by performing a sequence of iterations. The initial

abstracted term $\Gamma_0 \vdash M_0 : T_0$ uses the coarsest abstraction $[\,]$ for any free identifier or local variable, and the abstraction $[0, n]$ or $[n, 0]$ for constants n. Other abstractions (such as those for integer expression subterms) are determined from the former by inference.

Each iteration consists of model checking (by calling the FDR tool), slicing [103], and refining abstractions. Only abstractions which occur in types of free identifiers or local variables are explicitly refined, and others are obtained by inference. That yields a refined abstracted term $\Gamma_{i+1} \vdash M_{i+1} : T_{i+1}$, which is passed to the next iteration.

The following are the steps of any iteration. If t is a trace which contains at least one special move marking a nondeterminism, let $\mathbf{pick}(t) = c$, where $nd.c$ is the first such move [4]. For ordering non-singleton abstracted integers, we use a bijection r to the natural numbers: $r(\mathbb{Z}) = 0$, $r(<n) = 2|n| + 2$, and $r(>n) = 2n + 1$. This has the property that $r(c) < r(c')$ whenever $c' \subseteq c$.

1 If $[\![\Gamma_i \vdash M_i : T_i]\!]^{CSP} u_\Gamma \setminus \{|nd|\}$ is unsafe, terminate with answer UNSAFE.

2 If $[\![\Gamma_i \vdash M_i : T_i]\!]^{CSP} u_\Gamma$ is safe, terminate with answer SAFE.

3 Among the counterexamples (i.e. traces of $[\![\Gamma_i \vdash M_i : T_i]\!]^{CSP} u_\Gamma$ which end in *abort*), select t such that $r(\mathbf{pick}(t))$ is minimal [5].

4 Apply the FDR trace-reveal feature to t, obtaining a fully revealed trace s.

5 Call a slicing procedure to determine a set S of all occurrences of non-singleton abstracted integers which were involved in causing the first

[4] This definition of $\mathbf{pick}(t)$ is currently implemented, but other definitions are possible. The crucial property is that, if t is used to refine abstractions, then one of the refinements will split $\mathbf{pick}(t)$.

[5] We implemented also a procedure which selects an arbitrary counterexample. It is correct, but it might not terminate for unsafe terms.

$nd.c$ move in s.

6 For any data type int_π of a free identifier or a local variable which corresponds to an occurrence of an abstracted integer b in S, refine π by splitting b.

Steps **2** and **3** are implemented as follows. The process $[\![\Gamma_i \vdash M_i : T_i]\!]^{CSP} u_\Gamma$ is composed in parallel with an auxiliary process *Rank_of_pick* which, once the first move of the form $nd.c$ has occurred, keeps in its state the value $r(c)$. FDR is called to model check that parallel composition, and for any reachable state which has an *abort* transition , to return a trace which reaches it. By Step **1**, any such trace must contain an $nd.c$ move. The parallel composition with *Rank_of_pick* ensures that, for any possible value of $r(\mathbf{pick}(t))$ with t a counterexample, at least one such counterexample is returned by FDR.

Step **5** is implemented as follows. The procedure $\mathsf{slice}(i)$ uses as a global parameter s, an interaction unsafe play, and takes as argument i, the index of the previous move in s to the first $nd.c$ move. It returns a set of moves S, which are involved in computing $s(i)$:

1 If move $s(i)$ is an answer in $\mathsf{exp}D$, which contains an occurrence of non-singleton abstracted integer, then:

(i) if $s(i)$ corresponds to a free identifier then return $s(i)$.

(ii) if $s(i)$ corresponds to an arithmetic or logic operation op then return $\mathtt{slice}(k) \cup \mathtt{slice}(i-1)$, where k is the index of the answer of the

first operand:

$$\begin{array}{lllllllllll} s = \cdots & q & q & \cdots & m & \cdots & q & \cdots & m' & n & \cdots \\ & & & & k & & & & i-1 & i & \end{array}$$

where $n = m \,\mathsf{op}\, m'$.

(iii) if $s(i)$ corresponds to the sequencing operator then return slice$(i-1)$, where

$$\begin{array}{lllllllllll} s = \cdots & q & run & \cdots & done & \cdots & q & \cdots & m' & m & \cdots \\ & & & & & & & & i-1 & i & \end{array}$$

(iv) if $s(i)$ corresponds to the if operator then return slice$(k)\cup$slice$(i-1)$, where k is the index of the answer of the guard:

$$\begin{array}{lllllllllll} s = \cdots & q & q & \cdots & b & \cdots & q & \cdots & m' & m & \cdots \\ & & & & k & & & & i-1 & i & \end{array}$$

(v) if $s(i)$ corresponds to a de-referencing then return slice$(i-1)$, where

$$\begin{array}{lllllll} s = \cdots & q & read & \cdots & m' & m & \cdots \\ & & & & i-1 & i & \end{array}$$

2 if $s(i)$ is an answer in var$D$, then:

(i) if $s(i)$ corresponds to a free variable and $s(i)$ is a non-singleton abstracted integer m, i.e. an answer to *read*, then return $s(i)$.

(ii) if $s(i)$ corresponds to a local variable and $s(i)$ is a non-singleton abstracted integer m, i.e. an answer to *read*, then
return slice(k), where k is the index of the *last write* operation on the variable:

$$\begin{array}{lllllllll} s = \cdots & write(m') & \cdots & ok & \cdots & read & \cdots & m & \cdots \\ & & & k & & & & i & \end{array}$$

or return $s(i)$ if there is no write operation on the variable before i.

3 if $s(i)$ is an answer in `com` then:

(i) if $s(i-1)$ is *ok* and it corresponds to an assignment to a local variable, then return $\texttt{slice}(k)$, where k is the answer of the right-hand side of the assignment:

$$\begin{array}{lllllllllll} s = \cdots & run & q & \cdots & m' & \cdots & write(m) & \cdots & ok & done & \cdots \\ & & & & k & & & & i-1 & i & \end{array}$$

(ii) otherwise, return $\emptyset$.

4 otherwise, return $\emptyset$.

The slicing procedure can be considered as an optimisation. The ARP will work correctly, even if we always refine all identifiers that occur in the interaction trace prior to the first nondeterminism. However, the size of the generated models will increase very rapidly.

Theorem 5.4.1 *If the abstraction refinement procedure terminates, its answer is correct. Moreover, it terminates for any unsafe term.*

Proof UNSAFE answers are correct because any trace which contains no special moves marking nondeterminism corresponds to a concrete trace. Correctness of SAFE answers is a consequence of the conservativity of abstraction (see Section 5.2).

Suppose $\Gamma \vdash M : T$ is unsafe. Let s be a fully revealed unsafe play of the game strategy for $\Gamma \vdash M : T$, and let m be an integer in s with maximum absolute value. For any non-singleton abstracted integer c, we define $d(c) = 2|m| + 1 - r(c)$. Then $d(c) > 0$ whenever $|n| \leq |m|$ for some $n \in c$.

For any iteration i, let D_i be the sum of all positive $d(c)$ as c ranges over the non-singleton partitions of all abstractions in $\Gamma_i \vdash M_i : T_i$. Steps **3**–**6** ensure that $D_0 > D_1 > \cdots > 0$, so the procedure must terminate. ∎

5.4.3 Using the Tool

GameChecker has been developed in Java [15].

The front end can be seen in Figure 5.4. The inputs are a concrete term, given using C-like syntax [6], and a property, given as an unsafe command whose executability will be checked. The default unsafe command is `abort`. The result pane shows the iteration steps, including all reported nondeterministic counterexamples and applied refinements. If the procedure terminates, it is reported whether the term is safe or unsafe. In the latter case, a genuine counterexample is returned.

By default, GameChecker executes a simpler abstraction refinement procedure than the one presented in Subsection 5.4.2, where any shortest counterexample is selected in Step **3**. This variant is more efficient per iteration, but it might not terminate for unsafe terms. The semi-terminating procedure is run by checking the Semi-Termination box in the Options menu. Figure 5.5 shows the tool architecture.

5.4.4 Examples

The following are three progressively more involved examples. Further examples can be found at the GameChecker website:
`http://www.dcs.warwick.ac.uk/~aleks/gamechecker.htm`.

[6] We prefer the C-like syntax for the sake of its presumed familiarity to most users.

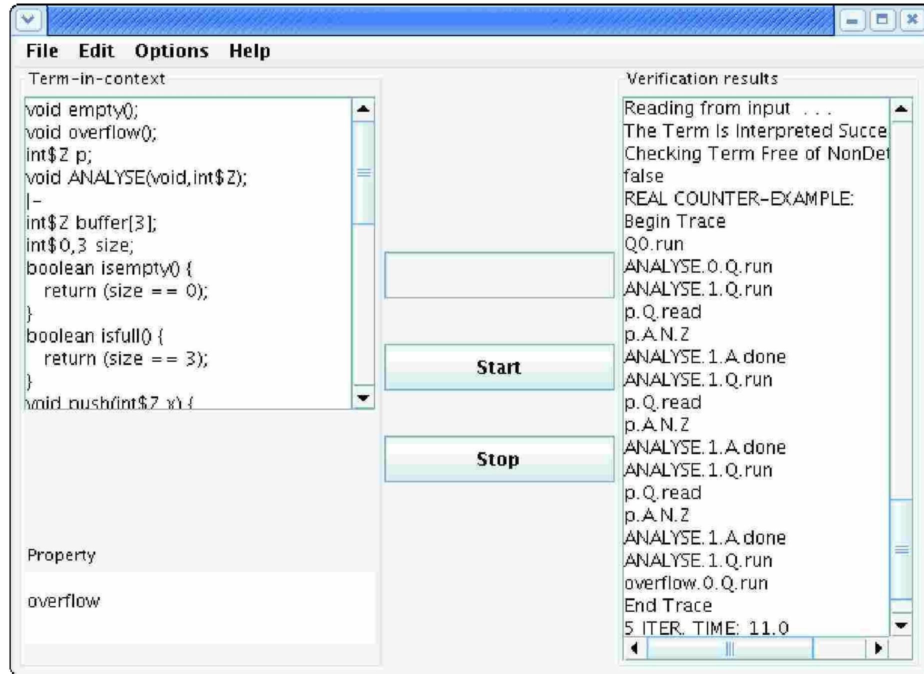


Figure 5.4: A screen-shot of the tool

A warm-up example

Consider the term

$$f(\mathsf{com}, \mathsf{com}) : \mathsf{com} \vdash \mathsf{new}_{\mathsf{int}}\, x := 0 \;\mathsf{in}\; f\big(x := !x + 1, \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort}\big)$$

which uses a local variable x, and a *non-local* function f. The program is not safe, and this is how GameChecker will discover the bug.

The initial abstracted term is

$$\mathsf{new}_{\mathsf{int}[\,]}\, x := 0 \;\mathsf{in}\; f\big(x := !x + 1, \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort}\big)$$

A nondeterministic counterexample is identified by FDR, corresponding to a function that evaluates its second argument:

$$run\ run^{f}\ run^{f,2}\ nd.\mathbb{Z}\ abort$$

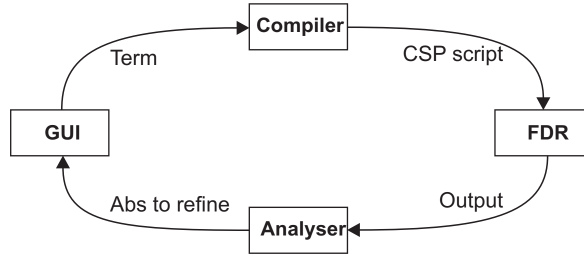


Figure 5.5: The tool architecture

Next, by processing the detailed output from FDR, we reconstruct a corresponding fully revealed play:

$$\begin{aligned} &run\ run^f\ run^{f,2}\ run_2\ q_{2,1}\ q_{2,1,1}\ read_{2,1,1,1}\ read^x\ \mathbb{Z}^x \\ &\qquad \mathbb{Z}_{2,1,1,1}\ \mathbb{Z}_{2,1,1}\ q_{2,1,2}\ 1_{2,1,2}\ tt_{2,1}\ nd.\mathbb{Z} \end{aligned}$$

The slicing procedure starts by examining the subterm with coordinates $\langle 2,1\rangle$, whose answer move precedes nd. The coordinates specify the path in the syntax tree of the term to reach the subterm, in this case the boolean expression of if. The nd move marks that this term has been evaluated non-deterministically. Since this subterm represents a logic operation, the slicing procedure is called recursively to examine its operands, i.e. terms $\langle 2,1,1\rangle$ and $\langle 2,1,2\rangle$. The answer move of the $\langle 2,1,1\rangle$ term is the abstract value $\mathbb{Z}$, so the examination proceeds for its subterm $\langle 2,1,1,1\rangle$. Here, it will be detected that the $\langle 2,1,1,1\rangle$ term is a de-referencing of x and that the abstract value $\mathbb{Z}$ is read from x. Thus, the slicing procedure will indicate that the abstraction of x needs to be refined.

The second iteration uses the refined term:

$$\mathsf{new}_{\mathsf{int}[0,0]}\ x := 0\ \mathsf{in}\ f\big(x := !x + 1, \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort}\big)$$

Another nondeterministic counterexample is found, which represents a func-

tion evaluating its first and then its second argument:

$$run\ run^f\ run^{f,1}\ done^{f,1}\ run^{f,2}\ nd.(>0)\ abort$$

The corresponding fully revealed play is:

$$\begin{aligned} run\ run^f\ run^{f,1}\ run_1\ q_{1,2}\ q_{1,2,1}\ read_{1,2,1,1}\ read^x\ 0^x\ 0_{1,2,1,1}\ 0_{1,2,1} \\ q_{1,2,2}\ 1_{1,2,2}\ 1_{1,2}\ write(1)_{1,1}\ write(>0)^x\ ok^x\ ok_{1,1} \\ ok_1\ done^{f,1}\ run^{f,2}\ run_2\ q_{2,1}\ q_{2,1,1}\ read_{2,1,1,1}\ read^x\ (>0)^x \\ (>0)_{2,1,1,1}\ (>0)_{2,1,1}\ q_{2,1,2}\ 1_{2,1,2}\ tt_{2,1}\ nd.(>0) \end{aligned}$$

Similarly as in the previous iteration, the analyser starts exploring the nondeterministic term $\langle 2, 1\rangle$. By searching for non-singleton abstracted integers recursively through its subterms, it will detect that the abstract value >0 read from x has caused the nondeterminism. So, further refinement of x will be recommended.

