Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics

Fabio Q. B. da Silva

Doctor of Philosophy

University of Edinburgh

1992

Abstract

In this thesis we study the use of semantics-based formal methods in the specification and proof of correctness of compilers and debuggers. We use a Structural Operational Semantics as the basis for the specification of compilers and propose a notion of correctness based on an observational equivalence relation. We define program evaluation and a notion of evaluation step based on a Structural Operational Semantics and use these definitions as the basis for the specification of debuggers. Debugger correctness is then defined by an equivalence relation between a specification and an implementation of the debugger based on the bisimulation concept.

The main results of this thesis are: a definition of a variant of Structural Operational Semantics, called Relational Semantics, which is the underlying formalism of this thesis; the definition of a notion of Observational Equivalence between Relational Semantics Specifications; a formulation of the problem of compiler correctness using Observational Equivalence; an evaluation model for programming languages and a definition of an evaluation step; an abstract definition of Interpreter-debuggers; a specification notation for the formal specification of debuggers, called DSL; a notion of equivalence between debuggers using bisimulation; a study on Compiler-debuggers and the problems involved in their definition.

These results form a theory for the formal specification and proofs of correctness of compilers and debuggers. Our starting point is that the use of this theory helps in building better compilers and debuggers. It is our goal to provide theoretical foundations and tools to show that our methods are achievable.

Acknowledgements

I would like to thank first of all my supervisor, Kevin Mitchell, for his help, encouragement, and precise hints in the development of this thesis. I would also like to thank my second supervisor Don Sannella for leading me through the paths of algebraic specification. His suggestions and guidance greatly influenced the results of Chapters 3 and 4.

The Department of Computer Science at Edinburgh provided me with a congenial work environment and I would like to thank all the people I got to know there during my stay. Special thanks to Kees and Hans for being such good office-mates, and to Eleanor for general help.

Silvio Meira influenced my decision to study in Edinburgh, and I thank him for this.

Matthew Morley, Ed Kazmierczak, Kees Goossens, and Dave Berry proof-read several parts of this thesis. My thanks go to them. Any remainning misteaks (including this one) are entirely my responsibility.

Thanks to Dave Berry, Dave Matthews, Mads Tofte, Tim Griffin, and Benjamim Pierce for good discussions on programming language semantics. Thanks also to Oliver Schoett for clarifying some aspects of his thesis to me. Although I never met Laurent Hascoët in person, our "electronic discussions" about TYPOL and CENTAUR were of great help. My thanks also go to him.

Thanks to my parents, Maria Aparecida and Saulo da Silva, and to my sisters, Luciana and Daniela, for their continuous support, and to all friends and relatives in Brazil for their letters. Special thanks go to André Villas Boas and Jair

Lopes. Thanks also to everybody in Edinburgh who kindly distract me from work: Roberto, Jamie, Natalie, Fotini, Mauro, Seraphin, Eliane, Zé Carlos, Pirko, and Mario.

Finally, I would like to thank my wife Cláudia whose support and patience played a major part in making this thesis possible.

This work was supported by a Brazilian government scholarship, CNPq process number 20.0459/88.0/CC.

Declaration

This thesis has been composed by myself. The work reported herein has not been presented for any university degree before, and, unless otherwise stated, is my own.

Fabio Q. B. da Silva

To my grandparents:

Dalila José Olga Francisco

Table of Contents

List of Definitions

List of Examples

List of Figures

List of Theorems

Glossary

The following tables summarise the notation used in this thesis. The page numbers refer to the page where the notation is first defined or used.

Set Theory

Constructor	Description	Page
$\emptyset$	empty set; same as $\{\ \}$	40
$\mathcal{N}$	Natural Numbers	40
$[n_+]$	$\{1, \ldots, n\} \subset \mathcal{N}$	40
$\mathcal{F}(A)$	finite power set of A	44
$\{a \in A : \Phi(a)\}$	set of $a \in A$ such that $\Phi(a)$	41
$[A \rightarrow B]$	function space from A to B	40
$[A \stackrel{\text{fin}}{\rightarrow} B]$	finite mappings from A to B	278
$A \cup B$	union of sets	40
$A \cap B$	intersection of sets	40
$A + B$	union of disjoint sets	40
$A \backslash B$	difference of sets	124

Set Theory (continued)

		Page
$A \times B$	cartesian product	40
dom R	domain of (binary) relation	52
ran R	range or image of (binary) relation	123
R^{-1}	converse of relation; $\{(y,x) : (x,y) \in \mathrm{R}\}$	75
R o Q	composion of relations	75
R/A	restriction of R to A; $\{(x,y) \in \mathrm{R} : x \in \mathrm{A}\}$	141
$\{a_i\}_{i \in \mathrm{I}}$	indexed set or family; $\{a_i\}$, when I understood	40
$\langle a_1, \ldots, a_n \rangle$	finite sequence of length n	40
$\langle\,\rangle$	empty sequence; same as ε	40
A^*	set of all finite sequences (words) over A	40
A^+	$\mathrm{A}^* \backslash \{\varepsilon\}$	42
$\lvert w \rvert$	length of w	167
$w \hat{} w'$	sequence concatenation	182
$\langle a_1, \ldots, a_n \rangle \diamond a_{n+1}$	$\langle a_1, \ldots, a_n, a_{n+1} \rangle$	157
$fst(\langle a_1, \ldots, a_n \rangle)$	a_1	157
$cur(\langle a_1, \ldots, a_n \rangle)$	a_n	157
$rem(\langle a_1, \ldots, a_n \rangle)$	$\langle a_1, \ldots, a_{n-1} \rangle$	157
$B^{\langle a_1, \ldots, a_n \rangle}$	$B_{a_1} \times \ldots \times B_{a_n}$	40
B^{ε}	$\{\emptyset\}$	40
$(b_1, \ldots, b_n)$	n-tuple, i.e., $(b_1, \ldots, b_n) \in B^{\langle a_1, \ldots, a_n \rangle}$	40

Single Entities

Entity Set	Entity Name	Type of Entity	Page
S	s	sort	42
F	σ	function name	42
Π	π	relation name	42
S^*	w	sequence of sorts	42
X	$x, \ldots$	meta-variable	42
$T_X(\Sigma)$	$t, t', \ldots$	Σ-terms with meta-variables in X	42
$F_X(\Omega)$	$f, f', \ldots$	Ω-formula with meta-variables in X	43
$P_X(\Omega)$	$p, p', \ldots$	program formula with meta-variables in X	99
$T(\Sigma)$	$\overline{t}, \overline{t}', \ldots$	ground Σ-terms	42
$F(\Omega)$	$\overline{f}, \overline{f}', \ldots$	ground Ω-formula	43
$P(\Omega)$	$\overline{p}, \overline{p}', \ldots$	ground program formula	99
	ϕ, ψ	Relational Inductive Definition	44

Compound Entities

Entity Set	Entity Name	Type of Entity	Entity	Page
	$\Sigma, \Sigma_{OBS}, \ldots$	algebraic signature	(S, F)	42
Sig	$\Omega, \Omega_{OBS}, \ldots$	first order signature	(S, F, Π)	42
	$\Omega, \Omega_{OBS}, \ldots$	directed signature	(S, F, Π, π)	99
$Alg(\Sigma)$	$A, B, \ldots$	partial Σ-algebra		51
$Mod(\Omega)$	$M, N, \ldots$	first order Ω-model		53
	$\mathcal{C}, \mathcal{S}, \mathcal{R}, \ldots$	Relational Specification	(Ω, ϕ, A)	52
$Spec(\Omega_{OBS})$	$\mathcal{C}, \mathcal{S}, \mathcal{R}, \ldots$	Relational Specification	$(\Omega, \phi, A, \Omega_{OBS})$	77
$Deb(\Sigma D_L, \Sigma_{OBS})$	$\mathcal{D}, \mathcal{E} \ldots$	Debugger-schema	$(\Sigma, \Delta, A, \Sigma D_{OBS})$	200

Substitutions

Constructor	Description	Page
$\mathcal{V}(t)$	meta-variables of t	43
θ	substitution	43
$t\theta$	instance of t by θ; same as $\mathcal{I}(t,\theta)$	43
$\theta\theta'$	composition of substitutions	44

Inductive Definitions

Constructor	Description	Page
(P,c)	inductive rule	40
ϕ	inductive system	40
$\mathrm{I}(\phi)$	inductively defined set	41
ϕ^{A}	inductive system derived by A	52

Universal Algebra Constructors

		Page
ψ^{A}	evaluation function of terms on algebra A	51
Ψ^{A}	evaluation function of formulae on algebra A	52
χ^{A}	simplification function on algebra A	123
ρ	signature morphism	65
$\Sigma \sqsubseteq \Sigma'$	algebraic signature inclusion	66
$\Omega \sqsubseteq \Omega'$	signature inclusion	65
A/Σ	the reduct of A by Σ	66
M/Ω	the reduct of M by Ω	65

Transition Systems

Set	Element	Description	Page
Γ	$st, \ldots$	states	124
	$\rightarrow$	transition relation	124
	$\rightarrow^*$	transitive-reflexive closure of $\rightarrow$	124
	$\rightarrow^+$	transitive closure of $\rightarrow$	124
$\mathcal{T}$	$st, \ldots$	terminal states	124
	$\langle \Gamma, \rightarrow \rangle$	transition system	124
	$\langle \Gamma, \rightarrow, \mathcal{T} \rangle$	terminal transition system	124
	$\langle \Gamma, \mathrm{L}, \rightarrow \rangle$	labelled transition system	156

Computational Semantics

Set	Element Name	Element	Description	Page
	$\mathcal{CS}$	$(\Gamma_{\mathrm{CS}}, \overset{\mathrm{CS}}{\Rightarrow}, \mathcal{T}_{\mathrm{CS}})$	Computational Semantics	126
Γ_{CS}	$st, \ldots$	$[\theta : \alpha]$	evaluation state	126
EH	$h, dh \ldots$	$\langle st_1, \ldots, st_n \rangle$	evaluation history	157

Transition Relations

Relation	Type	Description	Page
$\overset{\mathrm{CS}}{\Rightarrow}$	$\Gamma_{\mathrm{CS}} \times \Gamma_{\mathrm{CS}}$	computational semantics	126
$\overset{\mathrm{EH}}{\Longrightarrow}$	EH $\times$ EH	evaluation history transition	157

Equivalence Relations

Relation	Type of Relation	Page
$\equiv_{\Omega_{\mathrm{OBS}}}$	Observational Equivalence	67
$\Longrightarrow\!\!\times_{\Omega_{\mathrm{OBS}}}$	Strong Correspondence	70
$\longleftrightarrow_{\Omega_{\mathrm{OBS}}}$	Model Correspondence	72
$\cong$	Equivalence of Relational Specifications	82
$\approx$	Debugger Bisimulation	202

One talks about mathematical discoveries. I shall try again and again to show that what is called a mathematical discovery had much better be called a mathematical invention.

In some cases to which I point, you will perhaps be inclined to say, "Yes, they had better be called inventions."; in other cases you may perhaps be inclined to say, "Well, it is difficult to say whether in this case something has been discovered or invented.

Ludwig Wittgenstein [Dia76, page 22]

Chapter 1

Introduction

This thesis examines two related issues in the implementation of programming languages: the specification and correctness proofs of compilers, and the specification and correctness proofs of debuggers. While this thesis focuses on the latter problem we use, for both problems, a *Structural Operational Semantics* of the programming language as the reference point for correctness.

In the following section we motivate the use of semantics-based methods in the design of compilers and debuggers. In Section 1.2, we give an overview of our approach to this problem. Section 1.3 briefly reviews the approachs to compiler correctness and semantics-based debugging that we find in the literature, and presents an outline of this thesis.

1.1 The Design of Compilers and Debuggers

The research on compiler design dates back to the early sixties. Since then, a vast literature has emerged describing techniques and tools for compiler writing (e.g. [Gri71,TS85,ASU86,PP92]). The problem of compiler correctness has also been widely studied in the past [MP67,Mor73,Mos79,TWW81,Pol81,CM86, Des86,Joy89,Sim90]. Therefore, we have a common understanding of what the problems of compiler specification and correctness involve: we must give a definition of a compiler and prove that the code it generates for each program executes

consistently with the semantics of the programming language.

Since there exists some freedom in defining what is meant by "consistent execution of the compiled code", it is in this aspect that the works in the literature differ most. This thesis extends and improves previous work on this subject by presenting a notion of compiler correctness in which the execution of the code of a program must be *observationally equivalent* (in the algebraic specification sense) to the semantics of the program as defined by a Structural Operational Semantics.

In contrast with compilers, debuggers have received little attention from a theoretical point of view. Therefore, it is not commonly agreed what the problems involved in the design of debuggers are, nor what debugger correctness means. However, debugging is an important phase in the development of programs accounting for a large percentage of the cost of this development [Jon77,Sho83]. For instance, in the telecommunication industry this cost may account for over 50 percent of the total development cost of a program [Sev87]; similar figures have been reported from other areas.

Therefore, an important problem in software technology is to define methods and tools to reduce the time spent on the debugging phase [Lew82]. This reduction can be achieved either by producing programs that have fewer errors with respect to their specification (no errors in the ideal case), or by improving the quality of the tools and methods used in debugging, or by a combined solution.

The problem of developing programs that are correct with respect to their specification is the subject of a wide area of research on formal specification and formal program development. We do not treat this problem in this thesis; the interested reader is directed to [GM86] for an overview of various approaches to formal program development and to [BKL$^+$91] for a survey on Algebraic Specification.

Although the research on formal specification experienced great advances in the last decade, it has not reached a state in which practical programs are de-

veloped entirely free of errors. Therefore debuggers are still necessary in the process of program development, and to improve the techniques and tools used in their design is an important problem to be addressed. We approach this problem by proposing the use of semantics-based formal methods in the design of debuggers. Our objective is to define a theory to address *formal specification*, *implementation*, and *correctness proofs* of such tools.

In the rest of this section we will study the process of *debugger design* and identify aspects in this process that can be improved by using semantics-based methods. Before we start looking into debugger design we should agree on what a debugger is and what distinguishes it from other programming tools. First, we are interested in *automated debuggers* rather than in manual debugging techniques like desk checking and memory dump analysis. As an initial proposal, we characterise debuggers as follows:

> A debugger is a tool that produces *information* about the *intermediate states* of the *evaluation* of programs under the *user's request.*

This characterisation emphasises that we are interested in *dynamic information* about a program rather than in a static analysis of its behaviour as in [Sev87]. We will refine and make this characterisation more precise throughout this chapter. Let us now analyse some questions that naturally arise from the above characterisation of debuggers:

- How are programs evaluated?
- What information about the evaluation may be requested?
- Is the process of requesting/obtaining information interactive?

The answers to these questions vary in the literature and define classes of debuggers. Programs may be interpreted, in which case the debugger is called an *interpreter-debugger* [vdLW85,SYO87]. Alternatively, programs may be compiled

into machine code that is then executed on the machine, in which case the debugger is known as a *compiler-debugger.* Finally, programs may be evaluated by a combination of interpretation and compilation, usually known as *mixed-execution* or *selective interpretation* [CH87].

The minimal information about the evaluation that is normally available is the value of the program data at intermediate states of the program evaluation. More sophisticated debuggers provide procedure and function trace-backs, and information about the control flow of the program. *Symbolic-debuggers* are debuggers in which this information is requested and presented at the level of the programming language structures. Finally, *interactive-debuggers* are those in which the process of requesting/obtaining information is interactive [Zel84, vdLW85,SYO87], in contrast with *post-mortem debuggers* in which the user cannot interact with the debugging process, and the information is delivered after program termination [Lau79].

Most proposals in the literature which set out to improve the quality of debugger focus on two main aspects: the *user interface* and the *information* that the user can access through the debugger. The recent advances in hardware technology and the wide availability of graphical workstations has made possible the design of debuggers with sophisticated graphic interfaces [Bov87,Moh88]. The increasing computational power and storage capabilities of recent computers allows debuggers to store complete histories of the evaluation of the program, so that it is possible to access information about an arbitrary point in the history of the evaluation [Moh88].

The above mentioned works and other related approaches in the literature (e.g. [vdLW85,Bov87]) have produced some improvements on the process of debugger design. However, this process still remains ad hoc and informal which often leads to debuggers that either have a *counter-intuitive behaviour*, or are *incorrect*, or both. Let us illustrate these problems so that we can understand how the use of semantics-based methods can assist in their solution.

Hereafter, we focus our interest on *interactive-debuggers*, for this is the most used kind of debugger and also because they provide the most interesting and challenging problems. In general, an interactive-debugger provides debugging commands that give the user the ability to step through the intermediate states in the evaluation of a program; to stop at desired break points; and to request information about the current, and possibly, previous states in the evaluation.

This characterisation of interactive-debuggers raises an important question: what is the meaning of "to step through the states in the evaluation of a program"? Answering this question involves the definition of a notion of an *evaluation step*, which is a central problem in the design of debuggers.

The importance of the definition of an evaluation step is brought to light in the example below in which we wish to emphasise how a *counter-intuitive* notion of evaluation step may lead to counter-intuitive behaviour of the debugger. In this case, the debugger is less effective in assisting the programmer to locate errors, and solutions to this problem are therefore necessary.

Suppose we have two versions of simple C [KR78] program in which the only difference is how the text of the program is written. The two versions of the program are shown below; we call the one on the left `p1.c` and the other `p2.c`:

```
main(){
  int i;
  i = 0;  while (1) {i++;};
}
```

```
main(){
  int i;
  i = 0;
  while (1) {i++;};
}
```

From our knowledge of the programming language, we develop an *intuition* about the evaluation steps of the above programs. According to this intuition, the first step in both evaluation is to assign `0` to the variable `i`. Then, the condition `(1)` of the `while` statement is tested. Since its value is different from `0`, the next step evaluates the statement `i++`, which increments the value of `i` by `1`. Since the value of `(1)` will always be different from `0`, the evaluation never leaves the

`while` statement, and at each step the statement `i++` is evaluated, incrementing `i` by 1.

The above description of the evaluation of `p1.c` and `p2.c` is what we call our *intuitive* notion of an evaluation step. When using a debugger, we expect to be able to observe the evaluation according to this intuitive notion. The following example shows a debugger, commonly used in practical applications, that does not behave as we expect.

Let us debug the programs `p1.c` and `p2.c` using DBX [Mic] to observe the values of the variable `i`. We first compile the programs generating debugging code, and then load DBX with this code. A debugging session for the program `p1.c` is shown below:

```
Reading symbolic information...
Read 34 symbols
(dbx) stop at 3
(2) stop at "p1.c":3
(dbx) run
Running: p1
stopped in main at line 3 in file "p1.c"
   3     i = 0;  while (1) {i++;};
(dbx) display i
i = 0
(dbx) step
^C
(dbx)
```

After the `step` command the execution enters an infinite loop and we cannot see the value of the variable `i` inside the `while` statement. A debugging session for the program `p2.c`, in which the first three debugging commands are as in the above session, is as follows:

```
Reading symbolic information...
Read 35 symbols
(dbx) stop at 3
(2) stop at "p2.c":3
(dbx) run
Running: p2
stopped in main at line 3 in file "p2.c"
   3     i = 0;
(dbx) display i
i = 0
(dbx) step
stopped in main at line 4 in file "p2.c"
   4     while (1) {i++;};
i = 0
(dbx) step
stopped in main at line 4 in file "p2.c"
   4     while (1) {i++;};
i = 2
(dbx)
```

Let us analyse two aspects of the above examples: the behaviour of the debugging command `step` in both debugging sessions; and the values output by DBX for the variable `i` during the evaluation of `p2.c`.

We start by analysing the `step` command. According to our intuition about the evaluation steps of `p1.c` and `p2.c`, we expect both programs to have identical evaluation steps, as we discussed on page 5. Therefore, we also expect the `step` command to have identical behaviours in both debugging sessions. Nevertheless, the behaviours of `step` do not agree with our intuition because DBX's notion of step is based on the lines of the program rather than on the sequence of primitive operations involved in the evaluation of the program.

We argue it is essential that a debugger behaves according to our intuition about how programs are evaluated. A debugger that behaves intuitively is easier to learn and may be more effective in assisting the programmer to locate errors. Therefore, an important problem in debugger design is to build debuggers with *intuitive behaviours.*

The particular problem we illustrated using the `step` command of DBX is an instance of a general problem. Let us characterise this problem to understand how semantic based methods may assist in its solution. On the one hand, as discussed above, we develop an intuitive notion of an evaluation step and consequently of how programs are evaluated. These intuitive notions form our *mental model* of the programming language [BOM81,Nor82] which is built mainly from the observations we make of program evaluations, and from the documentation we are given on the programming language, e.g, the definition of the programming language. On the other hand, a debugger also has a model of the programming language, which defines its notion of evaluation step and how programs are evaluated; we use the term *evaluation model* to refer to the debugger's model of the programming language. Problems therefore arise whenever our mental model and the debugger's evaluation model are different.

An obvious solution to this problem is to define debuggers whose evaluation model behaves close to our mental model. A first aspect in which semantics-based methods may help the design of debuggers is in the definition of an *intuitive evaluation model.* Some formalisms used to define semantic aspects of programming languages have an explicit operational meaning, e.g., definitional interpreters [Lan64] and Structural Operational Semantics [Plo81]. Other formalisms can naturally be given an operational interpretation, e.g., Natural Semantics [Kah88].

This operational meaning may be used to define evaluation models parametric on a formal semantics of the programming languages; when instantiated with a particular semantics such an abstract evaluation model yields an actual evaluation model that may be used to evaluate programs. For instance, this is the approach taken by Berry in his Animator Generator [Ber91a].

Our starting point is that, using a suitable semantics formalism and a particular intuitive semantics of the programming language, we can define an evaluation model that has an intuitive notion of an evaluation step. Our main contention is that the use of such an evaluation model helps in the definition of debuggers that have *intuitive behaviour*. It is our first objective to choose a suitable semantic formalism and then to define an evaluation model based on this formalism.

Let us now analyse the values output by DBX for the variable `i` in the debugging session for program `p2.c` (page 6). In that debugging session, the value of `i` is shown as incrementing by two at each step. However, we intuitively expect it to be incrementing by just one. Since there is no formal specification of DBX, there are two possible interpretations for this behaviour. First, this is the behaviour intended by the debugger designer, in which case the behaviour of DBX is again counter-intuitive. Second, this behaviour is not intentional, in which case it is an *error* in DBX's implementation.

The second possibility is the most likely and is clearly the most harmful because if we use a debugger to locate errors in the program it is imperative that the debugger is *correct*. Another important aspect in debugger design is to produce *correct debuggers*.

Another aspect in which semantics-based methods may assist in the debugger design is in the *formal specification* of the behaviours of the debuggers. The ability to build formal specifications is the first essential step towards correct debuggers. Furthermore, a formal specification is an improvement on an informal one because it offers an unambiguous common reference for the users, the designers, and the implementors of the debugger, with the following advantages:

- A common reference between user and designer means that the behaviour of the debugger that the user learns is exactly what was specified by the designer.

- A common reference between the designer and the implementor means that the implementor may implement a debugger that has the behaviour specified by the designer in the formal specification. Moreover, because the

specification is formal we can develop a notion of correctness between specification and implementation such that we can prove that the implemented debugger has the same behaviour as its specification.

- As a consequence of the two previous points, a common reference between the implementor and the user means that the latter uses a concrete debugger that behaves exactly like the specification.

Our second objective is to define a theory for the *specification of debuggers* based on a formal semantics of the programming language. In this semantics-based approach, an evaluation model of the programming language is used as the basis of the debugger specification. Since this evaluation model will be defined to behave as close as possible to our intuition about the programming language, we argue that our approach will assist in formalising debuggers with intuitive behaviours.

Our third and final objective is to define a notion of *equivalence* between the behaviours of a specification and an implementation of a debugger. The definition of an *evaluation model*, the methods and tools for *semantics-based specification* of debuggers, and the notion of *equivalence* between specification and implementation of debuggers form an integrated *theory of debugger design*. Our thesis is that the use of this theory helps in designing debuggers that more effectively assist the programmer because they are easier to learn, more intuitive to understand, and correct with respect to their formal specification. It is our goal to provide theoretical foundations and tools to show that our methods are achievable.

1.2 The Approach of this Thesis

In this section we give an overview of the main results of this thesis. The objective is to show how a *theory of debugger design* can be defined. The *design of a debugger* involves three separate phases. First, the *specification*, in which the

behaviour of the debugger is defined. Second, the *implementation*, i.e., another specification of the debugger given at a lower level of abstraction, and closer to an actual machine implementation. Third, the *proof of correctness* of the implementation with respect to the specification. The following discussions show how each phase is addressed in this thesis.

Choosing the Semantic Formalism

Our ability to construct formal specifications of debuggers depends on the existence of a formal notion of program evaluation and of evaluation step; we propose to use a formal semantics in the definition of programming languages and from such a semantics to derive these formal notions. The success of this proposal depends on the choice of a suitable semantic formalism, for on the one hand it is difficult to derive a notion of program evaluation from some formalisms (e.g., axiomatic semantics [Hoa69]). On the other hand, some formalisms are difficult to reason about because of the complexity of their underlying mathematics (e.g. denotational semantics [Sto89]). Using such a formalism could make it difficult to reason about debuggers.

However, some semantic formalisms have a simple underlying mathematics and a natural and explicit operational meaning that can be used in the definition of program evaluation. The generic name *operational semantics* is often used to refer to such a formalism. In fact, operational semantics is a class of formalisms that includes a diversity of styles of formal semantics: for instance, definitional interpreters (e.g., Landin's SECD machine [Lan64], or Milner's SMC machine [Mil76]), Plotkin's Structural Operational Semantics [Plo81], and Kahn's Natural Semantics [Kah88].

Structural Operational Semantics is a representative example of what we call the *transitional* style of operational semantics, in which the semantics of the programming language is defined by a transition system whose steps describe the evaluation of the programs. On the other hand, Natural Semantics is an example of what we call the *relational* style in which the semantics of a programming

language is defined by a mathematical relation between programs and results.

The main difference between formalisms in the transitional and the relational styles that is relevant for our work at this stage is in the notion of an evaluation step. On the one hand, such a notion is an explicit component of formalisms in the transitional style. On the other hand, for formalisms in the relational style a notion of evaluation step must be defined as a component outside the formalism. This seems to imply that because our goal is a definition of an evaluation step for the debuggers, we should use a semantic formalism in the transitional style. This avoids the task of defining an explicit notion of evaluation step that is necessary if the chosen formalism is in the relational style of operational semantics.

Nevertheless, simple comparisons between concrete semantics written in Structural Operational Semantics and in a relational style, revealed that the latter provides semantics that are more concise and easier to reason about ([Ber91a, pages 48–50], where what we call a Structural Operational Semantics is called a *transition semantics*). The use of *evaluation contexts* proposed in [WF91] helps in making Structural Operational Semantics more concise. However, this approach does not solve a limitation of the formalism: certain language constructors can only be defined by a Structural Operational Semantics provided the language has a rich enough set of constructors. For instance, a `while` statement can only be defined provided the language has some kind of conditional statement.

We propose to use the relational style of operational semantics for the definition of semantic aspects of programming languages in this work. Therefore, we shall need to give an explicit definition of evaluation step to be used by the debuggers, as will be discussed later on.

Let us illustrate the semantic formalism that we shall use in the rest of this work, which we will call *Relational Semantics*. For this, suppose we have a simple language *Sum* of constant and sum expressions, defined by the following BNF rules:

$$\frac{}{\mathbf{num}(n) \Rightarrow n}$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 \; + \; e_2 \Rightarrow \mathbf{plus}(n_1, n_2)}$$

Figure 1: A Semantics of *Sum*

$$\begin{array}{lcl} exp & ::= & \mathbf{num}(nat) \mid exp \; + \; exp \\ nat & ::= & 0 \mid 1 \mid \cdots \end{array}$$

and a function **plus** : $nat \times nat \rightarrow nat$ that takes two *nat* numbers as arguments and returns a *nat* number that is the sum of its arguments. It is convenient to emphasise that "+" is the language constructor whereas **plus** denotes the mathematical sum operation. Moreover, **num** is a coercion operator that constructs an *exp* expression from a *nat* number. The rôle of coercion operators will be clarified in Chapter 2. A Relational Semantics for the expressions in this language may be given by the set of inference rules in Figure 1.

The rules in Figure 1 define how to evaluate a *Sum* expression to a *nat* number: the first rule states that a constant expression $\mathbf{num}(n)$ evaluates to the *nat* number n; the second rule states that if the expression e_1 evaluates to a number n_1 and the expression e_2 evaluates to a number n_2 then the sum expression $e_1 \; + \; e_2$ evaluates to the result of the function call $\mathbf{plus}(n_1, n_2)$. This informal interpretation is what we call an *operational interpretation* of the rules; such an interpretation will be essential in the definition of an intuitive notion of evaluation step based on the Relational Semantics formalism, as will be discussed later on.

The rules in Figure 1 can also be interpreted as an inductive definition of the binary relation $\Rightarrow$ between *exp* and *nat*, which we denote by $\Rightarrow$: $exp \times nat$. We formalise this interpretation in Chapter 2, whereas the operational interpretation is defined in Chapter 5.

$$\frac{}{\mathbf{num}(n) \Rightarrow n}$$

$$\frac{e_1 \Rightarrow v' \qquad (v', e_2) \rightarrow v}{e_1 + e_2 \Rightarrow v}$$

$$\frac{e_2 \Rightarrow v' \qquad (v', e_1) \rightarrow v}{e_1 + e_2 \Rightarrow v}$$

$$\frac{e \Rightarrow v'}{(v, e) \rightarrow \mathbf{plus}(v, v')}$$

Figure 2: An Alternative Semantics for *Sum*

Equivalence between Definitions of Programming Languages

The same semantic aspect of a programming language may be defined by different semantic specifications. For instance, we could define the semantics of the language *Sum* by another set of rules as in Figure 2.

An important question is whether the set of rules in Figure 2 and Figure 1 define equivalent semantics for *Sum*, for if they do so we may use either of the two sets of rules to determine the result of a *Sum* expression. To answer this question we must first define what we mean by equivalent semantics. Using the operational interpretation of the two sets of rules, a suitable notion of equivalence for the above examples may be as follows:

> For all expressions e in *Sum*, e evaluates to a number n in the set of rules of Figure 1 if and only if e evaluates to n in the set of rules of Figure 2.

Given that the function **plus** is commutative, a simple inspection of the two sets of rules tells us that they are equivalent in the above sense.

However, there are examples of pairs of semantics that we intuitively regard

Direct

$$\frac{}{\mathbf{num}(n) \hookrightarrow \mathbf{num}(n)}$$

$$\frac{p_1 \hookrightarrow p_1' \qquad p_2 \hookrightarrow p_2'}{\mathbf{cons}(p_1, p_2) \hookrightarrow \mathbf{cons}(p_1', p_2')}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{first}(p) \hookrightarrow p_1}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{second}(p) \hookrightarrow p_2}$$

Reverse

$$\frac{}{\mathbf{num}(n) \hookrightarrow \mathbf{num}(n)}$$

$$\frac{p_1 \hookrightarrow p_1' \qquad p_2 \hookrightarrow p_2'}{\mathbf{cons}(p_1, p_2) \hookrightarrow \mathbf{cons}(p_2', p_1')}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{first}(p) \hookrightarrow p_2}$$

$$\frac{p \hookrightarrow \mathbf{cons}(p_1, p_2)}{\mathbf{second}(p) \hookrightarrow p_1}$$

Figure 3: Two Semantics for *Pair*

as being equivalent, but are not equivalent under the above informal notion. This suggests that we should look for a more general notion of equivalence. Let us show an example that illustrates this problem.

Consider a language *Pair* of pairs of numbers, defined by the following BNF rules:

$$pair \ ::= \ \mathbf{num}(nat) \mid \mathbf{cons}(pair, pair) \mid \mathbf{first}(pair) \mid \mathbf{second}(pair)$$

where *nat* was defined in the BNF rules of the language *Sum* (page 12). The sets of rules in Figure 3 define two semantics for *Pair*. The set called **Direct** defines the intuitive semantics in which a *pair* expression is evaluated to another *pair* expression by removing all **first** and **second** constructors. The set called **Reverse** gives a semantics in which a *pair* expression is also evaluated to another *pair* expression without **first** and **second** constructors, but the *pair* expressions of the form $\mathbf{cons}(pair, pair)$ are constructed in reverse order.

According to the notion of equivalence discussed above, **Direct** and **Reverse** do not define equivalent semantics for the language *Pair*. For instance, the *pair* expression $\mathbf{cons}(\mathbf{num}(1), \mathbf{num}(2))$ evaluates to $\mathbf{cons}(\mathbf{num}(1), \mathbf{num}(2))$ in **Dir-**

ect and to **cons**(**num**(2), **num**(1)) in **Reverse**. However, the *pair* expression **first**(**cons**(**num**(1), **num**(2))) evaluates to **num**(1) in both cases. Thus, if we are only interested in results of the form **num**(n) then the difference between **Direct** and **Reverse** becomes irrelevant, in which case we would like to consider the two sets of rules as equivalent semantics of *Pair*. Therefore, we need a more flexible notion of equivalence in which only a subset of the *pair* expressions are required to evaluate to the same result under the two sets of rules, namely those expressions that evaluate to some result of the form **num**(n).

Instances of the above problem arises naturally in the semantics of practical programming languages. For instance, in languages with abstract data types, we may not be interested in the details of the representation of abstract data type values. In functional languages, we are often not interested in the details of the representation of function values. Therefore, semantics that choose different representation for such values can still be considered equivalent if we use a suitable notion of equivalence.

The idea of regarding only a subset of the results of expressions as being relevant, or observable, is known in the area of algebraic specification as observational or behavioural equivalence between algebras. However, a similar notion already appears implicitly in the simulation method introduced by Milner [Mil71]. This idea applied to algebras first appears as the concept of the "semantics" of an algebra in [GGM76].

In Chapter 3 we formalise a notion of equivalence that has the intuitive meaning discussed above; this particular notion of equivalence is based on *observational equivalence* as defined in [ST87]. Also in Chapter 3, we extend the definition of *strong correspondence relations* given in [Sch87,Sch90] to the formalism of Relational Semantics to obtain a practical proof method for proofs of equivalence between semantics. We then use an example to illustrate the use of this proof method.

Compiler Correctness

In order to motivate the following discussion about the compiler correctness problem let us consider a simple stack machine *M-Sum* which has a stack on which we can push *nat* numbers, an accumulator that is used for arithmetic operations, and three instructions that manipulate the contents of the stack and the accumulator: **push** that pushes the value of the accumulator on the top of the stack; **add** that adds the top element of the stack to the accumulator, leaving the result in the accumulator; and **load** that loads a *nat* number into the accumulator. The syntax of the machine language is given by the following BNF rules:

$$\begin{aligned} \mathit{stack} &::= \mathbf{empty} \mid \mathit{nat} \cdot \mathit{stack} \\ \mathit{inst} &::= \mathbf{push} \mid \mathbf{add} \mid \mathbf{load}(\mathit{nat}) \end{aligned}$$

A program in this machine is a sequence of instructions, for instance,

$$\langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle$$

The semantics of this machine may also be defined using a Relational Semantics. In this definition, a machine state is a pair (S, n) where S is a stack and n is the accumulator. The rules in Figure 4 define how machine instructions operate on a machine state to produce the new values of the stack and accumulator. In those rules, the concatenation of two sequences of instructions i_1 and i_2 is denoted by $i_1 @ i_2$.

For instance the *M-Sum* program $\langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle$ evaluating on a state $(\mathbf{empty}, 0)$ produces the state $(\mathbf{empty}, 3)$. This machine may be used as the target machine of a compiler for the language *Sum*. The objective is to translate expressions into machine programs that evaluate to the same result. For instance, the expression $1 + 2$ could be translated into the program $\langle \mathbf{load}(1), \mathbf{push}, \mathbf{load}(2), \mathbf{add} \rangle$.

In Chapter 4 we treat the problem of defining compilers in Relational Semantics and how to establish their correctness with respect to a semantics of the programming language. For the correctness we apply the notion of Observational

$$\frac{}{(S, n) \vdash \langle \mathbf{push} \rangle \rightarrow (n \cdot S, n)}$$

$$\frac{}{(n_1 \cdot S, n_2) \vdash \langle \mathbf{add} \rangle \rightarrow (S, \mathbf{plus}(n_1, n_2))}$$

$$\frac{}{(S, n) \vdash \langle \mathbf{load}(n') \rangle \rightarrow (S, n')}$$

$$\frac{(S, n) \vdash i_1 \rightarrow (S', n') \qquad (S', n') \vdash i_2 \rightarrow (S'', n'')}{(S, n) \vdash i_1 @ i_2 \rightarrow (S'', n'')}$$

Figure 4: A Semantics for *M-Sum*

Equivalence between Relational Semantics that is defined in Chapter 3; this is an important practical application of the notion of equivalence between Relational Semantics.