The third iteration term is:

$$\mathsf{new}_{\mathsf{int}[0,1]}\ x := 0\ \mathsf{in}\ f(x := !x + 1, \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort})$$

Now, a genuine unsafe trace is detected: f increments x twice, then evaluates its second argument:

$$run\ run^f\ run^{f,1}\ done^{f,1}\ run^{f,1}\ done^{f,1}\ run^{f,2}\ abort$$

The model generated for the third iteration term is shown in Figure 4.1. Observe that, in this case, the model contains no nd moves.

A semi-termination example

The term

$$e : \mathsf{expint}, f(\mathsf{com}, \mathsf{com}) : \mathsf{com} \vdash$$
$$\mathsf{new_{int}}\ x := e\ \mathsf{in}\ \mathsf{new_{int}}\ y := 0\ \mathsf{in}$$
$$\mathsf{if}\ (!x = !x + 1)\ \mathsf{then\ abort\ else}\ f\big(y := !y + 1, \mathsf{if}\ (!y > 1)\ \mathsf{then\ abort}\big)$$

is an example of an unsafe term for which the simpler abstraction refinement procedure, where any shortest counterexample is selected in Step **3**, does not terminate: it keeps refining the abstraction of x. If the tool is instructed to perform the semi-terminating procedure presented in Section 5.4.2, then after a few iterations of refining the abstraction of x, the abstraction of y will be refined. For this particular example, a genuine counterexample is reported after refining the abstractions of e and x to $[0, 1]$, and the one for y to $[0, 1]$.

A stack example

Consider the following implementation of a stack of maximum size k (see also Figure 4.8).

$$\mathsf{empty} : \mathsf{com}, \mathsf{overflow} : \mathsf{com}, p : \mathsf{exp\,int},$$
$$\mathtt{ANALYSE}(\mathsf{com}, \mathsf{exp\,int}) : \mathsf{com} \vdash$$
$$\mathsf{new_{int}}\ \mathtt{buffer}[k] := 0\ \mathsf{in}\ \mathsf{new_{int}}\ \mathtt{top} := 0\ \mathsf{in}$$
$$\mathsf{let\ com}\ \mathtt{push}(\mathsf{exp\ int}\ x)\ \{$$
$$\quad \mathsf{if}\ (!\mathtt{top} = k)\ \mathsf{then\ overflow\ else}\ \{\mathtt{buffer}[!\mathtt{top}] := !x;\ \ \mathtt{top} := !\mathtt{top} + 1\}\}\ \mathsf{in}$$
$$\mathsf{let\ exp\ int\ pop}\ \{$$
$$\quad \mathsf{if}\ (!\mathtt{top} = 0)\ \mathsf{then\ empty\ else}\ \{\ \mathtt{top} := !\mathtt{top} - 1;\ \ \mathsf{return}\ !\mathtt{buffer}[!\mathtt{top} + 1]\}\}\ \mathsf{in}$$
$$\mathtt{ANALYSE}(\mathtt{push}(p), \mathtt{pop})$$

	empty		overflow		oub	
k	Iterations	Time	Iterations	Time	Iterations	Time
5	2	0.3	7	1.2	7	1.5
10	2	0.9	12	5.6	12	8
15	2	2	17	18	17	23
20	2	4	22	47	22	59
30	2	11	32	190	32	230
50	2	60	52	1570	52	1831
100	2	630	failed		failed	

Table 5.1: Experimental results for checking a stack implementation

We can check a range of properties of the stack implementation. By replacing empty with abort and overflow with abort command, we can check separately for 'empty' (reads from empty stacks) and 'overflow' (writes to full stacks) errors, both of which are present for any k. We can also check that array-out-of-bounds errors are not present in the term: arrays are syntactic sugar, in which abort is executed if an array is referenced out of its bounds.

Table 5.1 contains the experimental results for checking the three properties on the stack implementation. We ran GameChecker on a Research Machine AMD Athlon 64(tm) Processor 3500$^+$ with 2GB RAM. We list the number of iterations and the execution times in minutes for different values of k. Abstraction and abstraction-refinement are crucial in generating models that are small enough to be analysed, by ensuring that the contents of the array can be disregarded (the initial abstraction $[\,]$ is not refined), and that the local variable tracking the top of the stack (`top`) is automatically adjusted to a small but safe domain (in practice, $[0,0]$ for 'empty', and $[0,k]$ for 'overflow' and 'oub' errors).

Chapter 6

Compositional Verification

One of the main problems in model checking is the *state explosion problem* [32]: the explored system states need to be stored in memory, which may be prohibitively large for realistic systems. Given that the state explosion problem is particulary acute in software model checking, the most desirable feature of this approach is *scalability*. *Compositional modelling and verification* (e.g., [60]) achieve scalability by breaking up a larger software system into smaller systems which can be modelled and verified independently. Hence, the properties of a program can be established from the properties of its individually checked components (subprograms) without requiring to check the whole program as an atomic "flat" entity.

Game semantics meets the first requirement for achieving scalability: *compositional modelling*. Game semantics is denotational, i.e. defined recursively on the syntax, therefore the model of a larger program is constructed from the models of its subprograms, using a notion of strategy composition.

Assume-guarantee reasoning [75, 93] addresses the second challenge: *compositional verification*. To check that a property P is satisfied by a model

M composed of two components M_1 and M_2, it suffices to find an assumption (model) A such that

1. the composition of M_1 and A satisfies P, and

2. M_2 is a refinement of A

If such an assumption A can be found and it is significantly smaller than M_2, then we can verify whether M satisfies P (by checking 1 and 2) without having to build the whole M. Developing such an assumption A is not trivial.

In this chapter, we describe an automatic procedure which generates assumptions as above using the L^* algorithm for learning a regular language. L^* iteratively learns a minimal deterministic finite automaton, which represents the desired assumption, from membership and equivalence queries. In each iteration, L^* produces a candidate assumption A which is used to check 1 and 2. Depending on results of the checks, we may conclude that the required property is satisfied, or violated in which case a witness counterexample is reported, or the current A needs to be revised. The learning-based approach to automatic assumption generation builds the assumption incrementally guided by the queries, and if it finds an appropriate assumption the procedure will stop and use it to prove the property. This procedure is set within an abstraction refinement loop which automatically extracts a game-semantic model from a data-abstracted program and refines the program if a spurious counterexample is found.

We have implemented this approach in the GameChecker tool (see Section 5.4). We report some initial experimental results, which indicate significant memory savings compared to a non assume-guarantee approach.

The assume-guarantee paradigm is the best studied approach to compo-

sitional reasoning [29, 60, 67, 75, 93, 102]. The primary difficulty in applying this approach to realistic systems is that, in general, the appropriate assumptions have to be constructed manually.

The work presented in this chapter is motivated by a recently proposed approach [33], which uses learning algorithms to automate assume-guarantee reasoning. In [33], a variant of Angluin's L^* algorithm [14, 96] for learning a regular language is used to generate appropriate assumptions. Compared to this approach, which is applied at the design level of a software system, our work makes the following contributions.

- We apply the method at the implementation level, and verify safety properties of open program fragments.
- While in [33] the method is used for verifying multi-threaded programs by building models and checking their constituting threads independently, here we apply compositional verification on sequential programs where individually checked components can be arbitrary subprograms of the given input program.
- The L^* algorithm is adapted to the specific game semantics setting for learning a game assumption.
- The method is integrated with a counterexample-guided abstraction refinement style loop. We thus obtain a procedure which embodies both compositional modelling and compositional verification.

The L^* learning algorithm has found a number of applications in automatic verification. For example, adaptive model checking [59] uses learning to compute an accurate finite state model of an unknown system starting from

an approximate model; substitutability analysis of evolving software systems [25] verifies an upgraded software system by learning; [13] uses a symbolic implementation of the L^* algorithm for compositional reasoning about symbolic modules; [12] uses learning along with predicate abstraction in the context of interface synthesis, etc.

6.1 The Learning Algorithm

Central to our compositional verification procedure is an algorithm for learning assumptions, which can be represented as regular languages. We define an *assumption* for a game A as a prefix-closed non-empty set of even-length sequences which satisfy the alternation condition. The algorithm is an adaptation of the L^* algorithm introduced by Angluin [14] which learns an unknown regular language and produces an automaton that accepts it. Since L^* needs to learn assumptions, the adaptation will consider only non-empty prefix-closed sets of even-length sequences (words) in which Opponent and Player moves alternate, thus achieving greater efficiency.

Let $A = \langle M_A, \lambda_A, \vdash_A, P_A \rangle$ be a game. Let $\mathrm{O}^A = \{m \in M_A \mid \lambda_A^{\mathrm{OP}}(m) = \mathrm{O}\}$ and $\mathrm{P}^A = \{m \in M_A \mid \lambda_A^{\mathrm{OP}}(m) = \mathrm{P}\}$ denote the sets of *Opponent* and *Player* moves in A, respectively. Since λ_A is a total function, $\{\mathrm{O}^A, \mathrm{P}^A\}$ is a partition of M_A. Given that the sequences from an assumption for a game A satisfy the alternation condition, it follows that they are sequences from $(\mathrm{O}^A \mathrm{P}^A)^*$.

Let α be an unknown assumption for a game A. L^* iteratively learns the structure of α using assistance from a *Teacher* who can answer two kinds of questions about α:

Membership query Given a sequence s from $(\mathrm{O}^A\mathrm{P}^A)^*$, the Teacher answers *true* if $s \in \alpha$, and *false* otherwise.

Equivalence query Given a DFA (Deterministic Finite Automaton) D, the Teacher replies that D is either correct, when $\mathcal{L}(D) = \alpha$, or incorrect, and in the latter case gives a counterexample which is a sequence in the symmetric difference of $\mathcal{L}(D)$ and α.

The basic data structure of the L^* algorithm is a two-dimensional table, called observation table (S, E, T), which keeps information about a finite collection of sequences over $(\mathrm{O}^A\mathrm{P}^A)^*$, classified as members or non-members of α. S is a *prefix-closed* set of even-length sequences (and thus include the empty sequence ε), $E \subseteq (\mathrm{O}^A\mathrm{P}^A)^*$ is a *suffix-closed* set of even-length sequences (and thus include ε as well), and T is a function mapping $(S \cup S \cdot \mathrm{O}^A\mathrm{P}^A) \cdot E \rightarrow \{true, false\}$, such that:

$$\forall\, s \in S \cup S \cdot \mathrm{O}^A\mathrm{P}^A . \forall\, e \in E : T(s, e) = true \Leftrightarrow s \cdot e \in \alpha$$

The rows of the table are the elements of $(S \cup S \cdot \mathrm{O}^A\mathrm{P}^A)$, while the columns are the elements of E. Finally T denotes the table entries.

Let us define a function $row(s)$ for any $s \in S \cup S \cdot \mathrm{O}^A\mathrm{P}^A$ as follows:

$$\forall\, e \in E : row(s)(e) = T(s, e)$$

A table is *closed* if for each $s \cdot m_O m_P \in S \cdot \mathrm{O}^A\mathrm{P}^A$ such that $T(s, \varepsilon) = true$, there is some $s' \in S$ such that $row(s') = row(s \cdot m_O m_P)$. A table is *consistent* if for each $s, s' \in S$ such that $row(s) = row(s')$, either $T(s, \varepsilon) = T(s', \varepsilon) = false$, or for each $m_O m_P \in \mathrm{O}^A\mathrm{P}^A$, we have that $row(s \cdot m_O m_P) = row(s' \cdot m_O m_P)$. Note that if the table is not consistent, then there are $s, s' \in S$, $m_O m_P \in \mathrm{O}^A\mathrm{P}^A$,

and $e \in E$, such that $row(s) = row(s')$ and $T(s \cdot m_O m_P, e) \neq T(s' \cdot m_O m_P, e)$. In this case we can add $m_O m_P \cdot e$ to E in order to make $row(s) \neq row(s')$.

We define an equivalence relation $\equiv$ over sequences in $S \cup S \cdot \mathrm{O}^A\mathrm{P}^A$ such that $s \equiv s'$ iff $row(s) = row(s')$. Denote by $[s]$ the equivalence class which includes s. Given a closed and consistent table (S, E, T), L^* constructs a candidate DFA $D = (Q, q_0, \mathrm{O}^A\mathrm{P}^A, \delta)$ as follows: $Q = \{[s] \mid s \in S, T(s, \varepsilon) = true\}$, $q_0 = [\varepsilon]$, and for every $s \in S$ and $m_O m_P \in \mathrm{O}^A\mathrm{P}^A$, the transition from $[s]$ on input $m_O m_P$ is enabled iff $T(s \cdot m_O m_P, \varepsilon) = true$ and then $\delta([s], m_O m_P) = [s \cdot m_O m_P]$. For example, see Figure 6.5 for a table and its candidate DFA. The facts that the table is closed and consistent guarantee that the transition relation is well-defined. All states in the automaton are accepting, since the language we learn is prefix closed. Note that every transition in this automaton is labelled by two-letters sequence: an Opponent and a Player move.