For instance, a compiler for *Sum* using *M-Sum* as the target machine could be defined by the following set of rules that describes how *Sum* expressions are translated into sequence of *M_Sum* instructions:

$$\frac{}{\mathbf{num}(n) \rightsquigarrow \langle \mathbf{load}(n) \rangle}$$

$$\frac{e_1 \rightsquigarrow c_1 \qquad e_2 \rightsquigarrow c_2}{e_1 + e_2 \rightsquigarrow c_1 @ \langle \mathbf{push} \rangle @ c_2 @ \langle \mathbf{add} \rangle}$$

A definition of the semantics of *Sum* using this compiler may be given by the following rule, which defines how an expression e is evaluated into a *nat* number:

$$\frac{e \rightsquigarrow c \qquad (\mathbf{empty}, 0) \vdash c \rightarrow (\mathbf{empty}, n)}{e \Rightarrow n}$$

The above rule has the following operational interpretation: if a *Sum* expression e compiles to a *M-Sum* machine program c and this program, runing on the state $(\mathbf{empty}, 0)$, produces a state $(\mathbf{empty}, n)$ then the result of the evaluation of e

is n. We use the term *Evaluation by Compilation* to refer to a semantics of a programming language given via its translation into some target machine code. In our approach, the compiler defined by the relation $\rightsquigarrow$ is considered correct if the above definition of the relation $\Rightarrow$ is equivalent to the definition of $\Rightarrow$ of Figure 1.

Compiler correctness has been widely studied in the past (e.g. in [MP67, Mor73,Mos79,TWW81,Pol81,CM86,Des86,Sim90]). Our approach strengthens and improves these previous approaches in two main aspects. First, it gives a more general, yet intuitively sufficient, criterion for correctness. Second, we define a proof method that can be used in practical proofs of correctness.

Program Evaluation and a Notion of Evaluation Step

We discussed above that a Relational Semantics of a programming language may be used to evaluate programs in the language, but we have not yet seen how this may be done. Let $\mathbf{num}(1) + \mathbf{num}(2)$ be a *Sum* expression that we want to evaluate using the semantics of Figure 1. By viewing the rules as an inductive definition of the relation $\Rightarrow$ between *exp* expressions and *nat* numbers, we can build a proof tree [DF87] for the formula $\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n$, where n is a meta-variable. The process of constructing such a tree finds an instantiation for n that is the result of the evaluation of $\mathbf{num}(1) + \mathbf{num}(2)$.

We are not interested in the details of how to build proof trees. However, let us show the complete proof tree for $\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n$ as an illustration:

$$\frac{\dfrac{}{\mathbf{num}(1) \Rightarrow 1} \qquad \dfrac{}{\mathbf{num}(2) \Rightarrow 2}}{\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow 3}$$

where $\mathbf{plus}(1, 2) = 3$. The root of this tree is an instance of the rule that defines "+" in Figure 1, and the leaves are instances of the rule for **num** in the same figure.

Proof tree construction is a possible method for evaluating programs using a Relational Semantics of the programming language. An implementation of an

algorithm for constructing proof trees that is parametric on a set of rules provides an abstract evaluation model that may be used to generate concrete evaluation models, or interpreters, for programming languages based on their semantics. This idea was first used in the CENTAUR system [CIK89] , and has also been used in other more recent works [Chi89,Ber91a].

In principle, we could use such an evaluation model as the evaluation model of the programming language to be used by debuggers, as discussed in Section 1.1. Then, the only remaining problem would be to define a notion of evaluation step based on this evaluation model. In [Ber91a], Berry defines such a notion of evaluation step as a function between partial proof trees (proof trees in which some sub-trees are not constructed); this function characterises a depth-first left-to-right traversal of a proof tree.

However, Berry's definition of evaluation step involves various complex definitions that make it difficult to understand and reason about, so cluttering the intuitive understanding of the notion of step. This complexity is necessary in that case because of the requirements of Berry's "theory of program animation". Debuggers in our framework do not have such requirements; thus we should seek a simpler notion of evaluation step that is intuitive and easier to reason about.

We propose an evaluation model for programming languages inspired by the SOS semantics of [And91]. This evaluation model is parametric on a Relational Semantics and is defined by a transition system whose states, called *evaluation states*, contain the current state of the evaluation of the program (i.e., the result of evaluated sub-programs) and the sub-programs or *goals* that still need to be evaluated. A transition step of this system uses the rules of the semantics to decompose a current goal into sub-goals to update the current state and advance the evaluation. For instance, suppose that $\mathbf{num}(1) + \mathbf{num}(2)$ is the expression to be evaluated; an initial state of the transition system is as follows:

$$[\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n]$$

In this evaluation state $\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n$ is the goal. A transition step

from this state advances the evaluation as follows:

$$[\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n] \overset{\text{CS}}{\Rightarrow} [n = \mathbf{plus}(n_1, n_2) : \mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2]$$

where $\overset{\text{CS}}{\Rightarrow}$ denotes a transition of the system, and $n = \mathbf{plus}(n_1, n_2)$ indicates that the meta-variable n is substituted by the term $\mathbf{plus}(n_1, n_2)$ in the current state. A substitution of terms for meta-variables is an explicit part of every state of the transition system. We omit the irrelevant parts of the substitutions from the states to simplify the presentation of the examples.

In the above transition, the resulting state indicates that the next goal is to evaluate $\mathbf{num}(1)$ producing a result n_1, then to evaluate $\mathbf{num}(2)$ producing a result n_2, and finally to compute the sum $\mathbf{plus}(n_1, n_2)$. This transition is defined using the rule for "+" expression of Figure 1 to expand the goal $\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow n$ into the sub-goals $\mathbf{num}(1) \Rightarrow n_1$ and $\mathbf{num}(2) \Rightarrow n_2$, each of which corresponds to a premiss of the rule. The next step in the evaluation is as follows:

$$[n = \mathbf{plus}(n_1, n_2) : \mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2] \overset{\text{CS}}{\Rightarrow}$$
$$[n = \mathbf{plus}(1, n_2), n_1 = 1 : \mathbf{num}(2) \Rightarrow n_2]$$

which is defined using the rule for the definition of $\mathbf{num}(n)$ expressions of Figure 1. After this transition the resulting state indicates that $\mathbf{num}(1)$ has been evaluated to 1 and $\mathbf{num}(2) \Rightarrow n_2$ is the current goal. Finally, we can apply a transition step to obtain the final state of the evaluation and the result of the sum:

$$[n = \mathbf{plus}(1, n_2), n_1 = 1 : \mathbf{num}(2) \Rightarrow n_2] \overset{\text{CS}}{\Rightarrow}$$
$$[n = \mathbf{plus}(1, 2), n_1 = 1, n_2 = 2 : \varepsilon]$$

where ε indicates that there are no more sub-goals and the evaluation terminated successfully. The substitution in the final state gives the result of the evaluation, in this case $n = \mathbf{plus}(1, 2) = 3$.

In Chapter 5 we formalise the transition system discussed above, which we call the *Computational Semantics* of the Relational Semantics formalism. This transition system is used as the evaluation model for programming languages; the

notion of evaluation step used by the debugger is given by the transition relation of this transition system. We give some evidence that this notion of evaluation step is indeed intuitive. Therefore, this transition system and the formalisation of Relational Semantics form the basic components of a theory of debugger design.

This transition system defines an evaluation model for a wide class of programming languages, including non-deterministic languages. Moreover, it defines a deterministic evaluation model for deterministic languages that may be implemented to yield an evaluation model generator for such languages. This provides a prototyping facility for programming languages similar to that provided by the CENTAUR system [CIK89] and the Animator Generator [Ber91a]. The advantages of our transition system over other methods in the literature is that it has an explicit, intuitive notion of an evaluation step.

Formal Specification of Debuggers

The next problem we shall address is how to construct formal specifications of debuggers using the evaluation model discussed above. Let us motivate the following discussion by considering the problems involved in specifying a debugger for the language *Sum*. The first step in this specification is the definition of the *debugging language*, i.e., the language in which we write debugging commands. The following BNF rules define a simple debugging language, called *Pico*:

deb ::= **step** | **trace on** | **trace off**

Recall that using the evaluation model described above we can evaluate *Sum* expressions step by step. Therefore, a possible interpretation for the above debugging language is as follows: **step** causes the evaluation of the current expression to advance by one transition step of the evaluation model; **trace on** switches on tracing mode; **trace off** switches off tracing mode. When tracing mode is on a **step** command outputs the result of the last sub-expression evaluated and the sub-expression that is about to be evaluated. Our goal is to discuss how this

informal interpretation may be formally defined, leading to a specification of the debugger.

There are at least two ways in which to formalise the definition of a particular debugger: in an ad hoc fashion, or by first defining a formal notion of an abstract debugger, and then giving a definition of the concrete debugger that conforms with this abstract notion. The advantages of the second method are that we have an abstract notion of debuggers that can be formally reasoned about, as well as a formal definition of the concrete debugger that is also amenable to formal reasoning. For instance, the definition of equivalence between the behaviour of two arbitrary debuggers, discussed later on, depends on such an abstraction.

To achieve an abstract notion of debuggers we need a more elaborated characterisation of debuggers than that given on page 3:

> A debugger is a tool that receives *debugging commands* from the user, applies those commands to its current *debugging state*, and outputs the information about the evaluation of the program requested by the command as its *result*. In this process the debugger may advance the current state of the evaluation, which is a component of its current debugging state.

In Chapter 6 we formalise the above characterisation leading to an abstract formal notion of debuggers. We also discuss various aspects of the design of concrete debuggers that conform to this notion. In this abstract notion a debugger is a monogenic labelled transition system whose states are the *debugging states* and the labels of the transition relation are the *debugging commands* and their *results*. An evaluation state of a program, according to the evaluation model of the programming language, is a component of each debugging state.

This view of a debugger as a labelled transition system is implicit in the event-action model of debugging proposed by some authors in the literature [BH83, GB85,LL89]. Our work improves on those ideas by the use of formal semantics

in the definition of the programming language, an evaluation model of the programming language in the evaluation of programs, and by developing an abstract definition of debuggers that is amenable to formal reasoning at the level of an arbitrary debugger.

A possible definition of the states of the debugger *Pico* is a pair of the form (h, tr) where: h is a sequence of evaluation states the evaluation model of the *Sum* language and tr is a boolean variable used as the tracing mode flag. If we want to debug the *Sum* expression $\mathbf{num}(1)+\mathbf{num}(2)$, we load the debugger with this expression, which causes the debugger to build an initial debugging state of the form:

$$(\langle[\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow n]\rangle, \mathbf{false})$$

and start the debugging session. A debugging session is an interactive process in which we input debugging commands and the debugger outputs the results; in this process we have no access to the debugging states. If we issue a command **trace on**, the resulting debugging state is:

$$(\langle[\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow n]\rangle, \mathbf{true})$$

If we issue a **step** command on this state, the resulting debugging state becomes:

$$(\langle[\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow n], [\mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2]\rangle, \mathbf{true})$$

with output: **no previous result.**

current expression: num(1)

The evaluation state $[\mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2]$ is obtained from the previous state $[\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow n]$ using $\stackrel{\text{CS}}{\Rightarrow}$. Here we omitted the substitution of $\mathbf{plus}(n_1, n_2)$ for n to simplify the presentation. If we issue another **step** command on the above debugging state, the state changes as follows:

$$(\langle[\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow n], [\mathbf{num}(1) \Rightarrow n_1, \mathbf{num}(2) \Rightarrow n_2], [\mathbf{num}(2) \Rightarrow n_2]\rangle, \mathbf{true})$$

with output: **previous result:** $n_1 = 1$

current expression: num(2)

The debugging session continues until the expression is fully evaluated or we abort the evaluation.

The debugger *Pico* changes only the *st* part of a debugging state (st, tr) by using the evaluation model $\overset{cs}{\Rightarrow}$. For this reason we say that debuggers in our framework are semantic-driven or *based on formal semantics.* This characteristic will be formalised in this thesis as a requirement every debugger must fulfill. Therefore, there is a repertoire of debugging commands in existing debuggers that are disallowed in our framework, e.g., changing the value of programming language variables. If such commands are allowed, a debugger could generate evaluation states that are not reachable from the initial evaluation state of a program using $\overset{cs}{\Rightarrow}$. In this case, we could not regard such a debugger as being *based on formal semantics.*

Once we have developed an abstract characterisation of debuggers, the next problem is to study how we may formally specify concrete debuggers that conform to this characterisation. This problem is studied in Chapter 6 in two stages: we study some aspects of the specification of debuggers and then define a notation to assist in the specification of concrete debuggers. The design aspects studied are generic and the proposed solutions to the problems may be applied to any concrete debugger that uses the evaluation model defined in this thesis in the evaluation of programs.

The specification notation, called DSL, is defined with the objective of making the specification of concrete debuggers simpler and less ad hoc. This is achieved because DSL provides a high-level abstraction of the objects of the evaluation model, and also a powerful set of constructs to express debugging commands. DSL is a specification notation in the sense that definitions written in the language are abstract and concise.

Our major goal in the definition of DSL is to achieve an expressive language in which a useful set of debugging commands, found in most practical debuggers can be specified. We present some examples that show that DSL is indeed expressive.

Nevertheless, the theory of debugger design does not depend on this particular design language; other languages may be defined and used with the theory to suit the needs of particular debuggers.

The notion of an abstract debugger and the DSL notation form a basic framework for the specification of concrete debuggers. This framework is based on the notion of program evaluation and evaluation step discussed above; thus the framework is formal with respect to the semantics of the programming language. Our goal is to show that this framework helps to define debuggers with intuitive behaviours; definitions that can be used as a documentation for the user and guides for implementors.

Implementation and Correctness of Debuggers

In the debugger *Pico*, programs are interpreted using the evaluation model $\stackrel{cs}{\Rightarrow}$. This definition of *Pico* can be viewed as a prototype of the debugger which is simple to specify and understand. Prototypes are useful for testing the functionality of the specified debuggers but they seldom have the performance required in practical applications. Therefore, there is the need for implementations of the debuggers that meet such practical requirements in performance. In this thesis an implementation is another specification of the debugger given at a lower level of abstraction which is often more efficient than the original specification.

As discussed in Section 1.1, the main purpose of formal specifications of debuggers is to serve as a common unambiguous reference for the users, the designers, and the implementors. It is therefore necessary that the implementation of a debugger behave as defined by its specification. The main problem discussed below is how to establish the conditions for an implementation to be correct with respect to a specification of a debugger.

To illustrate the problems involved in implementing debuggers, and their correctness, let us first define another debugger, called *M_Pico*, that has the same debugging language as the debugger *Pico*. The debugger *M_Pico* uses the com-

pilation of *Sum* expressions into *M_Sum* code to evaluate expressions. Therefore, the debugging commands of *M_Pico* work on the machine states instead of on the states of the evaluation model for *Sum*.

Let us consider a modified definition of the machine states in which we add the code that is executing as a component; we write this new state as:

$$(S, \text{Acc}) \vdash c$$

where (S, Acc) is the state of the machine as defined on page 17, and c is a *M_Sum* program. An *M_Pico* debugging state is a pair $[m\_st, tr]$ where m_st is a machine state as above and tr is a tracing mode flag. In *M_Pico* **step** advances the evaluation by one machine step, which corresponds to the evaluation of one machine instruction. The command **trace on** and **trace off** work as in *Pico*, with the difference that here the result of the last sub-expression is the value loaded into the accumulator by the previous instruction and the current sub-expression is the instruction that is about to be evaluated.

If we want to debug the *Sum* expression $\textbf{num}(1) + \textbf{num}(2)$ we load *M_Pico* with the expression; the debugger calls the compiler that translates the expression into its corresponding *M_Sum* code. Finally, the debugger creates the initial debugging state:

$$((\textbf{empty}, 0) \vdash \langle \textbf{load}(1), \textbf{push}, \textbf{load}(2), \textbf{add} \rangle, \textbf{false})$$

and starts the debugging session. We can switch on tracing mode by issuing a **trace on** command, such that the next debugging state is:

$$((\textbf{empty}, 0) \vdash \langle \textbf{load}(1), \textbf{push}, \textbf{load}(2), \textbf{add} \rangle, \textbf{true})$$

If we issue a **step** command on this state the resulting debugging state is:

$$((\textbf{empty}, 1) \vdash \langle \textbf{push}, \textbf{load}(2), \textbf{add} \rangle, \textbf{true})$$

with output: **previous result:** Acc = 1
current instruction: push

Another **step** command produces the debugging state and the output below:

$$\langle (1 \cdot \mathbf{empty}, 1) \vdash \langle \mathbf{load}(2), \mathbf{add} \rangle, \mathbf{true} \rangle$$

with output: **no previous result.**

current instruction: load(2)

Here, since **push** does not load a value into the accumulator, we do not have a previous result. This process continues until the evaluation of the code finishes or we abort the debugging session.

Although *M_Pico* can be used to debug *Sum* expressions we could hardly regard it as a correct implementation of *Pico* in the sense of *having the same behaviour.* The two major reasons for the different behaviours of the two debuggers are:

- There are more steps in the evaluation of the expression according to *M_Pico* than there are using *Pico*; we need to issue four **step** commands in *M_Pico* to entirely evaluate the above expression, whereas it is only necessary to issue three **step** commands in *Pico.*
- *M_Pico* shows machine registers and instructions as the information about the evaluation whereas *Pico* shows source language expressions and results.

The possibility that differences arise between an implementation of a debugger and its specification should be ruled out by a notion of correctness between implementation and specification. If we consider that an implementation of a debugger is also defined as a labelled transition system then we may informally characterise a notion of correctness as an equivalence between debuggers, as follows:

> Two debuggers are equivalent if there exists a one-to-one correspondence between the states of the debuggers such that whenever the two debuggers are at corresponding states each debugging command produces equal results and the debuggers move to corresponding states.

Requiring a one-to-one correspondence between the debugging states rules out the first difference, and requiring that outputs from the same command at corresponding states must be equal makes sure that the implementation will not show machine registers and instructions if the specification does not do so. This notion is sufficient as far as the two aspects discussed above are concerned, but requiring equality between results may be too strong.

Suppose that we define two debuggers for the language *Pair* of Figure 3 with the same debugging language as *Pico*: the debugger called *D* evaluates *pair* expressions using the *Direct* semantics; the debugger *R* evaluates *pair* expressions using *Reverse*. It is not difficult to check that the states of the evaluation in both semantics are in a one-to-one correspondence. However, suppose that at some state in the evaluation of a *pair* expression the result of the last sub-expression evaluated is a *pair* of the form **cons**(*pair*, *pair*) then, since *Reverse* implements such a pair in reversed order the two debuggers will not be considered equivalent according to the above definition.

As in the problem of equivalence between definitions of programming languages, if we are not interested in the representation of *pair* results of the form **cons**(*pair*, *pair*) then we would like to regard the two debuggers as being equivalent. Therefore, we need a notion of equivalence between debuggers that compares the result of debugging commands up to observational equivalence.

In Chapter 7 we define such a notion of equivalence between debuggers, inspired by the bisimularity concept of [Par81] and strong congruence of [Mil89]. We extend such ideas so that labels of transitions are only compared up to observational equivalence. Therefore, equivalence between programming language definitions – in particular, compiler correctness – and equivalence between debuggers are strongly related. This relationship is formally stated in various results in Chapter 7.

These definitions constitute the last components of the theory of design of debuggers. Therefore, in this theory, debuggers may be formally specified using the DSL language and implementations of debuggers may be formally defined and

proved correct with respect to their specifications. The examples throughout the thesis have the objective of demonstrating that, using the theory,

- It is practical to formally define debuggers that have intuitive behaviour.
- It is practical to prove the correctness of debugger implementations.

Compiler-Debuggers

Another important problem treated in this thesis is the study of *Compiler-debuggers* and the problems related with their design. We use the name Compiler-debugger for debuggers in which programs are compiled into machine code and debugging is performed on the execution of the code on the machine. In such a debugger, the debugging commands must be defined on the machine states instead of on a state of the evaluation model of the language; *M_Pico* is an example of a Compiler-debugger.

In Chapter 7 we characterise Compiler-debuggers and study various aspects of their design. In this study we clarify the main aspects in which a Compiler-debugger differs from a specification based on the evaluation model of the programming language, and investigate mechanisms that may be used to resolve these differences. The objective is to define Compiler-debuggers that are correct with respect to a specification of the debugger. The study of the design of Compiler-debuggers clarifies two aspects that were already discussed above:

- Since we expect the evaluation of the machine code of a program to involve more steps than its interpretation under the evaluation model, how can we establish a one-to-one correspondence between those two notions of evaluation step?
- How do we recover the information about the evaluation that is needed by the debugging commands from a machine state and output the results in a source language form?

We treat these problems at the level of an abstract Compiler-debugger so that the proposed solutions may be used with any such a debugger. We also consider other aspects of Compiler-debuggers, e.g., optimiser-debuggers [Hen82, ZJ91], and show that these aspects may have a formalisation in the theory defined in this thesis.

1.3 Outline of the Thesis

In this section we outline the contents of the following chapters, and briefly list various approaches in the literature related to the problems addressed in this thesis. In the introductory section of each chapter we study these approaches in more detail.

In *Chapter 2* we define the syntactical and meta-semantical aspects of the formalism of Relational Semantics. For the syntactical aspects we define an abstract notation in which semantics may be written, characterising a *Relational Semantics Specification.* The meta-semantical aspect involves the definition of the semantics of this notation, which we call the *Declarative Semantics* of the formalism.

In Section 2.1 we compare our approach to Relation Semantics with definitional interpreters [Lan64,Mil76], Structural Operational Semantics [Plo81], Natural Semantics [Kah88], and Inductive Semantics [Ast89,Ast91].

In *Chapter 3* we define a notion of *Observational Equivalence* of Relational Semantics Specification, and extend the proof method of strong correspondence presented in [Sch87] to a proof method for Observational Equivalence between Relational Semantics Specifications, which we call *Model Correspondence.* We also present an example of a proof of equivalence using this method.

In Section 3.1 we study two approaches to equivalence between algebras: Hoare's notion of representation relation [Hoa72] and observational equivalence (e.g. [Rei81,ST87,NO88,Sch90]). We discuss the limitations of representation relation and motivate the use of observational equivalence as a notion of equivalence

between Relational Semantics of programming languages.

In *Chapter 4* we study the problem of compiler correctness in the framework of Relational Semantics. We first characterise *Evaluation by Compilation* as a Relational Specification in which programs are evaluated by the compilation into machine code and the execution of the code on the machine. Then, the notion of Observational Equivalence between Relational Semantics specifications is used as the notion of compiler correctness. We give an example of a compiler correctness proof using the method developed in Chapter 3.

In Section 4.1 we show how the problem of compiler correctness evolved from the early works of McCarthy and Painter [MP67] and Burstall and Landin [BL69] to the algebraic approaches of Morris [Mor73] and the ADJ group [TWW81]. We then discuss the combined use of Natural Semantics and Morris' notion of compiler correctness in the work of Despeyroux [Des86]. Other related approaches are also discussed, e.g., [Mos79,Pol81,CM86,Joy89,Sim90]. Since these approaches use some form of representation relation in the sense of [Hoa72] they suffer from similar limitations. We discuss how Observational Equivalence improves on those approaches.

In *Chapter 5* we define how to evaluate programs using a Relational Semantics Specification of the programming language; this leads to the definition of the *Computational Semantics* of the Relational Semantics formalism. We prove the Computational Semantics is correct with respect to the Declarative Semantics defined in Chapter 2. Therefore, program evaluation is consistent with the Relational Semantics of the programming language. This notion of program evaluation gives rise to a natural notion of an *evaluation step* when the Computational Semantics is used with deterministic languages. We give some evidence that this notion of evaluation step is intuitive.

In Section 5.1 we compare our definition of program evaluation based on a Relational Semantics to the CENTAUR system [CIK89] and Berry's Animator Generator [Ber91a]. The emphasis of this comparison is on the suitability of these approaches to the definition of an intuitive notion of evaluation step.

In *Chapter 6* we treat the problem of formally specifying debuggers. We start by giving an abstract definition of debuggers that uses the notion of evaluation step defined in Chapter 5. This definition characterises the class of *Interpreter-debuggers*. Then we study various aspects of the design of concrete debuggers. Finally, we define a notation to assist in the specification of debuggers, called DSL. We give an example of a debugger specified using DSL.

In Section 6.1 we discuss the features for the specification of debuggers that are found in the CENTAUR and the Animator Generator. We also discuss approaches to semantic based debugging that use denotational semantics as the underlying formalism: the PSG system [BS86] and Monitor Semantics [KHC91].

In *Chapter 7* we study the problems involved in the implementation of debuggers and how to establish a notion of correctness of the implementation with respect to a specification of the debugger. We define a notion of equivalence between debuggers inspired by the notion of bisimularity [Par81] and strong congruence between transition systems [Mil89]. We also treat the problem of the formal specification of Compiler-debuggers by studying various problems involved in the specification of such a debugger. We demonstrate that those problems have a formalisation in our framework. Finally, we give an example of the definition of a Compiler-debugger and a proof of equivalence between this Compiler-debugger and a specification of it constructed in Chapter 6.

In Section 7.1 we compare our definition of equivalence between debuggers with the bisimulation technique of Park [Par81].

In *Chapter 8* we summarise the conclusions of this thesis and give directions for future work.

Finally, in *Appendix A* we give the complete definition of the syntax and semantics of DSL.

Conventions

Definitions, lemmas, theorems, corollaries, and examples are numbered consecutively throughout each chapter. Example 2 of Chapter 3 will be referred to as

Example 2 within Chapter 3 and as Example 3.2 elsewhere. Rules of Relational Semantics that appear in the examples are numbered consecutively throughout each example. Rule 5 of Example 2 of Chapter 3 will be referred to as Rule 5 within Example 2, as Rule 2(5) elsewhere within Chapter 3, and as Rule 3.2(5) elsewhere. The same convention for rules applies to other numbered items in examples, and parts of definitions, lemmas, and theorems.

When a new term is defined it is written in *slanted font.* Terms defined in this thesis start with a capital letter to distinguish from other usage of the same terms in the literature. For instance, Relational Semantics refers to the definition of the formalism that is given in this thesis. Other notational conventions are always defined before their first use. The glossary contains a summary of the notational conventions and the main symbols used in the thesis.

Chapter 2

The Semantics Formalism: Relational Semantics

In this chapter we define the semantic formalism that will be used in the rest of this thesis to define semantic aspects of programming languages. The choice of formalism is primarily guided by the main objective of this thesis: to define a theory of debugger design. An essential component of this theory is a formal notion of program evaluation and evaluation step; thus a requirement on the semantic formalism is that it must have a natural operational interpretation from which these two notions can be derived.

The term Operational Semantics has been used to refer to a semantic formalism in which a programming language is defined by a description of how programs are evaluated. However, Operational Semantics is not a single formalism but a class of styles of formalisms. Relational semantics is such a style in which the semantics of a programming language is defined by a relation between programs and their results. This style has an operational interpretation that makes it suitable for use in a theory of debugger design. Relational Semantics also provides concise and abstract definitions of semantics, and for these reasons is the formalism that will be used in this thesis.

The ways in which a relation between programs and results is defined varies

among the approaches to relational semantics that we find in the literature. We will discuss some of these approaches in Section 2.1. In the rest of this chapter we shall develop an entirely formal definition of a particular approach to relational semantics, which we call *Relational Semantics.*

This formalisation is accomplished by the definition of the syntactic and meta-semantic aspects of the formalism. For the syntactic aspects we define the concept of a *Relational Specification Syntax* in Section 2.3. In Section 2.4 we define the meaning of a Relational Specification Syntax by the definition of its *Declarative Semantics.*

2.1 Introduction

Operational Semantics is the generic name used to refer to semantic formalisms in which programming languages are defined by a description of how programs should be evaluated into their results. In fact, Operational Semantics is a class of formalisms which includes a variety of styles of formal semantics. All formalisms in the Operational Semantics class can be given an operational interpretation which makes them suitable candidates to be used in a theory of debugger design. The choice of a particular formalism is then guided by two main factors: the expressiveness and level of abstractness provided by the formalism, and how intuitive is the notion of evaluation step that may be derived from it.

Definitional Interpreters are the earliest approach to formal Operational Semantics. In this formalism a language is defined by an explicit description of how programs are evaluated by an abstract machine such as Landin's SECD machine [Lan64] or Milner's SMC machine [Mil76]. The main drawbacks of such a description are: the components of the machine state, e.g., memory and stack, must be explicitly manipulated; the semantic operations related to the evaluation of the program are interleaved with operations that mimic the use of a program counter of concrete machines. These problems obscure the understanding of se-

mantic description and give a counter-intuitive notion of evaluation step.

In Plotkin's Structural Operational Semantics [Plo81] a programming language is defined by a transition system whose transition relation describes how to transform a program by steps into its result. Such a transition relation is defined by an inductive definition in a form of inductive rules guided by the (abstract) syntax of the programs, hence the name Structural. This formalism has a clear and simple notion of an evaluation step given by the transition relation itself. However, there is evidence that concrete semantics in this formalism are larger and more difficult to be reasoned about than in other formalisms [Ber91a, pages 48–50].

Kahn's Natural Semantics [Kah87,Kah88] aims for a more abstract account of Structural Operational Semantics by focusing on its logical aspects. A Natural Semantics of a programming language is defined by a deduction system in the form of a sequent calculus [Gen69]. Deduction systems are written and interpreted as sets of natural deduction rules [Pra65], whose premisses and conclusions are sequents. Reasoning about a semantics involves proving that a sequent is a logical consequence of the rules of the semantics. Sequents are composed of two parts: an antecedent, which is in general an environment in which values of program variables are stored, and a consequent, which is a predicate about programs, in general a relation between programs and results. Because the consequents usually define a relation between program and results, Natural Semantics is included in what we call the *relational style* of Operational Semantics; in contrast, Definitional Interpreters and Structural Operational Semantics are called the *transitional style.*

Astesiano's Inductive Semantics [Ast89,Ast91] is essentially the same formalism as Structural Operational Semantics. The goal of Astesiano's work is to provide a broader interpretation of Structural Operational Semantics by relating it to Denotational Semantics.

The relational style of operational semantics, or simply relational semantics,

has been successfully used in the definition of real languages [HMT89]. Concrete semantics in this formalism are often highly abstract, and it is possible to define a notion of evaluation of programs using such a formalism [Ber91a]. Therefore, relational semantics is the style of formalism we use in this thesis. The objective of this chapter is to give a formal definition of the syntactical and meta-semantical aspects of an approach to relational semantics, which we call *Relational Semantics.*

The formalisation of the syntactical aspects of Relational Semantics defines the concept of a Relational Specification Syntax in which concrete Relational Semantics can be expressed. This definition allows syntactic manipulations to be defined over an abstract specification as well as over concrete ones. For instance, in [dS90] we presented an algorithm that performs a syntactical transformation on an abstract Relational Specification Syntax; such transformations require the syntactical aspects of a Relational Semantics to be fully formalised.

The formalisation of the meta-semantics of Relational Semantics allows meta-reasoning about an abstract Relational Semantics, as well as reasoning about concrete ones. For instance, the proof that the transformation algorithm presented in [dS90] transforms the syntax of a Relational Specification preserving its meta-semantics requires this level of meta-reasoning, and is possible only if all aspects of the meta-semantics are formalised.

In the approaches to relational semantics in the literature, it is possible to reason about a particular concrete relational semantics, but this reasoning often does not generalise to the level of an arbitrary semantics. Moreover, in some approaches even reasoning about concrete semantics may be difficult because of the lack of a precise definition of the meta-semantics of the formalism. For instance, the meta-semantics of Kahn's Natural Semantics is formalised by its translation into a concrete meta-language called TYPOL [Des88] and by the compilation of TYPOL into Mu-Prolog [Nai83b]. This meta-semantics is difficult to reason about; moreover it suffers from some of the idiosyncrasies of the underlying Prolog semantics. For instance, the absence of an occurs check in Mu-Prolog allows

infinite objects to be created in a concrete Natural Semantics. Although the use of such objects gives concise definitions, e.g., in the definition of recursive functions, they require more complex proof methods to reason about the semantics.

Our goal in this chapter is to achieve a formalisation of Relational Semantics which allows syntactic manipulations and meta-reasoning over a concrete as well as an abstract Relational Specification. In Section 1.2 we illustrated the formalism that will be defined in this chapter. In that presentation we described an *operational interpretation* for inductive rules of the form:

$$\frac{}{\mathbf{num}(n) \Rightarrow n}$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 \; + \; e_2 \Rightarrow \mathbf{plus}(n_1, n_2)}$$

This operational interpretation shows how to evaluate an expression to a *nat* number through the intermediate evaluation of its subexpressions. Another interpretation for the above rules is as an *inductive definition* of the relation $\Rightarrow$ between expressions and natural numbers. We shall formalise this interpretation in the rest of this chapter. In this formalisation the (abstract) syntax of inductive rules is defined, leading to a notion of Relational Specification Syntax, and the interpretation of the rules as inductive definitions of sets of relations is defined to be the meta-semantics of the formalism, which we call the *Declarative Semantics.*

In Chapter 5, the above mentioned operational interpretation of the inductive rules is used in the definition of a *Computational Semantics* of the Relational Semantics formalism. This Computational Semantics defines how programs are evaluated using a Relational Specification of the programming language. We then prove the consistency of the Computational Semantics with respect to the Declarative Semantics. The Computational Semantics is the basis of the theory of debugger design that will be defined in Chapters 6 and 7.

2.2 Set Theoretical Preliminaries

The set theory of this thesis is the axiomatic set theory of Zermelo-Fraenkel, whose formalisation can be found for example in [Hal74,Kun80]. We assume that the reader is familiar with the basic concepts of set theory. The glossary summarises the basic set theoretical notation used in this thesis. In this section we present some abbreviations and terminology that are not defined in the glossary.

Let A and B be sets, the notation $A + B$ denotes the *union of two disjoint sets*, i.e., $A + B = A \cup B$ for $A \cap B = \{\ \}$. This notation is used when A and B are disjoint to help the reader. If g is a function from A into B, i.e., $g \in [A \rightarrow B]$, we write $g : A \rightarrow B$.

A *family* is a function whose domain is called its *index set*. A family a with index set I, called an I-indexed or I-sorted family, is denoted by $\{a_i\}_{i \in \mathrm{I}}$, or $\{a_i\}$ when I is understood; a_i stands for $a(i)$. If the range of a family is a subset of a set A, the family is called a family of elements of A.

Let B be an I-indexed family of sets, i.e., for each $i \in \mathrm{I}$, B_i is a set. Whenever, the range of B is pairwise disjoint we will write $b \in B$ as an abbreviation for $b \in B_i$ for some $i \in \mathrm{I}$.

A *finite sequence* is a family a whose index set is a set $[n_+]$, for some $n \in \mathcal{N}$. We often use $a_1 \ldots a_n$ as an abbreviation for a finite sequence $\langle a_1, \ldots, a_n \rangle$.

If S is a set (of sorts), $w \in \mathrm{S}^*$ is a word $\langle s_1, \ldots, s_n \rangle$, and A is an S-sorted family of sets then $A^w = A_{s_1} \times \ldots \times A_{s_n}$. An n-tuple is an element of A^w, denoted by $(a_1, \ldots, a_n)$. In this case, A^ϵ is defined to be $\{\emptyset\}$.

The next definitions introduce the basic concepts of the theory of inductive definitions; these definitions can be found in [Acz77].

Let $\mathcal{U}$ be a set. An *inductive rule* is a pair (P, c), where $\mathrm{P} \subseteq \mathcal{U}$ is called the set of premisses, and $\mathrm{c} \in \mathcal{U}$ is called the conclusion of the rule. An *inductive system* is a set of inductive rules, usually denoted by ϕ. An inductive system is a way of defining subsets of a given set $\mathcal{U}$. If ϕ is an inductive system, and A is

some subset of $\mathcal{U}$, we define A to be ϕ-closed if each rule in ϕ whose premisses are in A also has its conclusion in A. More precisely, $A \subseteq \mathcal{U}$ is ϕ-closed if for each $(\mathrm{P}, \mathrm{c}) \in \phi$, $\mathrm{P} \subseteq A$ implies $\mathrm{c} \in A$. If ϕ is an inductive system, then the set inductively defined by ϕ, written $\mathrm{I}(\phi)$, is the least (with respect to set inclusion) ϕ-closed set.

Let S be a set (of sorts) and $\{\mathcal{U}_s\}_{s \in \mathrm{S}}$ be an S-sorted set. An S-*sorted inductive* rule is a pair $(\{\mathrm{P}_s\}_{s \in \mathrm{S}}, \mathrm{c}_{s'})$, where $\mathrm{P}_s \subseteq \mathcal{U}_s$ is a set (with elements of sort s) and $\mathrm{c}_{s'} \in \mathcal{U}_{s'}$ is an object (of sort s'), for $s, s' \in \mathrm{S}$. The definitions of ϕ-closed set and $\mathrm{I}(\phi)$ extend trivially to sets of S-sorted inductive rules. Inductive rules are the special case of S-sorted rules where S is a singleton. Hereafter, we use the term "rules" for "S-sorted inductive rules", for some set S.