Figure 6.1 contains the L^* algorithm. Each iteration of this algorithm starts with either a table with $S = E = \{\varepsilon\}$, or a table which was prepared in the previous iteration. Then T is updated using membership queries until the table is consistent and closed. If the table is not consistent, E is increased with a suffix which replaces the inconsistent equivalence class with two new classes. If the table is not closed, then S is increased with sequences that represent missing equivalence classes. Next a candidate automaton D is proposed and an equivalence query with D is made. If the answer for the equivalence query is *true*, i.e. $\mathcal{L}(D) = \alpha$, L^* terminates and returns the automaton D. Otherwise, L^* analyses the counterexample c reported by the Teacher and adds all even-length prefixes of c to S. Then, a new iteration is started.

Let n be the number of states of the minimal DFA M equivalent to the unknown language we learn. It was shown in [14, Theorem 1] that the

let $L^*(S, E)$ **be**
  repeat :
    Update T using queries
    while (S, E, T) is not consistent **or** not closed **do**
      if (S, E, T) is not consistant **then**
        find $s \in S,\ m_O m_P \in \mathrm{O}^A\mathrm{P}^A,\ e \in E$:
          $row(s) = row(s')$ and $T(s \cdot m_O m_P, e) \neq T(s' \cdot m_O m_P, e)$
        $E = E \cup \{m_O m_P \cdot e\}$
        Update T using queries
      if (S, E, T) is not closed **then**
        find $s \in S,\ m_O m_P \in \mathrm{O}^A\mathrm{P}^A$
          $s \cdot m_O m_P \notin [t],$ for all $t \in S$
        $S = S \cup \{s \cdot m_O m_P\}$
        Update T using queries
    $D = \mathrm{MakeAutomaton}(S, E, T)$
    if D is correct **then**
      return D
    else
      let c be reported counterexample
      foreach ($s \in$ even_prefix(c) and $s \notin S$) $S = S \cup \{s\}$

Figure 6.1: The L^* algorithm for learning assumptions

candidate automata made by L^* strictly increase in size, i.e. each candidate automaton must have at least one more state than the one from the previous iteration, and all incorrect candidates are smaller than M. Hence, L^* is guaranteed to construct M using at most $n-1$ equivalence queries (i.e. iterations). It was also shown in [14] that L^* terminates in time polynomial in n and the length of the longest counterexample provided by the Teacher.

Each new call to L^* starts normally with $S = E = \{\varepsilon\}$. But in cases where a previously learned assumption (language) exists (and hence a table), we want to start the algorithm for learning a new modified assumption by reusing the information proposed in the previous table. Thus with this *dynamic version* of L^*, we try to speed up the learning by reusing the previously

inferred sets S and E for assumption α, to learn a new modified assumption α' which differs slightly from α. We apply this optimisation using the fact that if L^* starts with any non-empty *valid table* (i.e. valid function T) then it will terminate with a correct result [25, Theorem 2]. A table is said to be valid if the answers to the membership queries for all sequences in the table are correct with respect to the unknown assumption α' which is learned by L^*, i.e. $\forall\, s \in S \cup S \cdot \mathrm{O}^A\mathrm{P}^A . \forall\, e \in E : T(s,e) = \mathit{true} \Leftrightarrow s \cdot e \in \alpha'$.

We can apply some further optimizations to the L^* algorithm specific for the languages we learn. A prefix closed language has the property that extensions of rejected sequences are rejected, i.e., if $s \notin \alpha$, then no extension of s is in α. Therefore, since the language we learn is prefix closed, before any membership query $s \in \alpha$, we first test whether it is an extension of a sequence already observed to be rejected. If so, we add the result immediately to the table.

6.2 Compositional Verification

In this section we describe in detail the compositional verification procedure which combines assume-guarantee reasoning and abstraction refinement.

6.2.1 Overview

We first examine how the game semantics of β-normal AIA_2 terms $\Gamma \vdash M : B$ is obtained. Since terms are interpreted recursively over the typing rules, consider a derivation tree of such a term $\Gamma \vdash M : B$. At the leaves, we have base subterms, which are language constants and free identifiers, and are interpreted by appropriate constant and copy-cat strategies. At each node,

there is a subterm obtained by a language construct c from some children subterms $M_1, \ldots, M_n$. Then, $\mathsf{c}(M_1, \ldots, M_n)$ is interpreted by composing the interpretations of the subterms and of the construct σ_{c}:

$$[\![\mathsf{c}(M_1, \ldots, M_n)]\!] = ([\![M_1]\!], \ldots, [\![M_n]\!]) \fatsemi \sigma_{\mathsf{c}} = ([\![M_1]\!]^{\dagger}, \ldots, [\![M_n]\!]^{\dagger}) \,;\, \sigma_{\mathsf{c}}$$

We also note that the promotion operator $\dagger$ is applied only to strategies σ for games of the form $[\![\Gamma]\!] \Rightarrow [\![B']\!]$, where B' are base types. The games $[\![B']\!]$ are flat, i.e. all their questions are initial and Player moves can only be answers. So $\sigma^{\dagger}$ consists of iterated plays of σ, such that a new play of σ can be started only when the previous one is completed. Basically, $\sigma^{\dagger}$ contains plays of the form $s_1 \ldots s_k s_{k+1}$ where each s_i is a play of σ and $s_1, \ldots, s_k$ are complete. That is a regular language operation, i.e. if σ is a regular language then $\sigma^{\dagger}$ is a regular language as well.

Now, for any strategies $\sigma_1, \ldots, \sigma_n$ and τ, we have $\big((\sigma_1^{\dagger}, \ldots, \sigma_n^{\dagger}) \,;\, \tau\big)^{\dagger} = (\sigma_1^{\dagger}, \ldots, \sigma_n^{\dagger}) \,;\, \tau^{\dagger}$ [3]. By thus distributing $\dagger$ over linear composition $;$, we conclude that the game semantics of $\Gamma \vdash M : B$ can be obtained by repeatedly applying $;$ to promoted strategies for base subterms and language constructs. For first-order free identifiers, the application strategy is first calculated and then $\dagger$ is applied to it. In other words, $\dagger$ does not need to be applied to any composite strategy, except for the application.

By the same argument, if $\Gamma' \vdash N : B'$ is a subterm of $\Gamma \vdash M : B$, the game semantics of $\Gamma \vdash M : B$ is given by:

$$[\![\Gamma \vdash M[N] : B]\!] = [\![\Gamma \vdash M[-] : B]\!]([\![\Gamma' \vdash N : B']\!]^{\dagger})$$

where $[\![\Gamma \vdash M[-] : B]\!](\sigma)$ is an operator on regular languages, which is obtained from the game semantic definitions for $\Gamma \vdash M : B$ by replacing the

promoted interpretation of the subterm $\Gamma' \vdash N : B'$ by σ, and in which only ; is applied to languages obtained from σ.

To check safety of $[\![\Gamma \vdash M[N] : B]\!]$, we use the concept of assume-guarantee (AG) reasoning. Recall that an assumption for a game A is a prefix-closed non-empty set of even-length sequences from $(\mathsf{O}^A\mathsf{P}^A)^*$.

Let α be an assumption for the game $[\![\Gamma']\!] \Rightarrow ![\![B']\!]$. We use the following AG rule:

$$\frac{\begin{array}{l}[\![\Gamma \vdash M[-] : B]\!](\alpha) \ \text{ is SAFE} \\ [\![\Gamma' \vdash N : B']\!]^\dagger \leq \alpha\end{array}}{[\![\Gamma \vdash M[N] : B]\!] \ \text{ is SAFE}}$$

The rule states that if there is an assumption α for $[\![\Gamma']\!] \Rightarrow ![\![B']\!]$, such that $[\![\Gamma \vdash M[-] : B]\!](\alpha)$ is safe and α is an abstraction of $[\![\Gamma' \vdash N : B']\!]^\dagger$, then $[\![\Gamma \vdash M[N] : B]\!]$ is safe. Our goal is to construct such an *appropriate assumption* α to show that $\Gamma \vdash M[N] : B$ is safe.

Theorem 6.2.1 *The AG rule is sound and complete.*

Proof By monotonicity of composition of strategies with respect to the $\leq$ ordering, we have that if $\alpha \leq \alpha'$ then $[\![\Gamma \vdash M[-] : B]\!](\alpha) \leq [\![\Gamma \vdash M[-] : B]\!](\alpha')$. To establish soundness, we use the fact that if α' is safe and $\alpha \leq \alpha'$ then α is also safe. Completeness follows by taking $\alpha = [\![\Gamma' \vdash N : B']\!]^\dagger$. ∎

For any operator $[\![\Gamma \vdash M[-] : B]\!]$, where the hole $-$ is in the place of a subterm of type $\Gamma' \vdash B'$, we define the *weakest safe assumption* $\alpha_W : [\![\Gamma']\!] \Rightarrow ![\![B']\!]$ as follows. Given an even-length sequence s of $(\mathsf{O}^{[\![\Gamma']\!] \Rightarrow ![\![B']\!]}\mathsf{P}^{[\![\Gamma']\!] \Rightarrow ![\![B']\!]})^*$, let τ_s be the set consisting of s and all its even-length prefixes. Let α_W consist of all sequences s such that $[\![\Gamma \vdash M[-] : B]\!](\tau_s)$ is safe.

By the definitions of $[\![\Gamma \vdash M[-] : B]\!]$ and $;$, we have that, for any strategy $\sigma : [\![\Gamma']\!] \Rightarrow ![\![B']\!]$,

$$[\![\Gamma \vdash M[-] : B]\!](\sigma) = \bigcup \{[\![\Gamma \vdash M[-] : B]\!](\tau_s) \mid s \in \sigma\}$$

Thus, $[\![\Gamma \vdash M[-] : B]\!](\sigma)$ is safe if and only $\sigma \leq \alpha_W$. The problem of finding an appropriate assumption α which satisfies both premises of the AG rule hence reduces to checking for a language which is subset of α_W and a superset of $[\![\Gamma' \vdash N : B']\!]^\dagger$. We use the L^* algorithm to learn such an assumption.

The verification procedure **CompVer** which uses the AG rule is presented in Figure 6.2. Given two terms $\Gamma \vdash M[-] : B$ and $\Gamma' \vdash N : B'$, it checks safety of $\Gamma \vdash M[N] : B$. The procedure uses an **AGCheck** algorithm, and iteratively performs the following steps:

1 Let $[\![\Gamma_0 \vdash M_0[-] : B_0]\!]$ and $[\![\Gamma'_0 \vdash N_0 : B'_0]\!]$ be obtained by data abstraction, and $S_0^0 = E_0^0 = \{\varepsilon\}$. Let $i := 0$.

2 Apply **AGCheck** on $[\![\Gamma_i \vdash M_i[-] : B_i]\!]$ and $[\![\Gamma'_i \vdash N_i : B'_i]\!]$, using S_i^0 and E_i^0. If the result is *true*, then terminate with answer SAFE. Otherwise, a counterexample c' is returned as well as updated values of S_i^k and E_i^k.

3 If c' is a nondeterministic (spurious) play, obtain $[\![\Gamma_{i+1} \vdash M_{i+1}[-] : B_{i+1}]\!]$ and $[\![\Gamma'_{i+1} \vdash N_{i+1} : B'_{i+1}]\!]$ by refining the abstractions in the current terms which were involved in causing the nondeterminism in c'. Set $S_{i+1}^0 = S_i^k$, $E_{i+1}^0 = E_i^k$ [1] and $i := i + 1$, and repeat from **2**.

4 Otherwise, c' is a deterministic (genuine) play and the procedure terminates with answer UNSAFE.

[1] If some sequences in S_i^k (E_i^k) contain abstracted values whose abstractions are refined, we replace them with sequences which are compatible with newly refined abstractions.

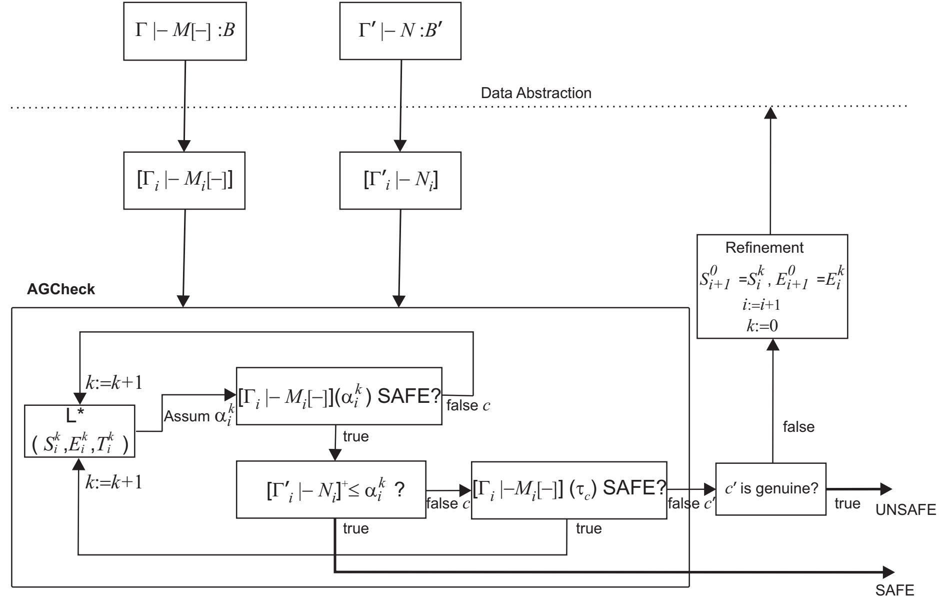


Figure 6.2: Compositional verification procedure

We say that a play is *nondeterministic* if it contains a special marker move *nd*, which identifies points in plays at which abstraction gives rise to nondeterminism. This happens when an arithmetic-logic operation or a conversion produces more than one result (see Section 5.4).