Associated with an inductively defined set there is an important proof method that can be used to prove properties about the set $\mathrm{I}(\phi)$. The next proposition states the Principle of Rule Induction, or ϕ-induction in Aczel's terminology.

Proposition 1 (Principle of Rule Induction) If Φ is a total predicate over $\mathrm{I}(\phi)$, such that whenever $(\mathrm{P}, \mathrm{c}) \in \phi$ and $\Phi(x)$ holds for all $x \in \mathrm{P}$ implies $\Phi(\mathrm{c})$ holds, then $\Phi(\mathrm{a})$ holds for every $\mathrm{a} \in \mathrm{I}(\phi)$.

Proof Clearly, $\{\mathrm{a} \in \mathrm{I}(\phi) : \Phi(\mathrm{a}) \text{ holds}\}$ is ϕ-closed, and therefore contains $\mathrm{I}(\phi)$. □

The principle of rule induction extends trivially to an S-sorted inductively defined set.

2.3 Relational Specifications

In this section we define the syntactical aspects of the formalism of Relational Semantics, by defining the syntactical aspects of a Relational Specification. This gives a formal and abstract notation in which the semantics of programming

languages may be expressed. This abstract notation may have a concrete realisation either for machine manipulation, or for making the concrete semantics more readable or easier to write, or both. We introduce such a concrete notation at the end of this section with the objective of making the examples throughout this thesis more readable.

The basic syntactical objects of a Relational Specification are type names (called sorts), terms, formulae, and inductive rules. Those objects are defined using the standard concept of first order signature; the particular definition used in this thesis is that of [GB90]. All algebraic concepts used hereafter are standard definitions in Universal Algebra which can be found for instance in [Coh65, TWW78,Grä79,Wec92].

A (many sorted) *algebraic signature* is a pair (S, F) where S is a set (of *sorts* or *type names*) and F is a $S^* \times S$-sorted family of sets (of *function names*). We denote an algebraic signature (S, F) by Σ. A (many sorted) *first order signature* (or simply a *signature*) is a triple (S, F, Π), where (S, F) is an algebraic signature and Π is a S^+-sorted family of sets (of *relation names*); we denote a first order signature by Ω. The class of all first order signatures is denoted by *Sig*.

As a notational convention, whenever a $\sigma \in F_{s_1 \ldots s_n, s}$, we write $\sigma : s_1 \times \ldots \times s_n \rightarrow s$; if $n = 0$ we write $\sigma : s$. Whenever a relation name $\pi \in \Pi_{s_1 \ldots s_n}$ we write $\pi : s_1 \times \ldots \times s_n$. In either case, we assume that the sorts $s_1 \ldots s_n \in S^*$ and $s \in S$ whenever S is understood in the context.

Every algebraic signature Σ defines a set of well formed terms that can be constructed from free meta-variables and the function names in F (called the Σ-terms). Let X be a S-sorted family of *meta-variables*. For each sort $s \in S$, the set $T_X(\Sigma)_s$ of *terms of sort* s (with meta-variables in X) is the least set (with respect to set inclusion) defined as follows:

1. For all $x \in X_s$, $x \in T_X(\Sigma)_s$
2. For all $\sigma \in F_{s_1 \ldots s_n, s}$, $n \geq 0$:
   if $t_i \in T_X(\Sigma)_{s_i}$, for $i \in [n_+]$, then $\sigma(t_1, \ldots, t_n) \in T_X(\Sigma)_s$

The notation $\mathcal{V}(t)$ denotes the set of meta-variables of a term t. Terms without meta-variables are called *ground terms*; the set of ground terms of sort s is denoted by $T(\Sigma)_s$. A term in some $T(\Sigma)_s$ is usually written $\bar{t}$ to emphasise that it is a ground term. The set $T_X(\Sigma)$ (respectively $T(\Sigma)$) denotes the family $\{T_X(\Sigma)_s\}_{s \in S}$ (respectively $\{T(\Sigma)_s\}_{s \in S}$).

Every first order signature $\Omega = (S, F, \Pi)$ defines a set of well formed atomic formulae that can be constructed from free meta-variables, and Σ-terms. It is convenient, for the definitions below, to impose a restriction that for all $w, w' \in S^+$ such that $w \neq w'$, $\Pi_w \cap \Pi_{w'} = \{\ \}$. This does not restrict the theory; however, its relaxation would make the presentation more complex.

For each relation name $\pi \in \Pi_{s_1 \ldots s_n}$, the set $F_X(\Omega)_\pi$ of *well formed atomic formulae* with relation name π, or π-formulae, is defined by:

$$F_X(\Omega)_\pi = \{(t_1, \ldots, t_n) : t_i \in T_X(\Sigma)_{s_i}\}$$

Hereafter, we use the term *well formed formula*, or simply *formula*, to refer to a well formed atomic formula. Formulae without meta-variables are called *ground formulae*; the set of ground π-formulae is denoted by $F(\Omega)_\pi$. We will use $\pi(t_1, \ldots, t_n)$ as an alternative notation for a π-formula to emphasise that $(t_1, \ldots, t_n)$ in an element in $F_X(\Omega)_\pi$, for some $\pi \in \Pi$; in which case we write $\pi(t_1, \ldots, t_n) \in F_X(\Omega)$ omitting the index π from $F_X(\Omega)_\pi$.

We shall often need to simultaneously substitute all occurrencies of a meta-variable in a term or formula by a term, therefore we need the concept of a (simultaneous) substitution. A *substitution* is an S-sorted family of total functions $\theta = \{\theta_s\}$ such that $\theta_s : X_s \to T_X(\Sigma)_s$. Then, the *instance* of t by θ is obtained using the function $\mathcal{I} : T_X(\Sigma)_s \times (X_s \to T_X(\Sigma)_s) \to T_X(\Sigma)_s$ recursively defined as follows:

1. If $t = x$, for $x \in X_s$, then $\mathcal{I}(t, \theta) = \theta_s(x)$.

2. If $t = \sigma(t_1, \ldots, t_n)$, $\sigma : s_1 \times \ldots \times s_n \to s \in F$,
   then $\mathcal{I}(t, \theta) = \sigma(\mathcal{I}(t_1, \theta), \ldots, \mathcal{I}(t_n, \theta))$.

We use $t\theta$ as an abbreviation for $\mathcal{I}(t,\theta)$. The *composition* of two substitutions θ and θ' is denoted by $\theta\theta'$. A substitution is a *ground instantiation* for $t \in T_X(\Sigma)_s$ if $t\theta \in T(\Sigma)_s$; in which case $t\theta$ is a *ground instance* of t. The above definitions extend trivially to formulae.

The last syntactical object of a Relational Specification is an inductive rule. A *Relational Inductive Rule* (or simply a Relational Rule) is a pair $(\{\mathrm{Pre}_\pi\}_{\pi\in\Pi}, \mathrm{ce}_{\pi'})$ where $\mathrm{Pre}_\pi \in \mathcal{F}(F_X(\Omega)_\pi)$ is a finite set of π-formulae (called the *premisses*) and $\mathrm{ce}_{\pi'} \in F_X(\Omega)_{\pi'}$, for some $\pi' \in \Pi$, is a π'-formula (called the *conclusion*). The definition of substitution and other related concepts extend trivially to Relational Rules.

A set of Relational Rules is an inductive definition, here called a *Relational Inductive Definition*, usually denoted by ϕ. An inductive definition is a finitary way of representing a (possibly infinite) inductive system (defined in Section 2.2). One possible way of deriving the inductive system represented by a Relational Inductive Definition is by generating all possible ground instantiations of each Relational Rule in the definition. The Declarative Semantics of Relational Specifications, defined in Section 2.4, describes an alternative way to perform this derivation using evaluation of terms in a Σ-algebra.

Finally, we define the concept of a Relational Specification Syntax, which characterises the syntactical aspects of a Relational Specification.

Definition 1 (Relational Specification Syntax) A *Relational Specification Syntax* is a pair (Ω, ϕ), where ϕ is a Relational Inductive Definition over the first order signature Ω. □

In the examples throughout this thesis it will often be necessary to name compound objects like signatures and Relational Specification Syntaxes. For this we adopt the following convention: whenever $\mathcal{S}$ is the name given to a Relational Specification Syntax (Ω, ϕ), we write $(\Omega^{\mathcal{S}}, \phi^{\mathcal{S}})$ to make it explicit that $\Omega^{\mathcal{S}}$ and $\phi^{\mathcal{S}}$ are objects of $\mathcal{S}$. In this case we assume that $\Omega^{\mathcal{S}} = (\mathrm{S}^{\mathcal{S}}, \mathrm{F}^{\mathcal{S}}, \Pi^{\mathcal{S}})$ and

$\Sigma^{\mathcal{S}} = (S^{\mathcal{S}}, F^{\mathcal{S}})$. This convention applies to every other compound object defined in later chapters.

The above definitions provide a formal characterisation of the syntactic aspects of the Relational Semantics formalism. With this characterisation it is possible to write Relational Specifications in the abstract notation of terms, formulae and Relational Inductive Definitions. In this sense, the theory of Relational Semantics is independent of any particular concrete syntax used to write Relational Specifications.

Other works, like TYPOL [Des88], and Berry's Language Specification Language [Ber91a], formalise the concept of a semantic specification by a concrete specification language. We prefer the abstract definition because it keeps the theory isolated from idiosyncrasies of the particular concrete language, and gives more freedom in the choice of a concrete language to suit the needs of particular applications of the theory.

In the examples throughout this thesis we use a concrete notation to write Relational Specifications with the objective of making them more concise and readable. The next paragraphs define this concrete notation. None of the theoretical results of the rest of this work depend on this concrete notation; all the theory is developed in terms of the abstract notation developed above. Moreover, it is always possible to unambiguously convert from a Relational Specification written in this concrete notation to one that fits the definition of a Relational Specification Syntax.

The choice of concrete syntax for a Relational Specification Syntax is inspired by the correspondence between context free grammars (CFG) and initial many sorted algebras, as discussed in [GTWW77, page 75]. This correspondence allows a many sorted initial algebra to be unambiguously derived from a CFG. Informally, let G be a CFG with a set of non-terminals S, a set of terminals F disjoint from S, and production rules $P \subseteq S \times (S + F)^*$. Then a suitable transformation on the right hand side of the productions of G defines an $S^* \times S$-sorted set Σ of

function constants. The initial Σ-algebra $T(\Sigma)$ has a carrier for every sort $s \in S$ which is the set of parse trees derived from a non-terminal s in P.

It is important to notice that the CFG is defining the *abstract syntax* of the language rather than its concrete syntax. It is in this sense that this approach to define a Σ-algebra factors out parsing problems so that the CFG may be ambiguous yet the algebra is well defined.

For instance, consider the grammar of the language *Sum* of previous examples, that is given below:

$$exp ::= \mathbf{num}(nat) \mid exp + exp$$
$$nat ::= 0 \mid 1 \mid \cdots$$

The non terminals of the grammar define the set of sorts $S = \{exp, nat\}$, and the right hand side of the productions define the operation constants $\mathbf{num} : nat \rightarrow exp$, $+ : exp \times exp \rightarrow exp$, $0 : \rightarrow nat$, $1 : \rightarrow nat$, and so on.

Since, in the concrete notation, we write a π-formula with the explicit relation name π, e.g., $\pi(t_1, \ldots, t_n)$, it is no longer necessary to consider the set of premisses of a Relational Rule as a Π-sorted family of sets in the concrete notation. This makes it simpler to write the concrete syntax for Relational Rules in the traditional Natural Deduction style of rules, such that a Relational Rule $(\mathrm{Pre}, \mathrm{ce})$, where Pre is the set $\{f_1, \ldots, f_n\}$, is normally written:

$$\frac{f_1 \quad \cdots \quad f_n}{\mathrm{ce}}$$

The apparent ordering on the premisses of the rules, suggested by the format of the above rule, has no meaning in the theory; thus, the set of premisses of any Relational Rule is still considered as just a set of formulae.

To define the set of meta-variables that is used to build terms and formulae in a Relational Specification Syntax we adopt the following convention. Let S be a set of sorts and $s \in S$, we say the component X_s of X is generated by some meta-variable v to mean that X_s contains v and all variations of v using index and superscripts.

The next example shows a complete Relational Specification Syntax written

in the concrete notation described above. In this example, we shall see how relation names are declared.

Example 1 This example defines a complete Relational Specification Syntax, called $Exp = (\Omega^{Exp}, \phi^{Exp})$, for a simple arithmetic expression language with local declarations of variables, also called *Exp*. The objectives of this example are to illustrate the use of the concrete notation described above, and to define some notational conventions that will be used in later examples. The sets S^{Exp} and F^{Exp} are defined by the following BNF rules:

$$\begin{aligned} exp &::= \mathbf{id}(var) \mid \mathbf{num}(nat) \mid exp + exp \mid \mathbf{let}\ var = exp\ \mathbf{in}\ exp \\ var &::= \mathbf{x} \mid \mathbf{y} \mid \cdots \\ nat &::= \mathbf{plus}(nat, nat) \mid 0 \mid 1 \mid \cdots \end{aligned}$$

The set X^E of meta-variables used to construct Σ^{Exp}-terms and Ω^{Exp}-formulae is as defined follows:

X^E_{exp} is generated by e
X^E_{var} is generated by id
X^E_{nat} is generated by n

We will use the set X^E in the examples throughout this thesis and new components will be added to X^E as required.

The rules below define the semantics of *Exp* by an inductive definition of two relations: $\Rightarrow_S$ between expressions and natural numbers, and $\longrightarrow_S$ that describes how to substitute the free occurrences of a variable in an expression by its value. This semantics is called a *substitution-model* semantics of *Exp*. In later examples we shall see how to define *Exp* by an *environment-model* semantics, and also prove that both semantics are equivalent in a sense to be defined.

In the rules below, the relation names in Π^{Exp} are declared in the boxes, such that $\boxed{exp \Rightarrow_S nat}$ declares $\Rightarrow_S \in \Pi^{Exp}_{exp \times nat}$ and emphasises that $\Rightarrow_S$-formulae are written in infix mode. The above BNF rules defined the *abstract syntax* of *Exp*. We should then use this abstract syntax to write terms in the rules. However, we

will use concrete syntax in the examples whenever it does not cause ambiguities to improve readability.

Evaluation of Expressions $\boxed{exp \Rightarrow_S nat}$

$$\frac{}{\mathbf{num}(n) \Rightarrow_S n} \quad (1)$$

$$\frac{e_1 \Rightarrow_S n_1 \qquad e_2 \Rightarrow_S n_2}{e_1 + e_2 \Rightarrow_S \mathbf{plus}(n_1, n_2)} \quad (2)$$

$$\frac{e_1 \Rightarrow_S n_1 \qquad (n_1, id, e_2) \longrightarrow_S e_2' \qquad e_2' \Rightarrow_S n_2}{\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \Rightarrow_S n_2} \quad (3)$$

Substitution $\boxed{(nat, var, exp) \longrightarrow_S exp}$

$$\frac{}{(n, id, \mathbf{num}(n')) \longrightarrow_S \mathbf{num}(n')} \quad (4)$$

$$\frac{}{(n, id, \mathbf{id}(id)) \longrightarrow_S \mathbf{num}(n)} \quad (5)$$

$$\frac{cond(id \neq id')}{(n, id, \mathbf{id}(id')) \longrightarrow_S \mathbf{id}(id')} \quad (6)$$

$$\frac{(n, id, e_1) \longrightarrow_S e_1' \qquad (n, id, e_2) \longrightarrow_S e_2'}{(n, id, e_1 + e_2) \longrightarrow_S e_1' + e_2'} \quad (7)$$

$$\frac{(n, id, e_1) \longrightarrow_S e_1'}{(n, id, \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2) \longrightarrow_S \mathbf{let}\ id = e_1'\ \mathbf{in}\ e_2} \quad (8)$$

$$\frac{cond(id \neq id') \qquad (n, id, e_1) \longrightarrow_S e_1' \qquad (n, id, e_2) \longrightarrow_S e_2'}{(n, id, \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2) \longrightarrow_S \mathbf{let}\ id = e_1'\ \mathbf{in}\ e_2'} \quad (9)$$

Comments

1. The sets of terminal symbols and productions of the CFG used in the above definition are infinite. Infinite grammars are a convenient device to define languages as already pointed out in [GTWW77,vW69].

2. The premisses of the kind $cond(id \neq id')$ are called "side conditions" in most approaches to relational semantics. The treatment of such premisses is discussed in detail below.

Some "Syntactic Sugar"

To simplify the examples we shall adopt the following convention: we omit the unary operators that are only used for sort conversion. An example of such an operator is the operator **num**, that is only used to convert *nat* into *exp*. Therefore, we shall write the definitions of S^{Exp} and F^{Exp} as follows:

$$\begin{array}{lll} exp & ::= & var \mid nat \mid exp + exp \mid \textbf{let } var = exp \textbf{ in } exp \\ var & ::= & \mathbf{x} \mid \mathbf{y} \mid \cdots \\ nat & ::= & \textbf{plus}(nat, nat) \mid 0 \mid 1 \mid \cdots \end{array}$$

and the Relational Rules will be changed accordingly, for instance:

$$\frac{}{n \Rightarrow_S n}$$

$$\frac{}{(n, id, id) \longrightarrow_S n}$$

In the first rule it should be obvious from the definition of X^E_{nat} that n is a meta-variable of sort *nat*. In Section 8.2 we discuss the use of order-sorted signatures and algebras as a formalisation of the above "syntactical sugar". However, we prefer to develop our theory in the simpler framework of many sorted algebras, and leave the use of order-sorted algebras to be studied in future research. This discussion finishes Example 2.1 □

The Syntactical Treatment of Side Conditions

The premiss $cond(id \neq id')$ in Rules 1(6) and 1(9) is usually called a *side-condition*. In most approaches in the literature side-conditions are treated in a

theory outside the semantics formalism. In this section we discuss the syntactical treatment that we give to side-conditions.

The unary relation *cond* has sort *MetaBool*, i.e., *cond* : *MetaBool*. Moreover, the following Relational rule defines this relation:

$$\frac{}{cond(\mathbf{MetaTrue})} \quad (0)$$

where **MetaTrue** is a nullary function name of sort *MetaBool*, i.e. **MetaTrue** : *MetaBool*. Hereafter, we assume that the Relational Rule 0, the constants **MetaTrue** and **MetaFalse**, and the function names of the form $\sigma : w \rightarrow MetaBool$, for some $w \in S^*$, are implicit components of any Relational Specification Syntax.