We continue by describing the **AGCheck** algorithm. Details of the data abstraction procedure and the abstraction refinement process can be found in Chapter 5.

6.2.2 Assume-Guarantee Algorithm

The **AGCheck** algorithm takes as inputs $[\![\Gamma_i \vdash M_i[-] : B_i]\!]$ and $[\![\Gamma'_i \vdash N_i : B'_i]\!]$ as well as S^0_i and E^0_i, and returns as answer *true* or a counterexample. **AGCheck** is actually the L^* algorithm given in Figure 6.1, where the membership and equivalence queries are answered using model checking. **AGCheck** proceeds as follows:

1 Generate a candidate assumption α_i^k using L^*.

2 If $[\![\Gamma_i \vdash M_i[-] : B_i]\!](\alpha_i^k)$ is not safe, then return a counterexample to the L^* algorithm, set $k := k+1$ and repeat from **1**.

3 If $[\![\Gamma'_i \vdash N_i : B'_i]\!]^\dagger \leq \alpha_i^k$ is true, terminate with answer *true*.

4 Otherwise, among the even-length counterexamples from $[\![\Gamma'_i \vdash N_i : B'_i]\!]^\dagger$, report a deterministic one, c. If one such does not exist, then report a nondeterministic one, c.

5 Generate a strategy τ_c from the sequence c which contains c and all its even-length prefixes. If $[\![\Gamma_i \vdash M_i[-] : B_i]\!](\tau_c)$ is safe, then report c to L^*, set $k := k+1$ and repeat from **1**.

6 Otherwise, terminate reporting a deterministic counterexample c'. If one such does not exist, report a nondeterministic play c'.

If in Step **2** a counterexample c is returned to L^*, then $c \in \alpha_i^k \backslash \alpha_W$, i.e. the current assumption α_i^k is too weak and it has to be *strengthened* by removing some sequences from it. Similarly, if in Step **5** a counterexample c is reported to L^*, then $c \in \alpha_W \backslash \alpha_i^k$, i.e. the current α_i^k must be *weakened* by adding some sequences. The result of such strengthening (resp., weakening) will be that at least the behaviour that the counterexample represents will be removed from (resp., allowed by) the next assumption α_i^{k+1}.

In the above procedure, L^* iteratively learns an appropriate assumption α, but the procedure terminates as soon as conclusive results are obtained. The Teacher which interacts with L^* is implemented using model checking. To answer a membership query for a sequence s, which is a play [2] of the

[2] Here we regard a play of a game as a valid play which satisfies visibility and bracketing conditions.

corresponding game, the Teacher first builds a strategy $\tau_s = \{s' \mid s' \sqsubseteq^{\text{even}} s\}$. The Teacher then model checks $[\![\Gamma \vdash M[-]]\!](\tau_s)$ for safety. If true is returned, then $s \in \alpha$ and the Teacher answers *true*, otherwise it answers *false*. The answer for all other sequences, which are not plays, is false. An equivalence query is answered by model-checking two premises of the AG rule in Steps **2** and **3**. If both checks succeed, then the answer is *true*, otherwise either a counterexample is reported to L^* or an unsafe counterexample is found.

Theorem 6.2.2 *Given* $[\![\Gamma_i \vdash M_i[-] : B_i]\!]$ *and* $[\![\Gamma'_i \vdash N_i : B'_i]\!]$*, the* **AGCheck** *algorithm is correct.*

Proof The algorithm returns true when both premises of the AG rule return true, and therefore correctness is guaranteed by the AG rule. An unsafe play is returned when there is a play s of $([\![\Gamma'_i \vdash N_i]\!])^{\dagger}$ which, when applied to $[\![\Gamma_i \vdash M_i[-]]\!]$ produces an unsafe play, which implies that $[\![\Gamma_i \vdash M_i[N_i] : B_i]\!]$ is not safe. ∎

Theorem 6.2.3 *If* **CompVer** *terminates, its answer is correct.*

Proof This follows from the correctness of the abstraction refinement procedure, which was shown in Chapter 5, and Theorem 6.2.2. ∎

6.2.3 Example

Consider the term

$$
\begin{aligned}
f : \mathsf{com} \to \mathsf{com} \ \vdash \quad & \mathsf{new}_{\mathsf{int}}\, x := 0 \,\mathsf{in} \\
& \quad f(x := x + 1)\,; \\
& \quad \mathsf{if}\ (x = 0)\ \mathsf{then\ abort};
\end{aligned}
$$

in which x is a local variable, and f is a non-local function. We want to check whether this term is safe from terminating abnormally for all safe instantiations of f. The program is not safe if function f does not use its argument at all. Its model is shown in Figure 6.3.

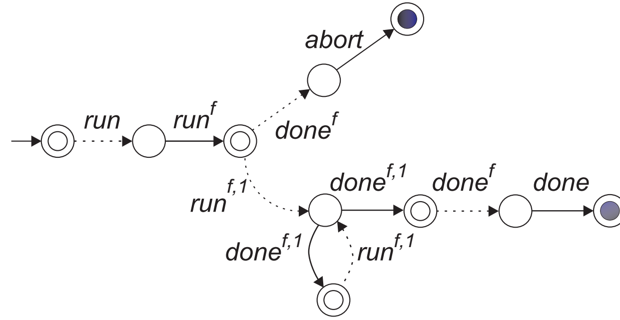


Figure 6.3: The strategy for the running example

We start with applying the coarsest abstraction [] to x, which means that x can only have the value $\mathbb{Z}$ (i.e. a nondeterministic choice over all integers).

Let the arbitrary subterm N be $f(x := x + 1)$. The model of the whole term is obtained by composing the model for the scope of variable declaration with the strategy $\mathsf{cell}_{x,0}$, which is used for remembering the initial (0) or the most recently written value into the variable x. This strategy ensures "good variable" behaviour of x.

In Figure 6.4 are shown the models $[\![f \vdash M[-]]\!](\alpha)$ and $[\![f, x \vdash f(x := x + 1)]\!]$ at the Abstraction Refinement iteration 0. The *nd* move [3] in the first strategy marks that nondeterminism has occurred due to abstraction. In this case, the guard of the 'if' command has been evaluated nondeterministically to *true* or *false*, since the value of x might be any integer.

[3] It is neither Opponent nor Player, but a special marker move.

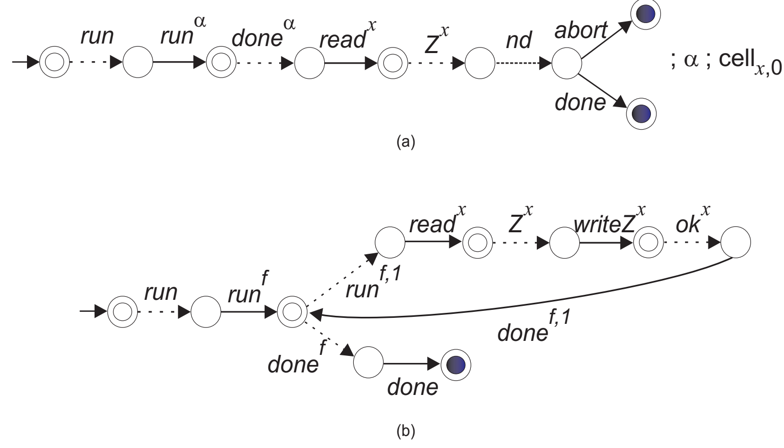


Figure 6.4: Strategies at AR iteration 0: (a) $[\![f \vdash M[-]]\!](\alpha)$ (b) $[\![f, x \vdash f(x := x{+}1)]\!]$

At each iteration, L^* updates its observation table and constructs a candidate assumption whenever the table becomes consistent and closed. The first such table produced and its associated assumption are given in Figure 6.5. Note that in observation tables we list only sequences from $S \cdot \mathsf{O}^{\mathsf{A}}\mathsf{P}^{\mathsf{A}}$ which are plays, and all other sequences are *false* by default. The equivalence query is then asked. The second AG premise fails and the Teacher returns a negative answer with a counterexample $c = \langle run \cdot run^f \cdot done^f \cdot done \rangle$, which is not safe when applied to $[\![f \vdash M[-]]\!]$. Thus, **AGCheck** reports $c' = \langle run \cdot run^f \cdot done^f \cdot nd \cdot abort \rangle$. Since this play is nondeterministic, our procedure decides to refine abstractions that caused the nondeterminism in c' and to continue. In this case, the abstraction of x is refined to $[0, 0]$, which contains three possible values: < 0, 0 and > 0.

At the Abstraction Refinement iteration 1, the strategies $[\![f \vdash M[-]]\!](\alpha)$ and $[\![f, x \vdash f(x := x + 1)]\!]$ are given in Figure 6.6.

Since we use a dynamic version of L^*, it starts with an observation table where S^0_1 and E^0_1 are the same as in the previous table T^0_0. The next

	T_1^1	E_1^1
		ε
S_1^1	ε	*true*
	$run \cdot done$	*false*
$S_1 \cdot \mathsf{O}^\mathsf{A}\mathsf{P}^\mathsf{A}$	$run \cdot done$	*false*
	$run \cdot read^x$	*true*
	$run \cdot writeZ^x$	*true*
	$run \cdot run^f$	*true*

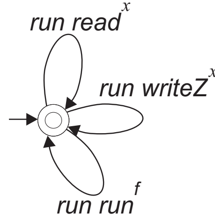


Figure 6.5: Observation table and assumption at AR iteration 0

candidate assumption is shown in Figure 6.7. The second AG rule premise fails giving $c = \langle run \cdot run^f \cdot done^f \cdot done \rangle$. Now, **AGCheck** reports a genuine counterexample $c' = \langle run \cdot run^f \cdot done^f \cdot abort \rangle$, and the procedure terminates informing that the input term is not safe.

6.3 Implementation

We implemented the compositional verification procedure presented above in the GameChecker tool (see Section 5.4). GameChecker compiles an abstracted open program into a process in the CSP process algebra, whose finite traces set represents the game-semantic model of the program. Membership and equivalence queries are answered using the FDR refinement checker [46]. If a counterexample is reported by the procedure, GameChecker is used to

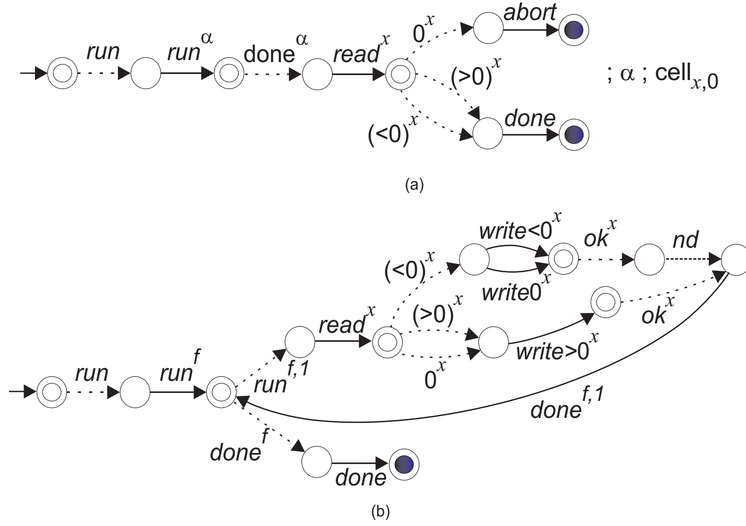


Figure 6.6: Strategies at AR iteration 1: (a) $[\![f \vdash M[-]]\!](\alpha)$ (b) $[\![f, x \vdash f(x := x+1)]\!]$

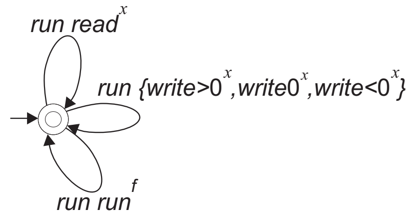


Figure 6.7: Assumption at AR iteration 1

analyse the counterexample and do abstraction refinement.

Consider the following implementation of a stack of maximum size k (see Section 5.4.4).

$\mathsf{empty} : \mathsf{com}, \mathsf{overflow} : \mathsf{com}, p : \mathsf{exp\,int},$

$\mathtt{ANALYSE}(\mathsf{com}, \mathsf{exp\,int}) : \mathsf{com} \vdash$

$\mathsf{new}_{\mathsf{int}}\ \mathtt{buffer}[k] := 0\ \mathsf{in}$

$\mathsf{new}_{\mathsf{int}}\ \mathtt{top} := 0\ \mathsf{in}$

$\mathsf{let\ com}\ \mathtt{push}(\mathsf{int}\ x)\ \{$

$\quad \mathsf{if}\ (!\mathtt{top} = k)\ \mathsf{then\ overflow\ else}\ \{\mathtt{buffer}[!\mathtt{top}] := !x;\ \ \mathtt{top} := !\mathtt{top} + 1\}\}\ \mathsf{in}$

$\mathsf{let\ exp\ int\ pop}\ \{$

$\quad \mathsf{if}\ (!\mathtt{top} = 0)\ \mathsf{then\ empty\ else}\ \{\mathtt{top} := !\mathtt{top} - 1;\ \ \mathsf{return}\ !\mathtt{buffer}[!\mathtt{top} + 1]\}\}\ \mathsf{in}$

$\mathtt{ANALYSE}(\mathtt{push}(p), \mathtt{pop})$

By replacing empty with abort command (resp. overflow with abort), we can check separately for 'empty', i.e. reads from empty stacks, (resp. 'overflow', i.e. writes to full stacks) errors, both of which are present for any n. For 'empty' (resp. 'overflow') error, a genuine counterexample is reported after refining the abstraction of `top` to $[0, 0]$ (resp. $[0, k]$). The counterexample corresponds to a single call of `pop` method (resp. $k + 1$ consecutive calls of `push` method) after which abort is executed.

	empty		overflow	
k	Direct	AG	Direct	AG
3	271	107	286	147
10	306	135	937	441
15	331	155	1462	651
25	381	195	2662	1071

Table 6.1: Experimental results for checking a stack implementation

We applied the AG procedure by learning an appropriate assumption for the `push` (resp. `pop`) method. In both cases, we obtain conclusive assump-

tions with null states since counterexamples are reported for all valid plays of the subterms we learn. Table 6.1 contains the experimental results for checking the two properties by using the AG procedure and the direct verification procedure without AG reasoning (see Section 5.4). We list the size of the largest generated transition system in each case for different values of k.