For instance the following BNF rule is assumed to be part of the semantics for *Exp* given in Example 1:

$$MetaBool ::= var \neq var \mid \mathbf{MetaTrue} \mid \mathbf{MetaFalse}$$

Therefore, we treat side conditions as normal premisses in this work as far as the syntactical aspects are concerned; in the next section we shall explain the meta-semantics of such treatment.

As another "syntactic sugar" we omit the relation name *cond* whenever this does not cause ambiguity. For instance, Rule 1(6) is written:

$$\frac{id \neq id'}{(n, id, id') \longrightarrow_S id'} \quad 1(6)$$

2.4 Declarative Semantics of Relational Specifications

This section presents the concept of the Declarative Semantics of a Relational Specification, that is, we define the meta-semantics of a Relational Specification Syntax. While the definitions of the last section formally characterise the

syntactic aspects of the Relational Semantics formalism, allowing us to write concrete Relational Specifications, the Declarative Semantics of this section defines a precise mathematical meaning for those syntactic objects.

We may informally understand the definitions of this section as follows. Suppose we have written a Relational Specification Syntax (Ω, ϕ), and now want to know which relations it defines. To answer this question we first choose an arbitrary Σ-algebra, where the meaning of the Relational Specification will be given with respect to this algebra. Then, we derive an inductive system from the Relational Inductive Definition ϕ. Intuitively, this derivation is done by constructing all possible ground instantiations of each rule in ϕ, and then evaluating the terms in these ground instantiations according to the chosen Σ-algebra.

A Relational Specification defines exactly the family of relations inductively defined by its *derived inductive system.* Therefore, a Declarative Semantics of a Relational Specification is a first order model that satisfies exactly the tuples that belong to the relations (inductively) defined by the specification. These notions are formalised below.

Throughout this section we fix an arbitrary Relational Specification Syntax (Ω, ϕ), where $\Omega = (\mathrm{S}, \mathrm{F}, \Pi)$, and an S-sorted set X of meta-variables.

The definition of the Declarative Semantics of a Relational Specification require the definition of term evaluation according to a Σ-algebra. Towards the definition of this concept we present the standard concept of a partial Σ-algebra. If $\Sigma = (\mathrm{S}, \mathrm{F})$ is an algebraic signature, a *partial* Σ*-algebra* A consists of an S-sorted family of sets $\{\mathrm{A}_s\}$ such that A_s is the *carrier* of sort s, and a partial function $\sigma^{\mathrm{A}} \in [\mathrm{A}^w \to \mathrm{A}_s]$ for each symbol $\sigma \in \mathrm{F}_{w,s}$, called the *interpretation* of σ in A. The class of all Σ-algebras is denoted by $Alg(\Sigma)$.

We now define evaluation of Σ-terms with meta-variables in $\{\mathrm{A}_s\}$; this is the standard definition of evaluation of terms over partial algebras [Grä79]. The *evaluation* of a Σ-term in A is a S-sorted partial function $\psi = \{\psi_s\}$ such that $\psi_s : T_{\mathrm{A}}(\Sigma)_s \to \mathrm{A}_s$ is defined as follows:

1. if $t = \mathrm{a}$, and $\mathrm{a} \in \mathrm{A}_s$, for some $s \in \mathrm{S}$, then $\psi_s(t) = \mathrm{a}$.

2. if $t = \sigma(t_1, \ldots, t_n)$, $\sigma : s_1 \times \ldots \times s_n \to s$, $t_i \in \mathrm{dom}\ \psi_{s_i}$, for $i \in [n_+]$ and $n \geq 0$, and $(\psi_{s_1}(t_1), \ldots, \psi_{s_n}(t_n)) \in \mathrm{dom}\ \sigma^{\mathrm{A}}$, then $\psi_s(t) = \sigma^{\mathrm{A}}(\psi_{s_1}(t_1), \ldots, \psi_{s_n}(t_n))$.

We write ψ for ψ_s whenever the sort s is understood from the context. The evaluation of Ω-formulae in A is a natural extension of ψ to a Π-sorted function $\Psi = \{\Psi_\pi\}_{\pi \in \Pi}$, such that $\Psi_\pi : F_{\mathrm{A}}(\Omega)_\pi \to \mathrm{A}^w$, for $\pi \in \Pi_w$, $w = s_1 \ldots s_n$, and $n > 0$, is defined as follows:

> If $(t_1, \ldots, t_n) \in F_{\mathrm{A}}(\Omega)_\pi$, $t_i \in T_{\mathrm{A}}(\Sigma)_{s_i}$, for $i \in [n_+]$, and $t_i \in \mathrm{dom}\ \psi_{s_i}$, then $\Psi_\pi((t_1, \ldots, t_n)) = (\psi_{s_1}(t_1), \ldots, \psi_{s_n}(t_n))$.

Hereafter, ψ^{A} (respectively Ψ^{A}) is used to make explicit that ψ (respectively Ψ) is the evaluation function of Σ-terms (respectively Ω-formula) with respect to a Σ-algebra A.

The next step towards the definition of the Declarative Semantics is to use term evaluation in the derivation of an inductive system from a Relational Inductive Definition. The *inductive system derived* from a Relational Inductive Definition ϕ by a Σ-algebra A is the set of Π-sorted rules ϕ^{A} obtained as follows. For all ground instantiation $(\{\overline{\mathrm{Pre}_\pi}\}, \overline{\mathrm{ce}_{\pi'}})$ of each Relational Rule $(\{\mathrm{Pre}_\pi\}, \mathrm{ce}_{\pi'}) \in \phi$, $(\{\mathrm{P}_\pi\}_{\pi \in \Pi}, \mathrm{c}_{\pi'}) \in \phi^{\mathrm{A}}$ is defined by:

$$\mathrm{P}_\pi = \{\Psi_\pi(f) : f \in \overline{\mathrm{Pre}_\pi} \text{ and } f \in \mathrm{dom}\ \Psi_\pi\}$$

$$\mathrm{c}_{\pi'} = \Psi_{\pi'}(\overline{\mathrm{ce}}_{\pi'}), \qquad \text{for } \overline{\mathrm{ce}}_{\pi'} \in \mathrm{dom}\ \Psi_{\pi'}$$

The set of relations inductively defined by a Relational Inductive Definition ϕ, with respect to a Σ-algebra A, is the Π-sorted family of relations $\mathrm{I}(\phi^{\mathrm{A}})$.

We achieve the definition of a Relational Specification by adding a Σ-algebra to a Relational Specification Syntax.

Definition 2 (Relational Specification) A *Relational Specification* is a triple $(\Omega, \phi, \mathrm{A})$, where (Ω, ϕ) is a Relational Specification Syntax, $\Omega = (\mathrm{S}, \mathrm{F}, \Pi)$, and A is a Σ-algebra. □

The definition of the Declarative Semantics of a Relational Specification uses the notion of a first order model for an Ω-signature, or an Ω-model, whose definition is given below. A *first order* Ω*-model* M consists of a Σ-algebra A and a relation $\pi^{\mathrm{M}} \subseteq \mathrm{A}^{w}$ for each $\pi \in \Pi_{w}$; π^{M} is the *interpretation* of π in M. $Mod(\Omega)$ is the class of all Ω-models.

Finally, with the above definition of Ω-models, we have all necessary elements to define the Declarative Semantics of a Relational specification. Intuitively, it is an Ω-model in which the interpretation of the relations is defined by the rules of the Relational Specification.

Definition 3 (Declarative Semantics) The *Declarative Semantics* of a Relational specification $(\Omega, \phi, \mathrm{A})$, where $\Omega = (\mathrm{S}, \mathrm{F}, \Pi)$, is the Ω-model M with Σ-algebra A such that for each $\pi \in \Pi$, $\pi^{\mathrm{M}} = \mathrm{I}(\phi^{\mathrm{A}})_{\pi}$. □

Once we have defined a Relational Semantics of a programming language we are interested to know whether a given formulae is valid in this semantics. Since formulae can be interpreted as a relation between programs and results, an answer to this question tells us whether a program evaluates to some result. The ability to answer such a question is achieved by the definition of the concept of *satisfaction* of formulae by an Ω-model.

Let A be a Σ-algebra, M be an Ω-model with Σ-algebra A, and $f \in F_{\mathrm{A}}(\Omega)_{\pi}$, for some $\pi \in \Pi$. The Ω-model M satisfies f, written $\mathrm{M} \models f$, if and only if $f \in \mathrm{dom}\ \Psi^{\mathrm{A}}_{\pi}$ and $\Psi^{\mathrm{A}}_{\pi}(f) \in \pi^{\mathrm{M}}$. Therefore, $f \in F_{\mathrm{A}}(\Omega)_{\pi}$ is *valid* in $(\Omega, \phi, \mathrm{A})$ if and only if $\mathrm{M} \models f$.

Now we want to extend the concept of satisfaction to formulae with meta-variables over an arbitrary S-sorted set X. Suppose that $f \in F_{\mathrm{X}}(\Omega)_{\pi}$ for some $\pi \in \Pi$. Intuitively, we want to interpret the validity of f in M as the validity of the *existential closure* of f, i.e., the logic formula $\exists x_1, \ldots, x_n . f$, where $x_1, \ldots, x_n$ are all the meta-variables in f. This departs from the standard logical interpretation of satisfaction, in which the *universal closure* of f is taken into account. However, it is in agreement with the logic programming view in which f is considered as a

query for a *satisfying substitution* in the model M. We are interested in the logic programming approach since it characterises program evaluation as we discuss in Chapter 5.

Therefore, we say that M satisfies $f \in F_X(\Omega)_\pi$ if and only if *there exists* a ground instantiation θ such that $M \models f\theta$. In the case when such a θ exists it is called a *satisfying substitution* for f in M. In Chapter 5 we shall define a procedure that finds a satisfying substitution whenever one exists. Based on that algorithm we define program evaluation and evaluation step, which form the basis of the theory of debugger design of Chapters 6 and 7.

The next example defines an algebra for the Relational Specification Syntax defined in Example 1; thus it also defines a Relational Specification for *Exp* together with its Declarative Semantics.

Example 2 Let $(\Omega^{Exp}, \phi^{Exp})$ be the Relational Specification Syntax defined in Example 1. We now define the Σ^{Exp}-algebra A^{Exp} to be used in the Relational specification $(\Omega^{Exp}, \phi^{Exp}, A^{Exp})$. To improve the readability, we use A for A^{Exp} in the rest of this example. The carriers of A are defined as follows:

$$A_{exp} = T(\Sigma^{Exp})_{exp}$$
$$A_{var} = T(\Sigma^{Exp})_{var}$$
$$A_{nat} = \{0, 1, \ldots\}$$

The algebra A interprets **plus** as the standard sum operation on the natural numbers and $\neq$ as the intuitive inequality between identifiers, and for each $op \in \{\mathbf{id}, \mathbf{num}, +, \mathbf{let}, 0, 1, \ldots, x, y, \ldots\}$ the interpretation op^A is defined as a follows. For $op : s_1 \ldots s_n \rightarrow s$ and $t_i \in A_{s_i}$, $i \in [n_+]$, $op^A(t_1, \ldots, t_n) = op(t_1, \ldots, t_n)$. This is the standard *term algebra* interpretation for the function names.

Comments

It is convenient to notice that A is a total algebra; this fact will be used in later examples.

The following examples illustrate the evaluation of terms in A, and show the effect of term evaluation in the derivation of the inductive system from ϕ^{Exp} using A. In these examples, we use the "desugared" version of the syntax of *Exp* to make the examples clearer. In A the evaluation of a term $+(\mathbf{num}(1), \mathbf{num}(2)) \in T(\Sigma^{Exp})_{exp}$ is defined by:

$$\begin{aligned}&\psi^{\mathrm{A}}(+(\mathbf{num}(1), \mathbf{num}(2))) = \\ &+^{\mathrm{A}}(\mathbf{num}^{\mathrm{A}}(1^{\mathrm{A}}), \mathbf{num}^{\mathrm{A}}(2^{\mathrm{A}})) = \\ &+(\mathbf{num}(1), \mathbf{num}(2))\end{aligned}$$

The evaluation of the term $\mathbf{plus}(1, 2) \in T(\Sigma^{Exp})_{nat}$ is defined by:

$$\psi^{\mathrm{A}}(\mathbf{plus}(1, 2)) = \mathbf{plus}^{\mathrm{A}}(1^{\mathrm{A}}, 2^{\mathrm{A}}) = \mathbf{plus}^{\mathrm{A}}(1, 2) = 3$$

In general, algebras used in Relational Specification will have this characteristic: compound values, programs, and other complex operators receive the term algebra interpretation described above; this constitutes the *syntactic algebra* of the specification. On the other hand, basic operators like **plus** are assigned an actual interpretation, which is usually well defined and understood; this constitutes the *semantic algebra* of the specification.

Using A in the derivation of an inductive system from ϕ^{Exp}, one rule of the derived system will have the form:

$$\frac{\mathbf{num}(1) \Rightarrow_S 1 \qquad \mathbf{num}(2) \Rightarrow_S 2}{\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow_S 3}$$

which makes the two formulae:

$$\begin{aligned}&\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow_S 3 \\ &\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow_S \mathbf{plus}(1, 2)\end{aligned}$$

valid in $(\Omega^{Exp}, \phi^{Exp}, \mathrm{A})$. This finishes the Example 2.2. □

The Semantic Treatment of Side Conditions

As discussed in Section 2.3, side-conditions are syntactically treated as normal premisses in Relational Rules. Let us discuss their meta-semantic interpretation.

Initially, let us assume that the interpretation of **MetaTrue** and **MetaFalse** is the term algebra interpretation in any Σ-algebra used in a Relational Specification. Moreover, any $\sigma : w \rightarrow \mathit{MetaBool}$ in F is interpreted in the semantic algebra. For instance, in Example 2 the interpretation of $\neq$ in A^{Exp} is the intuitive inequality on identifiers. Therefore, the inductive rule below is obtained from the rule 1(6) and belongs to the inductive system derived from the inductive definition ϕ^{Exp}:

$$\frac{\mathit{cond}(\mathbf{MetaTrue})}{(1, \mathbf{x}, \mathbf{y}) \longrightarrow_S \mathbf{y}}$$

because in that derivation the term $\mathbf{x} \neq \mathbf{y}$ evaluates to **MetaTrue**. Considering that the rule (0) defined in page 49 is part of *Exp*, the premiss of the above rule is valid in *Exp*, and the conclusion is therefore valid.

Therefore, side-conditions are also treated as normal premisses at the meta-semantic level. The uniformity achieved by treating side conditions as normal premisses makes the definition of program evaluation and evaluation steps simpler; thus, it helps in the definition of an intuitive notion of evaluation step. For the purposes of this thesis the above treatment of side conditions is an advantage over other approaches in which side-conditions have special treatment.

Finite Length Proofs and Proof Trees

We briefly define the standard concepts of finite length proofs and proof trees for an inductive system. Those concepts provide alternative characterisations of inductively defined sets; thus giving an alternative characterisation for the Declarative Semantics of the Relational Specification. This alternative characterisations are important devices for proving properties of inductively defined sets, thus they may also be used to prove properties of Relational Specifications.

The set inductively defined by a finitary inductive system can be alternatively characterised by *finite length proofs* of objects [Acz77]. A *finitary rule* is an inductive rule whose set of premisses is finite. A *finitary inductive system* is a set of finitary rules.

Definition 4 (Finite Length Proof) Given a finitary inductive system ϕ, a sequence $\langle b_0, \ldots, b_n \rangle$ is a finite length proof, or a ϕ-proof, of an object b if:

1. $b_n = b$, and
2. for all $m \leq n$ there is a set $\mathrm{B} \subseteq \{b_i | i < m\}$ such that $(\mathrm{B}, b_m) \in \phi$

□

The next proposition shows that the set defined by an inductive system ϕ is the set of objects that have finite ϕ-proofs in the inductive system. The proof is omitted here and can be found in [Acz77].

Proposition 2 For every finitary set of rules ϕ:

$$\mathrm{I}(\phi) = \{a | a \text{ has a } \phi\text{-proof}\}$$ □

The notion of proof trees is widely known in logic programming [Cla79], and is usually taken as an alternative declarative semantics for definite clause programs as in [DM85]. A particular definition of proof trees is used below as an alternative characterisation of inductively defined sets. Proof trees are usually used to show that an object belongs to an inductively defined set. The proposition below formally justifies this use of proof trees.

Definition 5 (Proof Tree) Let ϕ be a finitary set of rules. A *proof tree* for an object a, called a *ϕ-tree* and denoted $\mathrm{PT}(a)$, is an object:

$$\frac{\mathrm{PT}(b_1) \ldots \mathrm{PT}(b_n)}{b}$$

where: there exists a rule $(\{b_1, \ldots, b_n\}, b) \in \phi$, such that $a = b$ and $\mathrm{PT}(b_i)$ is a ϕ-tree of b_i, for $0 \leq i \leq n$. □

In a proof tree of a in the above definition, b is called the *root* of the tree.

Proposition 3 For every finitary set of rules ϕ:

$$\mathrm{I}(\phi) = \{a | a \text{ has a } \phi\text{-tree}\}$$

Proof (Sketch) It is easier to show first that:

$$\{a | a \text{ has a } \phi\text{-tree}\} = \{a | a \text{ has a } \phi\text{-proof}\}$$

This argument is trivial. For each ϕ-tree for an object a we can obtain a ϕ-proof for a by linearising the tree by some traversal of it. Conversely, for each ϕ-proof of a we can build a ϕ-tree by putting together the rules that were used in the ϕ-proof. This proposition then follows from Proposition 2. □

2.5 Summary and Conclusions

This chapter defined the semantics formalism that will be used throughout this thesis. This definition was carried out in two stages. We first defined the syntactic aspects of the formalism by defining the concept of a Relational Specification Syntax. We then defined the meta-semantic aspects of the formalism by defining the concept of a Relational Specification and its Declarative Semantics.

The most important difference between Relational Semantics as defined in this chapter and other related works is the precise, yet abstract, definition of a Relational Specification. This definition allows formal reasoning to be done over an abstract Relational Specification. This level of reasoning is essential for the later developments in this thesis.

Although other works have a precise definition of a semantic specification, for instance TYPOL [Des88], and Berry's Language Specification Language [Ber91a], these definitions are given by concrete specification languages with the main purpose of machine manipulation of the semantics. This may have an adverse effect on the formalism; thus we favour the abstract approach defined in this chapter. Moreover, based on our abstract approach it is possible to define concrete semantics specification languages with the purpose of machine manipulation.

The class of programming languages that may be described by Relational Specification is large, including non-deterministic languages. The programming languages discussed in texts on Operational Semantics (e.g., [Plo81,Hen90a]) have straightforward definitions as Relational Specifications. Furthermore, the entire Standard ML semantics [HMT89], including the static semantics, fits in the definitions of this chapter.

A Relational Specification has some similarities to the notion of many-sorted definite clause programs (DCP) [DM85]. Moreover, a many-sorted Herbrand Model (see [Llo87] for the one sorted case) extended with term evaluation could be considered as an alternative characterisation of the Declarative Semantics of Relational Specifications. This relationship with logic programming is not entirely surprising since Relational Specifications and DCP's are both used to define relations. It would be interesting to study of this relationship in more detail to see how results from one theory could be applied to the other.

We shall see in Chapter 5 that the operational semantics of DCP is also similar to the Computational Semantics of Relational Specifications. In that chapter we discuss the problems that arise when term evaluation is used in the definition of the Declarative Semantics, and compare the problems to approaches that set out to extend DCP's with external functions.

Chapter 3

Equivalence of Relational Specifications

In Chapter 2 we described how to define the semantic aspects of programming languages by Relational Specifications. Such semantic aspects may have various definitions, given by different Relational Specifications. One reason why different definitions are desirable is that they may define the programming language at different levels of abstraction: a definition may be highly abstract and concise, and therefore suitable for reasoning about the language or it may serve as a reference for the users to learn the language. Another definition may be more concrete and closer to an actual implementation of the language on the machine, and therefore suitable as a guide for the implementors of the language.

Furthermore, different definitions may be given at the same abstraction level to increase our confidence in the definition of the language. For similar reasons, the same semantic aspect of a language may be given in different styles of semantics, for instance in operational and denotational styles. This problem has been addressed in the literature (e.g., [Sto81,BF90]), but it is outside the scope of this thesis.

Whatever the reason we provide different definitions of the same semantic aspects of a programming language, it is necessary for the definitions to be equi-

valent in some sense, so that we may use them interchangeably. For instance, in the case we provide an abstract definition as a reference for the user and a more concrete definition as a guide to the implementor, the equivalence between the definitions guarantees that the user learns the language which is implemented.

In this chapter we define a notion of equivalence between Relational Specifications. We start by defining a relation of Observational Equivalence between first order models in Section 3.3. Then, using this relation, we define an equivalence relation between Relational Specifications in Section 3.4. We also define a proof method that may be used in practical proofs of equivalence, and give a complete example of the use of this proof method in Section 3.5.

3.1 Introduction

In this section we shall discuss some of the problems involved in the definition of a notion of equivalence between Relational Specifications. We also discuss the particular solution we adopt and compare it with other approaches in the literature.

To illustrate the following discussion, let us formalise the semantics of the language *Pair*, described in Section 1.2. The following BNF rules define the sets S^{Pair} and F^{Pair} of the signature Ω^{Pair}:

$$\begin{aligned} pair &::= \mathbf{num}(nat) \mid \mathbf{cons}(pair, pair) \mid \mathbf{first}(pair) \mid \mathbf{second}(pair) \\ nat &::= 0 \mid 1 \mid \cdots \end{aligned}$$

The sets of rules in Figure 1.3 (page 15) define two inductive definitions of the relation $\hookrightarrow$: $pair \times pair$, i.e., $\hookrightarrow \in \Pi^{Pair}_{pair,pair}$. Let $\Sigma^{Pair} = (\mathrm{S}^{Pair}, \mathrm{F}^{Pair})$ and A^{Pair} be the Σ^{Pair}-algebra that gives the term algebra interpretation to all function names in F^{Pair}. Using A^{Pair} we obtain two Relational Specifications of *Pair*:

$$\begin{aligned} Direct &= (\Omega^{Pair}, \mathbf{Direct}, \mathrm{A}^{Pair}) \\ Reverse &= (\Omega^{Pair}, \mathbf{Reverse}, \mathrm{A}^{Pair}) \end{aligned}$$

Although the algebras in the two Relational Specifications are the same, the Declarative Semantics D of *Direct* and R of *Reverse* give different interpretation for the relation $\hookrightarrow$ i.e., $\hookrightarrow^{\mathrm{D}} \neq \hookrightarrow^{\mathrm{R}}$. As outside observers, we can *observe* this difference by testing the satisfaction of (ground) formulae in both specifications. For instance, the following observations reveal that $\hookrightarrow^{\mathrm{D}} \neq \hookrightarrow^{\mathrm{R}}$:

$$\mathrm{D} \models \mathbf{cons}(\mathbf{num}(1), \mathbf{num}(2)) \hookrightarrow \mathbf{cons}(\mathbf{num}(1), \mathbf{num}(2)), \text{ but}$$

R does not satisfy the same formula

In this sense, the satisfaction of formulae offers the *observations* we are allowed to make on a Relational Specification. We then consider two semantics to be equivalent if they have the same *behaviour* with respect to the satisfaction of all possible observations. For the semantics *Direct* and *Reverse*, since the formulae in $F(\Omega^{Pair})$ are the all the observations we can make on the two semantics, this notion of equivalence requires that for all $f \in F(\Omega^{Pair})$, $\mathrm{D} \models f$ if and only if $\mathrm{R} \models f$.

Clearly, the flexibility of this notion of equivalence depends on the definition of the observations that can be made. So far, we are considering the entire set of ground formulae of a Relational Specification as the observations. However, in many practical applications we are not interested in all details of a specification, e.g., internal representation of values in abstract data types, function values in functional languages, and so on. Therefore, it is natural to make such details non observable by using a smaller set of formulae as the observations. The practical effect of this on the number of possible observations is that more specifications will become equivalent.

For example, suppose that we are only interested to observe the subset of $F(\Omega^{Pair})$ of formulae of the form $pr \hookrightarrow \mathbf{num}(n)$ for some *pair* expression pr and *nat* number n. In this case, the following holds

$$\mathrm{D} \models pr \hookrightarrow \mathbf{num}(n) \text{ if and only if } \mathrm{R} \models pr \hookrightarrow \mathbf{num}(n)$$

Therefore, if we restrict the observations we can make on *Direct* and *Reverse* to the set of formulae of the form $p \hookrightarrow \mathbf{num}(n)$, the specifications can be considered

equivalent. In this case, we say that *Direct* and *Reverse* are *observationally equivalent* with respect to the formulae of the form $pr \hookrightarrow \mathbf{num}(n)$.

There are various approaches to the correctness problem in the theory of Algebraic Specification that formalise the above intuitive notion of equivalence. We will discuss two of those approaches: Hoare's notion of *representation relation* and *observational equivalence.* The objective is to compare our solution with existing ones.

The notion of representation of a Σ-algebra by another, due to Hoare [Hoa72], requires the existence of a mapping, called an *abstraction function*, from the carriers of the representation algebra to the carriers of the algebra represented, such that each operation in the algebra is compatible with this mapping. This mapping needs only to be defined for a subset of the carrier set of the representation algebra, which is usually characterised by a predicate called the *representation invariant.*

For *Reverse* to be considered a representation of *Direct* in this sense, it would be necessary the existence of a mapping h from a subset of A_{pair}^{Pair} into A_{pair}^{Pair} such that for all *pair* expressions pr, the relation $\hookrightarrow$ is *compatible* with h in the following sense:

1. If $\exists pr_1$ such that $\mathrm{D} \models pr \hookrightarrow pr_1$
   then $\exists pr_2$ such that $\mathrm{R} \models pr \hookrightarrow pr_2$ and $pr_2 = h(pr_1)$,
2. and conversely.

Clearly, there exists an h such that *Reverse* is a representation of *Direct* under the above notion: just take h to be the identity on *pair* values of the form $\mathbf{num}(n)$ and the recursive reversal of each *pair* value of the form $\mathbf{cons}(pr_1, pr_2)$. Therefore, representation relation would be a suitable notion of correctness between the above Relational Specifications, although it is not an equivalence relation since it is not symmetric.

Behavioural or observational equivalence has also been used in the field of

algebraic specification as a formal criterion for one algebra to be a correct representation of another. However, the idea of regarding behaviour, rather than the representation of data, as the relevant aspect of algebras already appears implicitly in the *simulation method* introduced by Milner [Mil71]. This idea applied to algebras first appears as the concept of the *semantics of an algebra* in [GGM76]. More recently, the notion of *observational equivalence* has been given many formalisations in the literature, for instance [Rei81,ST87,NO88,Sch90].

In particular, the notion of observational equivalence of [ST87] can naturally be applied to Relational Specifications. Under this notion two algebras are equivalent if they satisfy the same set of *observable formulae.* Continuing with the example above, suppose we consider as observable the formulae of the form $pr \hookrightarrow \mathbf{num}(n)$, for all *pair* expressions *pr.* Let us call OBS the set of all such formulae.

Using the notion of observational equivalence of [ST87], the specifications *Direct* and *Reverse* are observationally equivalent with respect to the formulae OBS if for all $f \in$ OBS:

$$\mathrm{D} \models f \text{ if and only if } \mathrm{R} \models f$$

Therefore, *Direct* and *Reverse* are observationally equivalent under this notion, because, as discussed above, they give the same result for formulae of the form $pr \hookrightarrow \mathbf{num}(n)$.

In this work we shall use observational equivalence as the notion of equivalence between Relational Specifications. This choice is motivated by several reasons. Initially, in [Sch87, page 255] Schoett proved that observational equivalence is more general than a representation relation in the sense that there exist algebras that are equivalent under observational equivalence but such that there is no representation relation from the representation algebra to the one represented. In Chapter 4 we shall see an example of practical interest in which two Relational Specifications, which we intuitively expect to be equivalent, are not related under any representation relation but are observationally equivalent.

Therefore, a notion of equivalence more general then representation relations is necessary for certain practical applications.

Furthermore, there exists a proof method for observational equivalence based on the notion of correspondence relations [Sch87] that may also be applied to equivalence of Relational Specifications. This method yields practical proofs of equivalence, as will be demonstrated in Section 3.5.

Finally, the particular notion of observational equivalence of [ST87] may be naturally applied to Relational Specification, yielding a simple theory of equivalence; this is the main reason why this particular approach to observational equivalence is chosen in this thesis.

The theory developed in this chapter is applied in Chapter 4 to the problem of compiler correctness. We demonstrate in Section 4.1 how the use of Observational Equivalence improves and extends previous approaches to compiler correctness. This application provides an important motivation for the definitions and theorems in this chapter.

3.2 Algebraic Preliminaries

In this section we define some concepts related to first order signatures and models. Let $\Omega = (\mathrm{S}, \mathrm{F}, \Pi)$ and $\Omega' = (\mathrm{S}', \mathrm{F}', \Pi')$, *signature morphism* ρ from Ω to Ω' is a triple $(\rho_{\mathrm{S}}, \rho_{\mathrm{F}}, \rho_{\Pi})$ such that for each $s \in \mathrm{S}$, $\sigma \in \mathrm{F}$, and $\pi \in \Pi$, $\rho_{\mathrm{S}}(s) \in \mathrm{S}'$, $\rho_{\mathrm{F}}(\sigma) \in \mathrm{F}'$, and $\rho_{\Pi}(\pi) \in \Pi'$; and, whenever $\sigma : s_1 \times \ldots \times s_n \rightarrow s \in \mathrm{F}$ and $\pi : s_1 \times \ldots \times s_n \in \Pi$ then:

$$\rho_{\mathrm{F}}(\sigma) : \rho_{\mathrm{S}}(s_1) \times \ldots \times \rho_{\mathrm{S}}(s_n) \rightarrow \rho_{\mathrm{S}}(s)$$
$$\rho_{\Pi}(\pi) : \rho_{\mathrm{S}}(s_1) \times \ldots \times \rho_{\mathrm{S}}(s_n)$$

If ρ is a signature morphism from Ω to Ω' we write $\rho : \Omega \rightarrow \Omega'$. If ρ is an inclusion mapping of sets, i.e., each ρ_{S}, ρ_{F}, and ρ_{Π} is an inclusion mapping on S, F, and Π respectively, then Ω is a sub-signature of Ω', written $\Omega \sqsubseteq \Omega'$. If $\Omega \sqsubseteq \Omega'$ and $\Omega \neq \Omega'$, we write $\Omega \sqsubset \Omega'$. Let $\Omega \sqsubseteq \Omega'$, and $\mathrm{M} \in \mathit{Mod}(\Omega')$, the *reduct* of M

by Ω, written M/Ω, is an Ω-model that assigns the same interpretation as M to the symbols of Ω.

Let $\Sigma = (S, F)$, $\Sigma' = (S', F')$, and $A \in Alg(\Sigma')$. The above definitions extend trivially to algebraic signatures. Therefore, $\Sigma \sqsubseteq \Sigma'$ denotes the algebraic signature inclusion, and A/Σ denotes the reduct of A by Σ. Furthermore, if $N \in Mod(\Omega)$ then N/Σ denotes the Σ-algebra part of the model N.

3.3 Observational Equivalence: the Generic Case

In this section we define a notion of observational equivalence between first order models. The particular definition of observational equivalence presented here is that of [ST87] extended to the case of first order models.

Although observational equivalence is a suitable equivalence between models, direct proofs that two models are observationally equivalent may be difficult in practice. Therefore, we also define a notion of *Model Correspondence* between first order models inspired by Schoett's concept of strong correspondence [Sch87], and use Model Correspondence as a proof method for observational equivalence. The main theoretical result of this chapter proves that two models are observationally equivalent if and only if there exists a Model Correspondence relation between them.

Intuitively, two models over a signature are observationally equivalent if it is not possible to distinguish between them by only testing the satisfaction of observable formulae in the models. This notion requires the definition of how observations are made, i.e., a definition of observable sentences. In the framework of first order theories, a natural choice for observations would be logical formulae, formed by atomic formulae and logical connectives. As noted in [ST87], the logic used defines the possible observations, whereas the kind of observations we want to perform dictates the choice of the logic.

In this work, we are interested in observing the semantics of programs accord-

ing to some Relational Specification. As discussed in Section 3.1, for an arbitrary Relational Specification $(\Omega, \phi, \mathrm{A})$ the set of ground formula $F(\Omega)$ defines all the observations we can make. We then suggested that, in practice, we are only interested in a subset of $F(\Omega)$ as our observations. This subset can be characterised by a sub-signature of Ω, called an *observation signature*, which we will denote by Ω_{OBS}. Therefore, the set $F(\Omega_{\mathrm{OBS}})$ defines the observations we can make for a given observation signature.

The above discussion leads to the definition of Observational Equivalence between first order Ω-models with respect to an observation signature. In the rest of this chapter let $\Omega \in \mathit{Sig}$ with algebraic signature Σ, $\Omega_{\mathrm{OBS}} \sqsubseteq \Omega$ be an observation signature of Ω such that $\Omega_{\mathrm{OBS}} = (\mathrm{S_{OBS}}, \mathrm{F_{OBS}}, \Pi_{\mathrm{OBS}})$, and let $\Sigma_{\mathrm{OBS}} \sqsubseteq \Sigma$ denote the algebraic signature $(\mathrm{S_{OBS}}, \mathrm{F_{OBS}})$ of Ω_{OBS}. In order to simplify the presentation we will denote by M and N two models in $\mathit{Mod}(\Omega)$ with Σ-algebras A and B respectively i.e., $\mathrm{M}/\Sigma = \mathrm{A}$ and $\mathrm{N}/\Sigma = \mathrm{B}$.

Definition 1 (Observational Equivalence) The Ω-models M and N are *observationally equivalent* with respect to Ω_{OBS}, written $\mathrm{M} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{N}$, if and only if both requirements are satisfied:

1. For all $s \in \mathrm{S_{OBS}}$ and $t \in T(\Sigma_{\mathrm{OBS}})_s$:

$$t \in \mathrm{dom}\ \psi_s^{\mathrm{A}} \text{ if and only if } t \in \mathrm{dom}\ \psi_s^{\mathrm{B}}$$

2. For all $\pi \in \Pi_{\mathrm{OBS}}$ and $f \in F(\Omega_{\mathrm{OBS}})_\pi$:

$$f \in \mathrm{dom}\ \Psi_\pi^{\mathrm{A}} \text{ if and only if } f \in \mathrm{dom}\ \Psi_\pi^{\mathrm{B}}$$
and if both sides of the above equivalence are true then
$$\Psi_\pi^{\mathrm{A}}(f) \in \pi^{\mathrm{M}} \text{ if and only if } \Psi_\pi^{\mathrm{B}}(f) \in \pi^{\mathrm{N}}$$

□

Two important facts about the relation of Observational Equivalence are stated below.

Fact 1 For any signature Ω and $\Omega_{OBS} \sqsubseteq \Omega$, $\equiv_{\Omega_{OBS}} \in Mod(\Omega) \times Mod(\Omega)$ is an equivalence relation on $Mod(\Omega)$, i.e., $\equiv_{\Omega_{OBS}}$ is transitive, reflexive, and symmetric. □

Fact 2 For any signature Ω, and observation signatures Ω_{OBS}, $\Omega'_{OBS} \sqsubseteq \Omega$, if $\Omega_{OBS} \sqsubseteq \Omega'_{OBS}$ then $\equiv_{\Omega'_{OBS}} \subseteq \equiv_{\Omega_{OBS}}$. □

Among the properties of $\equiv_{\Omega_{OBS}}$ stated above, symmetry requires some comments. In the field of algebraic specification of programs there are various notions of representation between algebras that do not have this property. These include the relations of *behavioural inclusion* [Sch87, page 216] and *simulation* [Nip86]. Those relations express the intuition that a program may be a *partial implementation* of its specification. For instance, a program may be undefined in more arguments than its specification, and yet be considered a suitable implementation.

In practice, these partial implementation notions account for limitations of actual implementations, for instance, finite memory space, finite size of numbers, and so forth. For the case of non-deterministic programs, if a program is related to its specification by a simulation in the sense of [Nip86], the program may be less non-deterministic than its specification. The intuition is that in a real implementation we would have to choose which deterministic behaviour to implement.

However, the problem of equivalence between Relational Specifications of programming languages differs from the problem of correctness in algebraic specification in this aspect. To illustrate this difference let us consider the non-deterministic choice operator + of CCS [Mil89]. The non-deterministic behaviour that + introduces in the language is an essential part of CCS's semantics, and must be preserved across alternative definitions of the language. Therefore, a definition of CCS that makes this operator into a deterministic choice operator should not be considered *correct* since it does not preserve this *essential*

non-determinism.

On the other hand, there is another kind of non-deterministic behaviour in the concrete definitions of some programming languages that we call *non-essential.* For instance, in the semantics of Standard ML [HMT89] memory locations are arbitrarily chosen, introducing non-determinism in the language with respect to the memory that results after the evaluation of an expression. However, this non-determinism does not introduce any new feature to the language for it cannot be exploited by the user in any interesting way. Therefore, a definition of Standard ML semantics that chooses memory locations deterministically should be considered a correct alternative definition.