Chapter 7

Conclusions

This final chapter looks back and summarises the main achievements presented in this thesis, focussing on the overall picture. We close by briefly looking ahead at on-going and future work.

The aim of this thesis was to develop a fully compositional semantic framework for verifying safety properties of open sequential programs. As a main presentation vehicle, the metalanguage AIA (Abstracted Idealized Algol) was considered. AIA is an expressive programming language which combines the fundamental features of imperative and functional languages such as block-allocated variables, expressions with side effects, higher order functions, abnormal termination etc. The language incorporates abstraction annotations at the level of data types, which allows the writing of abstracted programs in a syntax similar to that of concrete programs. In Chapter 2 the language AIA was introduced. Chapter 3 presented a cartesian closed category of games and strategies on games which yields an interpretation of AIA. This model was shown to be fully abstract, and so it was used for proving safety of programs, i.e. to show that a program cannot execute the designated unsafe command

abort. Chapter 4 showed that for the second-order finitary AIA the model can be given certain kinds of concrete automata-theoretic representations, which are independent of the syntax. In particular, the model was represented using regular languages and the CSP process algebra. Once the model was constructed standard model-checking methods were applied to check safety of finite programs. In Chapter 5 an abstraction refinement procedure was proposed which enables programs with infinite integer types to be verified, and which terminates for unsafe programs. A prototype tool implementing the procedure was outlined, and some positive experimental results were reported. Chapter 6 combined abstraction refinement, assume-guarantee reasoning and the L^* algorithm for learning regular languages to yield a procedure for compositional verification. Game semantics was used to construct accurate models of programs compositionally, and an automatic assume-guarantee procedure with learning was used for achieving compositional verification.

Given the novelty of the game semantics approach to software verification, our practical work was concentrated on prototyping and evaluation of a variety of academic examples. Further work is therefore necessary to make the approach scale to industrial software. Some interesting extensions, which will bring the approach closer to realistic applications, are the following.

New features The language fragment we study includes some basic imperative and functional features. We consider extending it with several features for which finite representations of the game semantic model are known. Finite-data programs with shared-variable concurrency and semaphores [53, 55], as well as call-by-value procedures [49] can be analysed via regular languages. Also, it was established in [87] that finitary IA with third-order procedures can be modelled using visibly pushdown

automata [11]. The problem of adaptation of the abstraction refinement procedure to a language enriched with these features is not trivial and is critical for software verification of a real programming language, such as a substantial subset of C or Java. Other interesting features to consider are general references [6], control primitives [76], recursive types [82], polymorphism [9], probabilistic constructs [35], etc.

Predicate abstraction Extending abstractions to arbitrary predicates is critical for achieving efficient verification. Predicate abstraction allows only relations between variables to be captured. This can be done by modelling programs of second-order IA with infinite data types using symbolic automata [65], in which data is not represented explicitly but symbolically. The construction of symbolic automata will be based on the regular language game semantics [51]. In this way we expect to obtain a SLAM-like predicate-abstraction refinement procedure which will be applicable to realistic industrial-size programs.

Liveness properties The game model for AIA is derived with respect to "may termination" program equivalence, so that two programs are considered equivalent if they can produce the same range of output values. This model is acceptable for reasoning about safety properties, but it gives no account of liveness properties. To address this problem, a model which is fully abstract with respect to "may and must termination" program equivalence is needed. We expect to construct such a model by reworking the idea of divergences from CSP in the context of game semantics in the similar way as it was done in [63] to obtain a may and must termination model of EIA.

Appendix A

CSP Scripts for Case Studies

This appendix provides the CSP scripts produced by our compiler for the stack implementation whose analyses are presented in Section 4.3.5. We first present the CSP script containing functions which implement all arithmetic-logic operations.

A.1 Functions

functions.csp

```
minus(down,upper,v) =
    if v==Z then Z
    else if v==Plus then Minus
    else if v==Minus then Plus
    else if -v>upper then Plus
    else if -v<down then Minus
    else -v

abs(v) =
 if v>=0 then v
```

```
    else -v

mod(down,upper,down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {Z}
    else if v1==0 then {0}
    else if v1==Plus or v2==Plus
            then union({down..upper},{Minus,Plus})
    else if v1==Minus or v2==Minus
            then union({down..upper},{Minus,Plus})
    else if v1%v2>upper then {Plus}
    else if v1%v2<down then {Minus}
    else {v1%v2}

div(down,upper,down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {Z}
    else if v1==0 then {0}
    else if v1==Plus or v2==Plus
            then union({down..upper},{Minus,Plus})
    else if v1==Minus or v2==Minus
            then union({down..upper},{Minus,Plus})
    else if v1/v2>upper then {Plus}
    else if v1/v2<down then {Minus}
    else {v1/v2}

mul(down,upper,down1,upper1,down2,upper2,v1,v2) =
    if (v1==0 or v2==0) then {0}
    else if v1==Z or v2==Z then {Z}
    else if v1==Plus and v2>=upper2
            and upper1*upper2>down1*down2 then {Plus}
    else if v1==Plus and v2==Minus
            and upper1*down2<=down1*upper2 then {Minus}
    else if v1==Plus then union({down..upper},{Minus,Plus})
    else if v2==Plus and v1>=upper1 then {Plus}
    else if v2==Plus and v1==Minus
            and upper2*down1<=down2*upper1 then {Minus}
```

```
    else if v2==Plus then union({down..upper},{Minus,Plus})
    else if v1==Minus and v2<=down2
            and down1*down2>=upper1*upper2 then {Plus}
    else if v1==Minus then union({down..upper},{Minus,Plus})
    else if v2==Minus and v1<=down1
            and down1*down2>=upper1*upper2 then {Plus}
    else if v2==Minus then union({down..upper},{Minus,Plus})
    else if v1*v2>upper then {Plus}
    else if v1*v2<down then {Minus}
    else { v1*v2 }

sub(down,upper,down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {Z}
    else if v1==Plus and v2<=down2 then {Plus}
    else if v1==Plus and v2==Plus
            then union({down..upper},{Minus,Plus})
    else if v1==Plus and upper1+1-v2>=down
            then union({upper1+1-v2..upper},{Plus})
    else if v1==Plus then union({down..upper},{Minus,Plus})
    else if v2==Plus and v1<=down1 then {Minus}
    else if v2==Plus and v1-(upper2+1)<=upper
            then union({down..v1-(upper2+1)},{Minus})
    else if v1==Plus then union({down..upper},{Minus,Plus})
    else if v1==Minus and v2>=upper2 then {Minus}
    else if v1==Minus and v2==Minus
            then union({down..upper},{Minus,Plus})
    else if v1==Minus and down1-1-v2<=upper
            then union({down..down1-1-v2},{Minus})
    else if v1==Minus then union({down..upper},{Minus,Plus})
    else if v2==Minus and v1>=upper1 then {Plus}
    else if v2==Minus and v1-(down2-1)>=down
            then union({v1-(down2-1)..upper},{Plus})
    else if v2==Minus then union({down..upper},{Minus,Plus})
    else if v1-v2>upper then {Plus}
    else if v1-v2<down then {Minus}
```

```
    else {v1-v2}

add(down,upper,down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {Z}
    else if v1==Plus and v2>=upper2 then {Plus}
    else if v1==Plus and v2==Minus
            then union({down..upper},{Minus,Plus})
    else if v1==Plus and upper1+1+v2>=down
            then union({upper1+1+v2..upper},{Plus})
    else if v1==Plus then union({down..upper},{Minus,Plus})
    else if v2==Plus and v1>=upper1 then {Plus}
    else if v2==Plus and v1==Minus
            then union({down..upper},{Minus,Plus})
    else if v2==Plus and upper2+1+v1>=down
            then union({upper2+1+v1..upper},{Plus})
    else if v2==Plus then union({down..upper},{Minus,Plus})
    else if v1==Minus and v2<=down2 then {Minus}
    else if v1==Minus and down1-1+v2<=upper
            then union({down..down1-1+v2},{Minus})
    else if v1==Minus then union({down..upper},{Minus,Plus})
    else if v2==Minus and v1<=down1 then {Minus}
    else if v2==Minus and down2-1+v1<=upper
            then union({down..down2-1+v1},{Minus})
    else if v2==Minus then union({down..upper},{Minus,Plus})
    else if v1+v2>upper then {Plus}
    else if v1+v2<down then {Minus}
    else {v1+v2}

greatereq(down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {false,true}
    else if v1==Plus and v2<=(upper1+1) and v2!=Plus
            then {true}
    else if v2==Plus and v1<=upper2 then {false}
    else if v1==Minus and v2>=down1 then {false}
    else if v2==Minus and v1>=(down2-1) and v1!=Minus
```

```
            then {true}
    else if v1==Minus or v2==Minus then {false,true}
    else if v1==Plus or v2==Plus then {false,true}
    else {v1>=v2}

lesseq(down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {false,true}
    else if v1==Plus and v2<=upper1 then {false}
    else if v2==Plus and v1<=(upper2+1) and v1!=Plus
            then {true}
    else if v1==Minus and v2>=(down1-1) and v2!=Minus
            then {true}
    else if v2==Minus and v1>=down2 then {false}
    else if v1==Minus or v2==Minus then {false,true}
    else if v1==Plus or v2==Plus then {false,true}
    else {v1<=v2}

greater(down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {false,true}
    else if v1==Plus and v2<=upper1 then {true}
    else if v2==Plus and v1<=(upper2+1) and v1!=Plus
            then {false}
    else if v1==Minus and v2>=(down1-1) and v2!=Minus
            then {false}
    else if v2==Minus and v1>=down2 then {true}
    else if v1==Minus or v2==Minus then {false,true}
    else if v1==Plus or v2==Plus then {false,true}
    else {v1>v2}

less(down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {false,true}
    else if v1==Plus and v2<=(upper1+1) and v2!=Plus
            then {false}
    else if v2==Plus and v1<=upper2 then {true}
    else if v1==Minus and v2>=down1 then {true}
```

```
    else if v2==Minus and v1>=(down2-1) and v1!=Minus
            then {false}
    else if v1==Minus or v2==Minus then {false,true}
    else if v1==Plus or v2==Plus then {false,true}
    else {v1<v2}

noteqn(down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {false,true}
    else if v1==Plus and v2<=upper1 then {true}
    else if v2==Plus and v1<=upper2 then {true}
    else if v1==Minus and v2>=down1 then {true}
    else if v2==Minus and v1>=down2 then {true}
    else if v1==Minus or v2==Minus then {false,true}
    else if v1==Plus or v2==Plus then {false,true}
    else {v1!=v2}

eqn(down1,upper1,down2,upper2,v1,v2) =
    if v1==Z or v2==Z then {false,true}
    else if v1==Plus and v2<=upper1 then {false}
    else if v2==Plus and v1<=upper2 then {false}
    else if v1==Minus and v2>=down1 then {false}
    else if v2==Minus and v1>=down2 then {false}
    else if v1==Minus or v2==Minus then {false,true}
    else if v1==Plus or v2==Plus then {false,true}
    else {v1==v2}

cast(fromdown,fromupper,todown,toupper,v) =
    if todown==Z then {Z}
    else if v==Z then union({todown..toupper},{Minus,Plus})
    else if v==Plus and fromupper>=toupper then {Plus}
    else if v==Minus and fromdown<=todown then {Minus}
    else if v==Plus then union({ fromupper+1..toupper},{Plus})
    else if v==Minus then union({fromdown-1..todown},{Minus})
    else if v>toupper then {Plus}
    else if v<todown then {Minus}
```

```
    else {v}

Rank(down,upper,v) =
    if v==Z then 0
    else if v==Minus and down>=0 then 6*down+3
    else if v==Minus and down<0 then (-6)*(down-1)+4
    else if v==Plus and upper>=0 then 6*upper+1
    else if v==Plus and upper<0 then (-6)*(upper+1)+2
    else if v>=0 then 6*v+5
    else (-6)*(v+1)+4
```

A.2 Stack Implementation

The complete CSP script for the stack implementation in Figure 4.8 with $k = 3$ is presented next.

```
stack.csp


include "functions.csp"
transparent diamond, sbisim, normal

UB(in,v) = (in.A.B!v -> UB(in,v)) [] (in.Q.writeB?v1 ->
    UB(in,v1)) [] SKIP
RTUB(in,v) = (in.A.B!v -> RTUB(in,v)) [] (in.Q.writeB?v1 ->
    RTUB(in,v1)) [] SKIP

UN(in,v,S) = (in.A.N!v -> UN(in,v,S)) [] (in.Q.writeN?v1:S ->
    UN(in,v1,S)) [] SKIP
RTUN(in,v,S) = (in.A.N!v -> RTUN(in,v,S)) []
    (in.Q.writeN?v1:S -> RTUN(in,v1,S)) [] SKIP
```

```
Mem = (Update?v -> Mem1(v)) [] ([]w:{|overflow|}@w -> Mem) []
    ([]w:{|OUB|}@w -> Mem) [] SKIP
Mem1(v) = (Update?v1 -> Mem1(v)) [] ([]w1:{|overflow|}@w1 ->
    What!v -> Mem1(v)) [] ([]w1:{|OUB|}@w1 -> What!v -> Mem1(v))
    [] SKIP

Minus = -1
Plus = 4
Z = 5

NumbersZ = {Z}
Numbers0t3 = union({Plus,Minus},{ 0 .. 3 })

datatype question = q | run | read | writeB.Bool |
    writeN.{Minus ..Z}
datatype answer = done | ok | B.Bool | N.{Minus .. Z}
datatype var = Q.question | A.answer

datatype nondet = GREAT.{ Minus.. Z } | LESS.{ Minus .. Z } |
    NUMBER.{ Minus .. Z } | BOOL.Bool

channel Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7 : question
channel A0, A1, A2, A3, A4, A5, A6, A7 : answer

channel OUB, top : var
channel EXH
channel ND : nondet

channel Update, What : { 0..6*Z}
channel empty : {0}.var

channel overflow : {0}.var
channel p : var

channel ANALYSE:{0,1,2}.var
```

```
channel buffer : {0,1,2}.var

channel top: var

P3 = Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 -> A0.N.v ->
    SKIP
M3 = SKIP [] ( Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 ->
    A0.N.v -> M3 )

P4 = Q0.q -> A0.N.3 -> SKIP
M4 = SKIP [] ( Q0.q -> A0.N.3 -> M4 )

P5 = P3 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] (P4 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] (Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q ->
    A2.N?v2:{ 3 } -> (let a1=eqn(0,3,3,3,v1,v2) within if
    (card(a1)==1) then (A0.B?va1:a1 -> SKIP) else (A0.B.va1 ->
    ND.BOOL?va1:a1 -> SKIP))) \ {|Q2,A2|}) \ {|Q1,A1|}
C5 = SKIP [] ( Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q ->
    A2.N?v2:{ 3 } -> (let a1=eqn(0,3,3,3,v1,v2) within if
    (card(a1)==1) then (A0.B?va1:a1 -> C5) else ( A0.B.va1
     -> ND.BOOL?va1:a1 -> C5 ) ) )
M5 = M3 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M4 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] C5 \ {|Q2,A2|} ) \ {|Q1,A1|}

P6 =  SKIP

P7 =  ( ( Q0.run -> overflow.0.Q.run -> P6 ) ;
    ( overflow.0.A.done -> A0.done -> SKIP ) )
C7 = SKIP [] ( ( Q0.run -> overflow.0.Q.run -> P6 ) ;
    ( overflow.0.A.done -> A0.done -> C7 ) )
M7 = C7

P10 = Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 ->
    A0.N.v -> SKIP
M10 = SKIP [] ( Q0.q -> top.Q.read ->
```

```
    top.A.N?v:Numbers0t3 -> A0.N.v -> M10 )

P11 = Q0.q -> A0.N.1 -> SKIP
M11 = SKIP [] ( Q0.q -> A0.N.1 -> M11 )

Numbers1t3 = union({Plus,Minus},{ 1 .. 3 })

P12 = P10 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] (P11 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] (Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q ->
    A2.N?v2:{ 1 } -> (let a1=add(1,3,0,3,1,1,v1,v2) within if
    (card(a1)==1) then ( A0.N?va1:a1 -> SKIP) else ( []va1:a1
    @ if (va1==Plus) then (A0.N.va1 -> ND.GREAT.3 -> SKIP) else
    (if (va1==Minus) then (A0.N.va1 -> ND.LESS.1 -> SKIP) else
    ( A0.N.va1 -> ND.NUMBER.va1 -> SKIP ) ) ) ) )
    \ {|Q2,A2|}) \ {|Q1,A1|}
C12 = SKIP [] ( Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q
    -> A2.N?v2:{ 1 } -> (let a1=add(1,3,0,3,1,1,v1,v2) within if
    (card(a1)==1) then (A0.N?va1:a1 -> C12) else ( []va1:a1 @ if
    (va1==Plus) then (A0.N.va1 -> ND.GREAT.3 -> C12) else (if
    (va1==Minus) then (A0.N.va1 -> ND.LESS.1 -> C12) else
    ( A0.N.va1 -> ND.NUMBER.va1 -> C12 ) ) ) ) )
M12 = M10 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M11 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] C12 \ {|Q2,A2|} ) \ {|Q1,A1|}

P9 = P12 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( Q0.run -> Q1.q ->
    A1.N?v:Numbers1t3 -> ( let a2=cast(1,3,0,3,v) within if
    (card(a2)==1) then ( top.Q.writeN?va2:a2 -> top.A.ok ->
    A0.done -> SKIP) else ( []va2:a2 @ if (va2==Plus) then
    (top.Q.writeN.va2 -> top.A.ok -> ND.GREAT.3 -> A0.done ->
    SKIP) else (if (va2==Minus) then (top.Q.writeN.va2
    -> top.A.ok -> A0.done -> ND.LESS.0 -> SKIP ) else
    (top.Q.writeN.va2 -> top.A.ok -> A0.done -> ND.NUMBER.va2
    -> SKIP ) ) ) ) ) \ {|Q1,A1|}
C9 = SKIP [] ( Q0.run -> Q1.q -> A1.N?v:Numbers1t3 -> ( let
    a2=cast(1,3,0,3,v) within if (card(a2)==1) then (
```

```
    top.Q.writeN?va2:a2 -> top.A.ok -> A0.done -> C9) else
    ([]va2:a2 @ if (va2==Plus) then (top.Q.writeN.va2
    -> top.A.ok -> A0.done -> ND.GREAT.3 -> C9 ) else (if
    (va2==Minus) then (top.Q.writeN.va2 -> top.A.ok
    -> A0.done -> ND.LESS.0 -> C9 ) else (top.Q.writeN.va2
    -> top.A.ok -> A0.done -> ND.NUMBER.va2 -> C9) ) ) ) )
M9 = M12 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] C9 \ {|Q1,A1|}

P14 = Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 -> A0.N.v ->
    SKIP
M14 = SKIP [] ( Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 ->
    A0.N.v -> M14 )

P13 = P14 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] (P1[[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] (Q0.run -> Q2.q -> A2.N?v2:{ Z } -> Q1.q ->
    A1.N?v1:Numbers0t3 -> if (member(v1,{0..2})) then ( ( let
    a2=cast(Z,Z,Z,Z,v2) within if (card(a2)==1) then (
    buffer.v1.Q.writeN?va2:a2 -> buffer.v1.A.ok -> A0.done ->
    SKIP) else ([]va2:a2 @ if (va2==Plus) then
    (buffer.v1.Q.writeN.va2 -> buffer.v1.A.ok -> A0.done ->
    ND.GREAT.Z -> SKIP ) else (if (va2==Minus) then
    (buffer.v1.Q.writeN.va2 -> buffer.v1.A.ok -> A0.done ->
    ND.LESS.Z -> SKIP) else (buffer.v1.Q.writeN.va2
    -> buffer.v1.A.ok -> A0.done -> ND.NUMBER.va2 ->
    SKIP ) ) ) ) ) else (if (card(greatereq(0,3,0,0,v1,0))!=1
    or card(lesseq(0,3,2,2,v1,2))!=1) then (OUB.Q.run ->
    OUB.A.done -> A0.done -> ND.BOOL.false -> SKIP) else
    (OUB.Q.run -> OUB.A.done -> A0.done -> SKIP)) )
    \ {|Q2,A2|} ) \ {|Q1,A1|}
C13 = SKIP [] ( Q0.run -> Q2.q -> A2.N?v2:{ Z } -> Q1.q ->
    A1.N?v1:Numbers0t3 -> if (member(v1,{0..2})) then ( ( let
    a2=cast(Z,Z,Z,Z,v2) within if (card(a2)==1) then (
    buffer.v1.Q.writeN?va2:a2 -> buffer.v1.A.ok -> A0.done ->
    C13) else ([]va2:a2 @ if (va2==Plus) then
    (buffer.v1.Q.writeN.va2 -> buffer.v1.A.ok -> A0.done ->
```

```
    ND.GREAT.Z -> C13 ) else (if (va2==Minus) then
    (buffer.v1.Q.writeN.va2 -> buffer.v1.A.ok -> A0.done ->
    ND.LESS.Z -> C13 ) else (buffer.v1.Q.writeN.va2
    -> buffer.v1.A.ok -> A0.done -> ND.NUMBER.va2 -> C13 )
    ) ) ) ) else (if (card(greatereq(0,3,0,0,v1,0))!=1 or
    card(lesseq(0,3,2,2,v1,2))!=1) then (OUB.Q.run ->
    OUB.A.done -> A0.done -> ND.BOOL.false -> C13) else
    (OUB.Q.run -> OUB.A.done -> A0.done -> C13)) )
M13 = M14 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M1 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] C13 \ {|Q2,A2|} ) \ {|Q1,A1|}

P8 = ( P13 [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|]( P9 [[Q0<-Q1,A0<-A1]]
    [|{|Q1,A1|}|] (Q0.run -> Q2.run -> A2.done -> Q1.run ->
    A1.done -> A0.done -> SKIP) \ {|Q1,A1|} )  \ {|Q2,A2|} )
C8 = SKIP []  (Q0.run -> Q2.run -> A2.done -> Q1.run -> A1.done
    -> A0.done -> C8)
M8 = ( M13 [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|]( M9 [[Q0<-Q1,A0<-A1]]
    [|{|Q1,A1|}|]C8 \ {|Q1,A1|} )  \ {|Q2,A2|} )

P15 = M5 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M7 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] (M8  [[Q0<-Q3,A0<-A3]] [|{|Q3,A3|}|]  (Q0.run
    -> Q1.q -> A1.B?v -> if v then (Q2.run -> A2.done -> A0.done
    -> SKIP) else (Q3.run -> A3.done -> A0.done -> SKIP ) )
    \ {|Q3,A3|} ) \ {|Q2,A2|} ) \ {|Q1,A1|}
C15 = SKIP [] ( Q0.run -> Q1.q -> A1.B?v -> if v then (Q2.run ->
    A2.done -> A0.done -> C15) else (Q3.run -> A3.done -> A0.done
    -> C15 ) )
M15 = M5 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M7 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] ( M8  [[Q0<-Q3,A0<-A3]] [|{|Q3,A3|}|] C15
    \ {|Q3,A3|} ) \ {|Q2,A2|} ) \ {|Q1,A1|}

P17 = Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 -> A0.N.v ->
    SKIP
M17 = SKIP [] ( Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 ->
    A0.N.v -> M17 )
```

```
P18 = Q0.q -> A0.N.0 -> SKIP
M18 = SKIP [] ( Q0.q -> A0.N.0 -> M18 )

P19 = P17 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] (P18 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] (Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q ->
    A2.N?v2:{ 0 } -> (let a1=eqn(0,3,0,0,v1,v2) within if
    (card(a1)==1) then (A0.B?va1:a1 -> SKIP) else (A0.B.va1
    -> ND.BOOL?va1:a1 -> SKIP) ) ) \ {|Q2,A2|}) \ {|Q1,A1|}
C19 = SKIP [] ( Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q ->
    A2.N?v2:{ 0 } -> (let a1=eqn(0,3,0,0,v1,v2) within if
    (card(a1)==1) then (A0.B?va1:a1 -> C19) else
    ( A0.B.va1 -> ND.BOOL?va1:a1 -> C19 ) ) )
M19 = M17 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M18 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] C19 \ {|Q2,A2|} ) \ {|Q1,A1|}

P21 = Q0.q -> A0.N.0 -> SKIP
M21 = SKIP [] ( Q0.q -> A0.N.0 -> M21 )

P22 = SKIP

P23 =  ( ( Q0.run -> empty.0.Q.run -> P22 ) ; ( empty.0.A.done ->
    A0.done -> SKIP ) )
C23 = SKIP [] ( ( Q0.run -> empty.0.Q.run -> P22 ) ; (
    empty.0.A.done -> A0.done -> C23 ) )
M23 = C23

P20 = ( P23 [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|](P21
    [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] (Q0.q -> Q2.run -> A2.done ->
    Q1.q -> A1.N?v:{ 0 } ->  (let a1=cast(0,0,0,0,v) within if
    (card(a1)==1) then (A0.N?va1:a1 -> SKIP ) else ([]va1:a1 @
    if (va1==Plus) then (A0.N.va1 -> ND.GREAT.0 -> SKIP) else
    (if (va1==Minus) then (A0.N.va1 -> ND.LESS.0 -> SKIP) else
    (A0.N.va1 -> ND.NUMBER.va1 -> SKIP) ) ) ) ) \ {|Q1,A1|} )
    \ {|Q2,A2|} )
```

```
C20 = SKIP []  (Q0.q -> Q2.run -> A2.done -> Q1.q -> A1.N?v:{ 0 }
    -> (let a1=cast(0,0,0,0,v) within if (card(a1)==1) then
    (A0.N?va1:a1 -> C20) else ([]va1:a1 @ if (va1==Plus) then
    (A0.N.va1 -> ND.GREAT.0 -> C20 ) else (if (va1==Minus) then
    (A0.N.va1 -> ND.LESS.0 -> C20 ) else
    (A0.N.va1 -> ND.NUMBER.va1 -> C20 ) ) ) ) )
M20 = ( M23 [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|](M21 [[Q0<-Q1,A0<-A1]]
    [|{|Q1,A1|}|]C20 \ {|Q1,A1|} )  \ {|Q2,A2|} )

P26 = Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 -> A0.N.v ->
    SKIP
M26 = SKIP [] ( Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 ->
    A0.N.v -> M26 )

P25 = P26 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] (Q0.q -> Q1.q ->
    A1.N?v1:Numbers0t3 -> if (member(v1,{0..2})) then
    (buffer.v1.Q.read -> buffer.v1.A.N?v2:NumbersZ -> A0.N.v2 ->
    SKIP) else (if (card(greatereq(0,3,0,0,v1,0))!=1 or
    card(lesseq(0,3,2,2,v1,2))!=1) then (OUB.Q.run ->
    OUB.A.done -> A0.N.Z -> ND.BOOL.false -> SKIP) else
    (OUB.Q.run -> OUB.A.done -> A0.N.Z -> SKIP)) )  \ {|Q1,A1|}
C25 = SKIP [] ( Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> if
    (member(v1,{0..2})) then (buffer.v1.Q.read ->
    buffer.v1.A.N?v2:NumbersZ -> A0.N.v2 -> C25) else (if
    (card(greatereq(0,3,0,0,v1,0))!=1 or
    card(lesseq(0,3,2,2,v1,2))!=1) then (OUB.Q.run ->
    OUB.A.done -> A0.N.Z -> ND.BOOL.false -> C25) else
    (OUB.Q.run -> OUB.A.done -> A0.N.Z -> C25)) )
M25 = M26 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] C25 \ {|Q1,A1|}

P28 = Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 -> A0.N.v ->
    SKIP
M28 = SKIP [] ( Q0.q -> top.Q.read -> top.A.N?v:Numbers0t3 ->
    A0.N.v -> M28 )
```

```
P29 = Q0.q -> A0.N.1 -> SKIP
M29 = SKIP [] ( Q0.q -> A0.N.1 -> M29 )
Numbers0t2 = union({Plus,Minus},{ 0 .. 2 })

P30 = P28 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] (P29 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] (Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q ->
    A2.N?v2:{ 1 } -> (let a1=sub(0,2,0,3,1,1,v1,v2) within if
    (card(a1)==1) then (A0.N?va1:a1 -> SKIP) else ( []va1:a1 @
    if (va1==Plus) then (A0.N.va1 -> ND.GREAT.2 -> SKIP) else
    (if (va1==Minus) then (A0.N.va1 -> ND.LESS.0 -> SKIP)
    else (A0.N.va1 -> ND.NUMBER.va1 -> SKIP ) ) ) ) )
    \ {|Q2,A2|}) \ {|Q1,A1|}
C30 = SKIP [] ( Q0.q -> Q1.q -> A1.N?v1:Numbers0t3 -> Q2.q ->
    A2.N?v2:{ 1 } -> (let a1=sub(0,2,0,3,1,1,v1,v2) within if
    (card(a1)==1) then (A0.N?va1:a1 -> C30) else ( []va1:a1 @ if
    (va1==Plus) then (A0.N.va1 -> ND.GREAT.2 -> C30) else (if
    (va1==Minus) then (A0.N.va1 -> ND.LESS.0 -> C30) else
    (A0.N.va1 -> ND.NUMBER.va1 -> C30 ) ) ) ) )
M30 = M28 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M29 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] C30 \ {|Q2,A2|} ) \ {|Q1,A1|}

P27 = P30 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( Q0.run -> Q1.q ->
    A1.N?v:Numbers0t2 -> ( let a2=cast(0,2,0,3,v) within if
    (card(a2)==1) then ( top.Q.writeN?va2:a2 -> top.A.ok ->
    A0.done -> SKIP) else ( []va2:a2 @ if (va2==Plus) then
    (top.Q.writeN.va2 -> top.A.ok -> A0.done -> ND.GREAT.3 ->
    SKIP ) else (if (va2==Minus) then (top.Q.writeN.va2
    -> top.A.ok -> A0.done -> ND.LESS.0 -> SKIP ) else
    (top.Q.writeN.va2 -> top.A.ok -> A0.done ->
    ND.NUMBER.va2 -> SKIP ) ) ) ) ) \ {|Q1,A1|}
C27 = SKIP [] ( Q0.run -> Q1.q -> A1.N?v:Numbers0t2 -> ( let
    a2=cast(0,2,0,3,v) within if (card(a2)==1) then (
    top.Q.writeN?va2:a2 -> top.A.ok -> A0.done -> C27) else
    ([]va2:a2 @ if (va2==Plus) then (top.Q.writeN.va2
    -> top.A.ok -> A0.done -> ND.GREAT.3 -> C27 ) else (if
```

```
    (va2==Minus) then (top.Q.writeN.va2 -> top.A.ok
    -> A0.done -> ND.LESS.0 -> C27 ) else (top.Q.writeN.va2
    -> top.A.ok -> A0.done -> ND.NUMBER.va2 -> C27 ) ) ) ) )
M27 = M30 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] C27 \ {|Q1,A1|}

P24 = ( P27 [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|](P25 [[Q0<-Q1,A0<-A1]]
    [|{|Q1,A1|}|] (Q0.q -> Q2.run -> A2.done -> Q1.q -> A1.N?v:
    {Z} -> (let a1=cast(Z,Z,0,0,v) within if (card(a1)==1) then
    (A0.N?va1:a1 -> SKIP) else ( []va1:a1 @ if (va1==Plus) then
    (A0.N.va1 -> ND.GREAT.0 -> SKIP ) else (if (va1==Minus)
    then (A0.N.va1 -> ND.LESS.0 -> SKIP ) else (A0.N.va1 ->
    ND.NUMBER.va1 -> SKIP ) ) ) ) ) \ {|Q1,A1|} ) \ {|Q2,A2|} )
C24 = SKIP [] (Q0.q -> Q2.run -> A2.done -> Q1.q -> A1.N?v:{ Z }
    -> (let a1=cast(Z,Z,0,0,v) within if (card(a1)==1) then
    (A0.N?va1:a1 -> C24) else ( []va1:a1 @ if (va1==Plus) then
    (A0.N.va1 -> ND.GREAT.0 -> C24 ) else (if (va1==Minus) then
    (A0.N.va1 -> ND.LESS.0 -> C24 ) else
    (A0.N.va1 -> ND.NUMBER.va1 -> C24 ) ) ) ) )
M24 = ( M27 [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|](M25 [[Q0<-Q1,A0<-A1]]
    [|{|Q1,A1|}|]C24 \ {|Q1,A1|} )  \ {|Q2,A2|} )

P31 = M19 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M20 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] ( M24  [[Q0<-Q3,A0<-A3]] [|{|Q3,A3|}|] (Q0.q ->
    Q1.q -> A1.B?v -> if v then (Q2.q -> A2.N?v1:Numbers0t0 ->
    (let a1=cast(0,0,0,0,v1) within if (card(a1)==1) then
    (A0.N?va1:a1 -> SKIP ) else ( []va1:a1 @ if (va1==Plus) then
    (A0.N.va1 -> ND.GREAT.0 -> SKIP) else (if (va1==Minus) then
    (A0.N.va1 -> ND.LESS.0 -> SKIP ) else (A0.N.va1 ->
    ND.NUMBER.va1 -> SKIP) ) ) ) ) else (Q3.q -> A3.N?v2:Numbers0t0
    -> (let a2=cast(0,0,0,0,v2) within if (card(a2)==1) then
    ( A0.N?va2:a2 -> SKIP ) else ( []va2:a2 @ if (va2==Plus)
    then (A0.N.va2 -> ND.GREAT.0 -> SKIP ) else (if
    (va2==Minus) then (A0.N.va2 -> ND.LESS.0 -> SKIP ) else
    (A0.N.va2 -> ND.NUMBER.va2 -> SKIP ) ) ) ) ) )
    \ {|Q3,A3|} ) \ {|Q2,A2|} ) \ {|Q1,A1|}
```

```
C31 = SKIP [] ( Q0.q -> Q1.q -> A1.B?v -> if v then (Q2.q ->
    A2.N?v1:Numbers0t0 -> (let a1=cast(0,0,0,0,v1) within if
    (card(a1)==1) then (A0.N?va1:a1 -> C31) else ( []va1:a1 @
    if (va1==Plus) then (A0.N.va1 -> ND.GREAT.0 -> C31 ) else
    (if (va1==Minus) then (A0.N.va1 -> ND.LESS.0 -> C31 ) else
    (A0.N.va1 -> ND.NUMBER.va1 -> C31) ) ) ) ) else (Q3.q ->
    A3.N?v2:Numbers0t0 -> (let a2=cast(0,0,0,0,v2) within if
    (card(a2)==1) then (A0.N?va2:a2 -> C31) else ( []va2:a2 @
    if (va2==Plus) then (A0.N.va2 -> ND.GREAT.0 -> C31 ) else
    (if (va2==Minus) then (A0.N.va2 -> ND.LESS.0 -> C31 ) else
    (A0.N.va2 -> ND.NUMBER.va2 -> C31 ) ) ) ) ) )
M31 = M19 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M20 [[Q0<-Q2,A0<-A2]]
    [|{|Q2,A2|}|] ( M24  [[Q0<-Q3,A0<-A3]] [|{|Q3,A3|}|] C31
    \ {|Q3,A3|} ) \ {|Q2,A2|} ) \ {|Q1,A1|}
Numbers0t0 = union({Plus,Minus},{ 0 .. 0 })

P33 = Q0.q -> p.Q.read -> p.A.N?v:NumbersZ -> A0.N.v -> SKIP
M33 = SKIP [] ( Q0.q -> p.Q.read -> p.A.N?v:NumbersZ -> A0.N.v
    -> M33 )

P1 = P33
M1 = M33

P34 = ( ANALYSE.1.Q.run -> Q1.run -> A1.done -> ANALYSE.1.A.done
    -> P34 ) []  ( ANALYSE.2.Q.q -> Q2.q -> A2.N?v: Numbers0t0 ->
    (let a1=cast(0,0,Z,Z,v) within if (card(a1)==1) then
    (ANALYSE.2.A.N?va1:a1 -> P34 ) else ( []va1:a1 @ if
    (va1==Plus) then (ANALYSE.2.A.N.va1 -> ND.GREAT.0 -> P34 )
    else (if (va1==Minus) then (ANALYSE.2.A.N.va1
    -> ND.LESS.0 -> P34 ) else (ANALYSE.2.A.N.va1
    -> ND.NUMBER.va1 -> P34 ) ) ) ) ) []  SKIP

P35 = ( M15 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M31
    [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|] ( ( Q0.run -> ANALYSE.0.Q.run
    -> P34 ) ; ( ANALYSE.0.A.done -> A0.done -> SKIP ) )
```

```
    \ {|Q2,A2|} ) \ {|Q1,A1|} )
C35 = SKIP [] ( ( Q0.run -> ANALYSE.0.Q.run -> P34 ) ;
    ( ANALYSE.0.A.done -> A0.done -> C35 ) )
M35 = ( M15 [[Q0<-Q1,A0<-A1]] [|{|Q1,A1|}|] ( M31
    [[Q0<-Q2,A0<-A2]] [|{|Q2,A2|}|] C35 \ {|Q2,A2|} )
    \ {|Q1,A1|} )

P32 = sbisim(diamond((P35 [|{|top.A.N,top.Q.writeN|}|]
    UN(top,0,Numbers0t3)) \ {|top|}))

Alpha0buffer(j) = if j==3 then
    Events else {|buffer.j.A.N,buffer.j.Q.writeN|}
Proces0buffer(j) = if j==3 then sbisim(diamond(P32)) else
    (UN(buffer.j,Z,{ Z }))

P36 = sbisim(diamond(( || j:{0..3} @ [Alpha0buffer(j)]
    Proces0buffer(j) ) \ {|buffer|}))

RUN=[]w:Events@w->RUN
Prop=([]w:diff(Events,{|overflow,OUB|})@w->Prop) [] SKIP
FreeNDProcess = ([]w:diff(Events,{|EXH,ND|})@w->FreeNDProcess)
    [] SKIP
FreeNDSearch = P36 [|Events|] FreeNDProcess

assert Prop [] RUN [T= FreeNDSearch
assert Prop [] RUN [T= P36
```

Bibliography

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full Abstraction for PCF (Extended Abstract). In Proceedings of *the International Symposium on Theoretical Aspects of Computer Software (TACS)*, LNCS **789**, (1994), 1–15. Springer.

[2] S. Abramsky, S. Gay, and R. Nagarajan. Interaction Categories and the Foundations of Typed Concurrent Programming. In Proceedings of *the 1994 Marktoberdorf Summer School: Deductive Program Design*, (1996), 35–113. Springer.

[3] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P.W.O'Hearn and R.D.Tennent, editors, *Algol-like languages.* (Birkhaüser, 1997).

[4] S. Abramsky and G. McCusker. Call-by-value games. In Proceedings of *the International Workshop on Computer Science Logic (CSL)*, LNCS **1414**, (1998), 1–17. Springer.

[5] S. Abramsky and G. McCusker. Game Semantics. In Proceedings of *the 1997 Marktoberdorf Summer School: Computational Logic* , (1998), 1–56.

Springer.

[6] S. Abramsky, K. Honda and G. McCusker. Fully Abstract Game Semantics for General References. In Proceedings of *the IEEE Symposium on Logic in Computer Science (LICS)*, (1998), 334-344. IEEE Computer Society Press.

[7] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with pasive expressions. *Theoretical Computer Science* **227**, (1999), 3–42.

[8] S. Abramsky. Algorithmic Game Semantics: A Tutorial Introduction. In Proceedings of *the 2001 Marktoberdorf Summer School: Proof and System Reliability*, (2001), 21–47. Springer.

[9] S. Abramsky and R. Jagadeesan. A Game Semantics for Generic Polymorphism. In Proceedings of *the International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, LNCS **2620**, (2003), 1–22. Springer.

[10] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying Game Semantics to Compositional Software Modeling and Verification. In Proceedings of *the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS **2988**, (2004), 421–435. Springer.

[11] R. Alur, and P. Madhusudan. Visibly pushdown languages. In Proceedings of *the ACM Symposium on Theory of Computing (STOC)*, (2004), 202–211. ACM Press.

[12] R. Alur, P. Cerny, G. Gupta, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In Proceedings of *the ACM*

Symposium on Principles of Programming Languages (POPL), (2005), 98–109. ACM Press.

[13] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In Proceedings of *the International Conference on Computer Aided Verification (CAV)*, LNCS **3576**, (2005), 548–562. Springer.

[14] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* **75**, (1987), 87–106.

[15] A.W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*, 2nd edition. (Cambridge University Press, 2002).

[16] A. Aspreti and G. Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist.* (MIT Press, 1991).

[17] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In Proceedings of *the International SPIN Workshop on Model Checking of Software (SPIN)*, LNCS **2057**, (2001), 103–122. Springer.

[18] G. Berry. Modèles complètement adéquats et stables des lambda-calculus typés. Thèse d'Etat, Université Paris VII, 1979.

[19] G. Berry and P. L.Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science* **20**, (1982), 265–321.

[20] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In Proceedings of *the International Conference on*

Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS **1579**, (1999), 193–207. Springer.

[21] A. Blass. A Game Semantics for Linear Logic. *Annals of Pure and Applied Logic* **56**, (1992), 183–220.

[22] S. Brookes. Full Abstraction for a Shared-Variable Parallel Language. *Information and Computation* **127(2)**, (1996), 145–163.

[23] A. Bucciarelli and T. Ehrhard. Extensional embedding of a strongly stable model of PCF. In Proceedings of *the International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS **510**, (1991), 35–46. Springer.

[24] S.Chaki, E. Clarke, A.Groce, S.Jha and H.Veith. Modular Verification of Software Components in C. In Proceedings of *the International Conference on Software Engineering (ICSE)*, (2003), 385–395. IEEE Computer Society.

[25] S. Chaki, E. Clarke, N. Sharygina, and N. Sinha. Dynamic Component Substiutability Analysis. In Proceedings of *the International Symposium of Formal Methods (FM)*, LNCS **3582**, (2005), 512–528. Springer.

[26] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzinski, F. Mang. Interface Compatibility Checking for Software Modules. In Proceedings of *the International Conference on Computer Aided Verification (CAV)*, LNCS **2404**, (2002), 428–441. Springer.

[27] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella. NuSMV 2: An OpenSource

Tool for Symbolic Model Checking. In Proceedings of *the International Conference on Computer Aided Verification (CAV)*, LNCS **2404**, (2002), 359–364. Springer.

[28] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Proceedings of *Logic of Programs Workshop (LOP)*, LNCS **131**, (1981), 52–71. Springer.

[29] E. M. Clarke, D. E. Long and K. L. McMillan. Compositional Model Checking. In Proceedings of *the IEEE Symposium on Logic in Computer Science (LICS)*, (1989), 353–362. IEEE Computer Society Press.

[30] E. M. Clarke, O. Grumberg and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **16(5)**, (1994), 1512–1542.

[31] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Proceedings of *the International Conference on Computer Aided Verification (CAV)*, LNCS **1855**, (2000), 154–169. Springer.

[32] E. M. Clarke, O. Grumberg and D. Peled. *Model Checking*. (MIT Press, 2000).

[33] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning Assumptions for Compositional Verification. In Proceedings of *the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS **2619**, (2003), 331–346. Springer.

[34] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code.

In Proceedings of *the International Conference on Software Engineering (ICSE)*, (2000), 439–448. ACM Press.

[35] V. Danos and R. Harmer. Probabilistic Game Semantics. In Proceedings of *the IEEE Symposium on Logic in Computer Science (LICS)*, (2000), 204–213. IEEE Computer Society Press.

[36] L. de Alfaro and T. Henzinger. Interface Theories for Component-Based Design. In Proceedings of *the First International Workshop on Embedded Software (EMSOFT)*, LNCS **2211**, (2001), 148–165. Springer.

[37] L. de Alfaro and T. Henzinger. Interface Automata. In Proceedings of *the International Symposium on Foundations of Software Engineering (FSE)*, (2001), 109–120. ACM Press.

[38] A. Dimovski and R. Lazić. CSP Representation of Game Semantics for Second-Order Idealized Algol. In Proceedings of *the International Conference on Formal Engineering Methods (ICFEM)*, LNCS **3308**, (2004), 146–161. Springer.

[39] A. Dimovski and R. Lazić. Software Model Checking Based on Game Semantics and CSP. In Proceedings of *the International Workshop on Automated Verification of Critical Systems (AVoCS)*, ENTCS **128:(6)**, (2005), 105–125. Elsevier.

[40] A. Dimovski, D. R. Ghica, and R. Lazić. Data-Abstraction Refinement: A Game Semantic Approach. In Proceedings of *the International Static Analysis Symposium (SAS)*, LNCS **3672**, (2005), 102–117. Springer.

[41] A. Dimovski, D. R. Ghica, and R. Lazić. A Counterexample-Guided Refinement Tool for Open Procedural Programs. In Proceedings of *the*

International SPIN Workshop on Model Checking of Software (SPIN), LNCS **3925**, (2006), 288–292. Springer.

[42] A. Dimovski and R. Lazić. Compositional Software Verification Based on Game Semantics and Process Algebras. *International Journal on Software Tools for Technology Transfer (STTT)* **9(1)**, (2007), 37–51.

[43] A. Dimovski and R. Lazić. Assume-Guarantee Software Verification Based on Game Semantics. In Proceedings of *the International Conference on Formal Engineering Methods (ICFEM)*, LNCS **4260**, (2006), 529–548. Springer.

[44] A. Ehrenfeucht. An application of games to the completeness problem for formalized theories. Fundamenta Mathematicae 49, (1961), 129–141.

[45] R. Floyd. Assigning meanings to programs. In Proceedings of *the Symposium on Applied Mathematics*, American Mathematical Society **19**, (1967), 19–32.

[46] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 Manual.* (http://www.fsel.com, 2000).

[47] R. Fraisse. Sur les classifications des systeḿes de relations. Universite d'Alger, Publications Scientifiques, Serie A, vol. 1 (1954), 35-182.

[48] D. R. Ghica and G. McCusker. Reasoning About Idealized Algol Using Regular Languages. In Proceedings of *the International Colloquium on Automata, Languages and Programming (ICALP)* , LNCS **1853**, (2000), 103–116. Springer.

[49] D. R. Ghica. Regular Language Semantics for a Call-by-Value Programming Language. In Proceedings of *the Conference on Mathematical Foundations of Programming Semantics (MFPS)* , ENTCS **45**, (2001), 85–98, Elsevier.

[50] D. R. Ghica. A Games-Based Foundation for Compositional Software Model Checking. PhD Thesis, Queen's University, Ontario, Canada, 2002.

[51] D. R. Ghica and G. McCusker. The Regular-Language Semantics of Second-order Idealized Algol. *Theoretical Computer Science* **309**, (2003), 469–502.

[52] D. R. Ghica and A. Murawski. Angelic Semantics of Fine-Grained Concurrency. In Proceedings of *the International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, LNCS **2987**, (2004), 211–225. Springer.

[53] D. R. Ghica, A. Murawski, and C.-H. L. Ong. Syntactic Control of Concurrency. In Proceedings of *International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS **3142**, (2004), 683–694. Springer.

[54] D. R. Ghica. Slot games: a quantitative model of computation. In Proceedings of *the ACM Symposium on Principles of Programming Languages (POPL)*, (2005), 85–97. ACM Press.

[55] D. R. Ghica and A. Murawski. Compositional Model Extraction for Higher-Order Concurrent Programs. In Proceedings of *the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS **3920**, (2006), 303–317. Springer.

[56] J.-Y.Girard. Linear Logic. *Theoretical Computer Science* **50(1)**, (1987), 1–102.

[57] J.-Y.Girard. Towards a geometry of interaction. In J.W.Gray and A.Scedrov, editors, *Categories in Computer Science and Logic, Contemporary Mathematics 92*. (AMS, 1989), 69–108.

[58] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Proceedings of *the International Conference on Computer Aided Verification (CAV)*, LNCS **1254**, (1997), 72–83. Springer.

[59] A. Groce, D. Peled, and M. Yannakakis. Adaptive Model Checking. In Proceedings of *the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS **2280**, (2002), 357–370. Springer.

[60] O. Grumberg, and D. E. Long. Model Checking and Modular Verification. In Proceedings of *the International Conference on Concurrency Theory (CONCUR)*, LNCS **527**, (1991), 250–265.

[61] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. (MIT Press, 1992).

[62] C. Hankin and P. Malacaria. Program analysis games. *ACM Computing Surveys* **31(3es):5**, (1999).

[63] R. Harmer and G. McCusker. A Fully Abstract Game Semantics for Finite Nondeterminism. In Proceedings of *the IEEE Symposium on Logic in Computer Science (LICS)*, (1999), 422–430. IEEE Computer Society Press.

[64] R. Harmer. Games and Full Abstraction for Nondeterministic Languages. PhD thesis, Imperial College, London, UK, 1999.

[65] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science* **138(2)**, (1995), 353–389.

[66] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In Proceedings of *the International SPIN Workshop on Model Checking of Software (SPIN)*, LNCS **2648**, (2003), 235–239. Springer.

[67] T. A. Henzinger, S. Qadeer, and S. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In Proceedings of *the International Conference on Computer Aided Verification (CAV)*, LNCS **1427**, (1998), 440–451. Springer.

[68] J. Hintikka and G. Sandu. Game-theoretical Semantics. In J. van. Benthem, editor, *Handbook of logic and language.* (Elsevier Science, 1997).

[69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM* **12(10)**, (1969), 576–580.

[70] C.A.R. Hoare. *Communicating Sequential Processes.* (Prentice Hall, 1985).

[71] G. J. Holzmann. *SPIN Model Checker.* (Addison Wesley Professional, 2004).

[72] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation (extended abstract). In Proceedings of *the International Collo-*

quium on Automata, Languages and Programming (ICALP), LNCS **1853**, (2000), 103–115. Springer.

[73] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* (Addison Wesley Longman, 2001).

[74] J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I, II, and III. *Information and Computation* **163(2)**, (2000), 285–400.

[75] C. B. Jones. Specification and Design of (Parallel) Programs. *Information Processing* **83**, (1983), 321–332.

[76] J. Laird. A Fully Abstract Game Semantics of Local Exceptions. In Proceedings of *the IEEE Symposium on Logic in Computer Science (LICS)*, (2001), 105–114. IEEE Computer Society Press.

[77] P. Lorenzen. Logik and Agon. *Atti del Congresso Internazionale di Filosofia*, (1960), 187–194.

[78] S. Mac Lane. *Categories for the Working Mathematician.* (Prentice-Verlag, 1971).

[79] P. Malacaria and C. Hankin. Generalised flowcharts and games. In Proceedings of *the International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS **1443**, (1998), 363–374. Springer.

[80] P. Malacaria and C. Hankin. A New Approach to Control Flow Analysis. In Proceedings of *the International Conference on Compiler Construction (CC)*, LNCS **1383**, (1998), 95–108. Springer.

[81] P. Malacaria and C. Hankin. Non-deterministic games and program analysis: An application to security. In Proceedings of *the IEEE Symposium*

on Logic in Computer Science (LICS), (1999), 443–452. IEEE Computer Society Press.

[82] G. McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types.* Distinguished Dissertation in Computer Science, (Springer-Verlag, 1998).

[83] K. L. McMilan. *Symbolic Model Checking.* (Kluwer Academic Press, 1993).

[84] S. Merz. Model Checking Techniques for the Analysis of Reactive Systems. In B. Lwe and F. Rudolph, editors, *Foundations of the Formal Sciences.* (Springer, 2000).

[85] R. Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science* **4**, (1977), 1–22.

[86] R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML.* (MIT Press, 1990).

[87] A. Murawski and I. Walukiewicz. Third-Oreder Idealized Algol with Iteration is Decidable. In Proceedings of *the International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, LNCS **3441**, (2005), 202–218. Springer.

[88] G. J. Myers. *The Art of Software Testing.* (John Wiley & Sons, 1979).

[89] J. Nash. Equilibrium points in n-person games. Schachspiels. In Procedings of *the National Academy of Sciences of the United States of America*, (1950), 48–49.

[90] H. Nickau. Hereditarily sequential functionals. In Proceedings of *the Symposium on Logical Foundations of Computer Science (LFCS)*, LNCS **813**, (1994), 253–264. Springer.

[91] C. Pasareanu, M. Dwyer, and W.Visser. Finding feasible counter-examples when model checking abstracted Java programs. In Proceedings of *the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS **2031**, (2001), 488–497. Springer.

[92] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science* **5**, (1977), 223–255.

[93] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. *Logic and Models of Concurrent Systems* **13**, (1984), 123–144.

[94] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Proceedings of *the International Symposium on Programming*, LNCS **137**, (1982), 337–351. Springer.

[95] J. C. Reynolds. The essence of Algol. In P.W.O'Hearn and R.D.Tennent, editors, *Algol-like languages.* (Birkhaüser, 1997).

[96] R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, **103(2)**, (1993), 299–347.

[97] A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson and J.B. Scattergod. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In Proceedings

of *the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS **1019**, (1995), 133–152. Springer.

[98] A. W. Roscoe. *Theory and Practice of Concurrency.* (Prentice-Hall, 1998).

[99] D. Scott. Continuous lattices. In *Toposes, Algebric Geometry and Logic*, LN in Mathematics **274**, (1972), 97–136. Springer.

[100] J. van. Neumann and O. Morgenstern. *The Theory of Games and Economic Behaviour.* (John Weily and Sons, 1944).

[101] G. Winskel. *The Formal Semantics of Programming Languages.* (MIT Press, 1993).

[102] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, **9(2)**, (1997), 149–174.

[103] M. Weiser. Program slicing. In Proceedings of *the IEEE Transactions on Software Engineering (TSE)*, **10(4)**, (1984), 352–357. IEEE Computer Society Press.

[104] E. Zermelo. Uber eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In Procedings of *the Fifth International Congress of Mathematicians*, (1913), 501–504.