This problem is solved in our framework by "hiding" the choice of memory location such that the difference between the original definition of the semantics and an alternative definition with deterministic choice cannot be observed. In general, we should make the essential characteristics of the language, like the non-determinism of the + operator of CCS, always observable, while non-essential features like the non-deterministic choice of memory location should not be observable for correctness purposes.

Let us now discuss the problem of giving concrete proofs of Observational Equivalence between two models. These proofs are essential in establishing the equivalence of Relational Specifications as shall be seen in the next section. However, proofs of observation equivalence can be difficult to be established. To understand this difficulty, suppose we try a proof by structural induction on the terms in the observable sentences. In general, in such a proof we will have to reason about non-observable sentences in order to apply the inductive hypothesis. However, the original theorem does not mention non-observable sentences, making it impossible to apply the inductive hypothesis directly.

In [Hen90b] Context Induction is proposed as a proof method for behavioural abstractions. A possible drawback of this proof method is that it requires one to identify all contexts in which a visible object may occur, which may involve

some non-trivial reasoning. Furthermore, proofs of context induction are, in general, large even for simple examples. However, a more detailed study is still required to assess the applicability of context induction in proofs of Observational Equivalence as defined in this thesis.

In [Sch87,Sch90], Schoett defines a notion of correspondence relation for many sorted partial algebras, which is a practical proof method to show that two algebras are observationally equivalent. In the rest of this section we will use the concept of correspondence to define a proof method for Observational Equivalence that we call *Model Correspondence.*

We first adapt Schoett's notion of *strong correspondence* [Sch87, page 240] to the case in which observation signatures are used instead of a set of visible sorts. Observation signatures are more natural to be used in our framework than a set of visible sorts, since observations in our approach are the satisfaction of formulae rather than the evaluation of terms to yield a visible result, as in [Sch87].

Recall from page 67 that, for some $\Omega \in \mathit{Sig}$ with algebraic signature Σ, $\Sigma_{OBS} \sqsubseteq \Sigma$ and $A, B \in \mathit{Alg}(\Sigma)$.

Definition 2 (Strong Correspondence)
A *strong* Σ_{OBS}*-correspondence*[1] between A/Σ_{OBS} and B/Σ_{OBS} is an S_{OBS}-sorted relation $G = \{G_s\}_{s \in S_{OBS}}$, where $G_s \subseteq A_s \times B_s$ for all $s \in S_{OBS}$, such that all $\sigma \in F_{OBS}$ are *compatible* with G in the following sense:

For all $\sigma : s_1 \times \ldots \times s_m \to s \in F_{OBS}$, $m \geq 0$:

whenever $(a_i, b_i) \in G_{s_i}$, for $i \in [m_+]$,

then $(a_1, \ldots, a_m) \in \mathrm{dom}\ \sigma^A$ if and only if $(b_1, \ldots, b_m) \in \mathrm{dom}\ \sigma^B$
and if both sides of the above equivalence are true then
$(\sigma^A(a_1, \ldots, a_m), \sigma^B(b_1, \ldots, b_m)) \in G_s$

The fact that G is a strong Σ_{OBS}-correspondence between A/Σ_{OBS} and B/Σ_{OBS} is denoted by $G : A/\Sigma_{OBS} \Longrightarrow\!\!\!\!\times_{\Sigma_{OBS}} B/\Sigma_{OBS}$. □

[1] We use the term *strong correspondence* to keep the consistency with the terminology used in [Sch87], even though we do not present a notion of (weak) correspondence in this thesis.

The above definition says that a relation is a strong correspondence between two algebras A/Σ_{OBS} and B/Σ_{OBS} when every observable function yields a result for some argument in A/Σ_{OBS} if and only if it yields a related result for every related argument in B/Σ_{OBS}.

The following proposition states that every strong correspondence contains all the pairs of values that result from the evaluation, in each algebra, of every observable term. The proposition is important for the proof of Theorem 1 and also for the proofs of equivalence throughout this thesis.

Proposition 1 (Strong Correspondence)
If G is a strong Σ_{OBS}-correspondence between A/Σ_{OBS} and B/Σ_{OBS} then for all $s \in S_{OBS}$ and $t \in T(\Sigma_{OBS})_s$:

$$t \in \text{dom } \psi_s^A \text{ if and only if } t \in \text{dom } \psi_s^B \tag{1}$$

and if both sides of the above equivalence are true then $(\psi_s^A(t), \psi_s^B(t)) \in G_s$

Proof The proof is by structural induction on t.

Base Case $t = \sigma$, for $\sigma : s \in F_{OBS}$. In this case, $(\psi_s^A(\sigma), \psi_s^B(\sigma)) = (\sigma^A, \sigma^B)$ and because G is a Σ_{OBS}-correspondence between A/Σ_{OBS} and B/Σ_{OBS}, then $(\sigma^A, \sigma^B) \in G_s$.

Inductive Step $t = \sigma(t_1, \ldots, t_n)$, for $\sigma : s_1 \times \ldots \times s_n \rightarrow s \in \Sigma_{OBS}$ and $n > 0$. From the definition of evaluation we have:

$$\text{If } t_i \in \text{dom } \psi_{s_i}^A, \text{ and } (\psi_{s_1}^A(t_1), \ldots, \psi_{s_n}^A(t_n)) \in \text{dom } \sigma^A$$
$$\text{then } \psi_s^A(t) = \sigma^A(\psi_{s_1}^A(t_1), \ldots, \psi_{s_n}^A(t_n))$$

$$\text{If } t_i \in \text{dom } \psi_{s_i}^B, \text{ and } (\psi_{s_1}^B(t_1), \ldots, \psi_{s_n}^B(t_n)) \in \text{dom } \sigma^B$$
$$\text{then } \psi_s^B(t) = \sigma^B(\psi_{s_1}^B(t_1), \ldots, \psi_{s_n}^B(t_n))$$

From the inductive hypothesis,

$$t_i \in \text{dom } \psi_{s_i}^A \text{ if and only if } t_i \in \text{dom } \psi_{s_i}^B, \tag{2}$$

and if both sides of the above equivalence are true then $(\psi_{s_i}^A(t_i), \psi_{s_i}^B(t_i)) \in G_{s_i}$

Now we have two cases to analyse:

1. If both sides of the equivalence (2) are false for some $i \in [n_+]$, then both sides of the equivalence (1) are false, and the proposition holds.

2. If both sides of the equivalence (2) are true for all $i \in [n_+]$, then $(\psi^{\mathrm{A}}_{s_i}(t_i), \psi^{\mathrm{B}}_{s_i}(t_i)) \in \mathrm{G}_{s_i}$. Since G is a strong Σ_{OBS}-correspondence then:

   $(\psi^{\mathrm{A}}_{s_1}(t_1), \ldots, \psi^{\mathrm{A}}_{s_n}(t_n)) \in \mathrm{dom}\,\sigma^{\mathrm{A}}$ if and only if $(\psi^{\mathrm{B}}_{s_1}(t_1), \ldots, \psi^{\mathrm{B}}_{s_n}(t_n)) \in \mathrm{dom}\,\sigma^{\mathrm{B}}$

   and when both sides of this equivalence are true then $(\psi^{\mathrm{A}}_{s}(t), \psi^{\mathrm{B}}_{s}(t)) \in \mathrm{G}_s$.

This finishes the proof of Proposition 1. □

The following definition extends the notion of strong correspondence to first order models by defining a pair of relations between models, which we call a Model Correspondence. Recall from page 3.3 that, for some $\Omega \in \mathit{Sig}$, $\Omega_{\mathrm{OBS}} \sqsubseteq \Omega$, and $\mathrm{M}, \mathrm{N} \in \mathit{Mod}(\Omega)$ are models with algebra A and B respectively.

Definition 3 (Model Correspondence)
A Ω_{OBS}-Model Correspondence between M and N is a pair of relations (G, H) such that:

1. G is a strong Σ_{OBS}-correspondence between $\mathrm{A}/\Sigma_{\mathrm{OBS}}$ and $\mathrm{B}/\Sigma_{\mathrm{OBS}}$.

2. $\mathrm{H} = \{\mathrm{H}_\pi\}$ is a Π_{OBS}-sorted family of relations such that for all relation names $\pi : s_1 \times \ldots \times s_n \in \Pi_{\mathrm{OBS}}$, $\mathrm{H}_\pi \subseteq \mathrm{A}_w \times \mathrm{B}_w$, where $w = s_1 \ldots s_n$, with the following restrictions, for $n > 0$ and for all $i \in [n_+]$:

   i. Whenever $((\mathrm{a}_1, \ldots, \mathrm{a}_n), (\mathrm{b}_1, \ldots, \mathrm{b}_n)) \in \mathrm{H}_\pi$ then $(\mathrm{a}_i, \mathrm{b}_i) \in \mathrm{G}_{s_i}$.

   ii. For all $t_i \in T(\Sigma_{\mathrm{OBS}})_{s_i}$, if $t_i \in \mathrm{dom}\ \psi^{\mathrm{A}}_{s_i}$ and $t_i \in \mathrm{dom}\ \psi^{\mathrm{B}}_{s_i}$ then

   $$((\psi^{\mathrm{A}}_{s_1}(t_1), \ldots, \psi^{\mathrm{A}}_{s_n}(t_n)), (\psi^{\mathrm{B}}_{s_1}(t_1), \ldots, \psi^{\mathrm{B}}_{s_n}(t_n))) \in \mathrm{H}_\pi$$

and π is compatible with H_π in the following sense:

$$\text{whenever} \quad ((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \in H_\pi$$
$$\text{then} \quad (a_1, \ldots, a_n) \in \pi^M \text{ if and only if } (b_1, \ldots, b_n) \in \pi^N$$

The fact that (G, H) is a Ω_{OBS}-Model Correspondence between M and N is denoted by $(G, H) : M \longleftrightarrow_{\Omega_{OBS}} N$ □

A Model correspondence is a natural extension of strong correspondence to first order models. Recall that a strong correspondence G is a relation between the carriers of observable sorts in a Σ-algebra, with the requirement that each observable function σ is *compatible* with G. The component H of a Model Correspondence extends the rôle of G to the relations of a first order model. Therefore, H is a relation between the relations in a model with the requirement that each observable π must be compatible with H.

Definition 3(2.i) says that H is consistent with G in the sense that every related tuple in H is built of related values in G. Definition 3(2.ii) requires that every pair of tuples built from the evaluation of the same observable terms must be in H. This requirement guarantees H is not empty whenever $T(\Sigma_{OBS})$, and consequently $F(\Omega_{OBS})$, are not empty.

The following theorem shows that the notion of Model Correspondence is an alternative characterisation of Observational Equivalence. Therefore, the former can be used as a proof method for Observational Equivalence between first order models. The example of Section 3.5 shows that this method yields practical proofs of equivalence between Relational Specifications.

Theorem 1 (Model Correspondence) The models M and N are observationally equivalent with respect to Ω_{OBS} if and only if there exists a Ω_{OBS}-Model Correspondence between M and N, i.e.:

$$M \equiv_{\Omega_{OBS}} N \text{ if and only if there exists } (G, H),\ (G, H) : M \longleftrightarrow_{\Omega_{OBS}} N$$

Proof This is based on the proof of Theorem 4.2.3 of [Sch87, pages 223–225], and is given in two parts as follows.

Soundness We need to prove that if there exists a pair of relations (G, H) such that $(G, H) : M \longleftrightarrow_{\Omega_{OBS}} N$ then $M \equiv_{\Omega_{OBS}} N$. Since G is a strong Σ_{OBS}-correspondence between A/Σ_{OBS} and B/Σ_{OBS} then Proposition 1 apllies. Therefore, Definition 1(1) is trivially satisfied. It remains to prove that Definition 1(2) is satisfied. From Proposition 1 it follows that for all $\pi : s_1 \times \ldots s_n \in \Pi_{OBS}$ and $\pi(t_1, \ldots, t_n) \in F(\Omega_{OBS})_\pi$,

$$t_i \in \text{dom } \psi^A_{s_i} \qquad \text{if and only if} \qquad t_i \in \text{dom } \psi^B_{s_i}$$

Therefore,

$$\pi(t_1, \ldots, t_n) \in \text{dom } \Psi^A_\pi \text{ if and only if } \pi(t_1, \ldots, t_n) \in \text{dom } \Psi^B_\pi$$

Whenever both sides of the above equivalence are true it follows that for all $i \in [n_+]$, $t_i \in \text{dom } \psi^A_{s_i}$ and $t_i \in \text{dom } \psi^B_{s_i}$. From the fact that H satisfies the requirement of Definition 3(2.ii) we obtain that:

$$((\psi^A_{s_1}(t_1), \ldots, \psi^A_{s_n}(t_n)), (\psi^B_{s_1}(t_1), \ldots, \psi^B_{s_n}(t_n)) \in H_\pi$$

Since for all $\pi \in \Pi_{OBS}$, π is compatible with H_π then:

$$(\psi^A_{s_1}(t_1), \ldots, \psi^A_{s_n}(t_n)) \in \pi^M \text{ if and only if } (\psi^B_{s_1}(t_1), \ldots, \psi^B_{s_n}(t_n)) \in \pi^N$$

Therefore, from the definition of formula evaluation (Section 2.4, page 51):

$$\Psi^A_\pi(\pi(t_1, \ldots, t_n)) \in \pi^M \text{ if and only if } \Psi^B_\pi(\pi(t_1, \ldots, t_n)) \in \pi^N$$

Therefore, Definition 1(2) is satisfied and we conclude that $M \equiv_{\Omega_{OBS}} N$. Since this proof is done for an arbitrary pair (G, H), it implies that the existence of any Ω_{OBS}-Model Correspondence between M and N implies $M \equiv_{\Omega_{OBS}} N$.

Completeness We need to prove that $M \equiv_{\Omega_{OBS}} N$ implies that there exists a pair of relations (G, H), such that $(G, H) : M \longleftrightarrow_{\Omega_{OBS}} N$. For this proof we will

show first that G defined as $G_s = (\psi^A_s)^{-1} \circ \psi^B_s$, for each $s \in S_{OBS}$ is a strong Σ_{OBS}-correspondence between A/Σ_{OBS} and B/Σ_{OBS}.

Let $(a_i, b_i) \in G_i$ for $n \geq 0$ and all $i \in [n_+]$. Then, choose $t_i \in \text{dom } \psi^A_{s_i} \cap \text{dom } \psi^B_{s_i}$ in $T(\Sigma_{OBS})_{s_i}$, such that $\psi^A_{s_i}(t_i) = a_i$, $\psi^B_{s_i}(t_i) = b_i$, and $t = \sigma(t_1, \ldots, t_n)$ belongs to $T(\Sigma_{OBS})_s$ for some $s \in S_{OBS}$. From Definition 1(1) we have that:

$$t \in \text{dom } \psi^A_s \text{ if and only if } t \in \text{dom } \psi^B_s$$

and if both sides of this equivalence are true then we can write:

$$(\sigma^A(\psi^A_{s_1}(t_1), \ldots, \psi^A_{s_n}(t_n)), \sigma^B(\psi^B_{s_1}(t_1), \ldots, \psi^B_{s_n}(t_n))) =$$
$$(\psi^A_s(t), \psi^B_s(t)) \in (\psi^A_s)^{-1} \circ \psi^B_s = G_s$$

Which proves G is a strong Σ_{OBS}-correspondence between A/Σ_{OBS} and B/Σ_{OBS}.

We will show that H defined as $H_\pi = (\Psi^A_\pi)^{-1} \circ \Psi^B_\pi$, for all $\pi \in \Pi_{OBS}$, satisfies the requirements of Definition 3(2). From this definition of H_π it follows that for all $\pi(t_1, \ldots, t_n) \in F(\Omega_{OBS})_\pi$, if $t_i \in \text{dom } \psi^A_{s_i}$ and $t_i \in \text{dom } \psi^B_{s_i}$ for every $i \in [n_+]$, then:

$$((\psi^A_{s_1}(t_1), \ldots, \psi^A_{s_n}(t_n)), (\psi^B_{s_1}(t_1), \ldots, \psi^B_{s_n}(t_n))) \in H_\pi$$

The requirements of Definition 3(2.i) follows trivially from the definition of H_π and Proposition 1. Definition 3(2.ii) is trivially satisfies by the definition of H_π. Now, we must show that each $\pi \in \Pi_{OBS}$ is compatible with H_π. For all $\pi : s_1 \times \ldots \times s_n \in \Pi_{OBS}$, whenever $((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \in H_\pi$, for all $i \in [n_+]$, $s_i \in S_{OBS}$ take $t_i \in \text{dom } \psi^A_{s_i} \cap \text{dom } \psi^B_{s_i}$ in $T(\Sigma_{OBS})_{s_i}$ such that $\psi^A_{s_i}(t_i) = a_i$ and $\psi^B_{s_i}(t_i) = b_i$. It is clear that $\pi(t_1, \ldots, t_n) \in F(\Omega_{OBS})_\pi$. Now, from the fact that $M \equiv_{\Omega_{OBS}} N$, it follows that:

$$\pi(t_1, \ldots, t_n) \in \text{dom } \Psi^A_\pi \text{ if and only if } \pi(t_1, \ldots, t_n) \in \text{dom } \Psi^B_\pi$$
and if both sides of the above equivalence are true then
$$\Psi^A_\pi(\pi(t_1, \ldots, t_n)) \in \pi^M \text{ if and only if } \Psi^B_\pi(\pi(t_1, \ldots, t_n)) \in \pi^N$$

It is obvious that $\pi(t_1, \ldots, t_n) \in \text{dom } \Psi^A_\pi$ and $\pi(t_1, \ldots, t_n) \in \text{dom } \Psi^B_\pi$. Therefore, from the definition of evaluation of formulae (Section 2.4, page 51):

$$(\psi^A_{s_1}(t_1), \ldots, \psi^A_{s_n}(t_n)) \in \pi^M \text{ if and only if } (\psi^B_{s_1}(t_1), \ldots, \psi^B_{s_n}(t_n)) \in \pi^N$$

Finally, we conclude that:

$$(a_1, \ldots, a_n) \in \pi^M \text{ if and only if } (b_1, \ldots, b_n) \in \pi^N$$

This proves that π is compatible with H_π, for all $\pi \in \Pi_{OBS}$, and therefore that (G, H) is a Ω_{OBS}-Model Correspondence between M and N. This finishes the proof of Theorem 1. □

Model Correspondences offers a suitable proof method for Observational Equivalence between first order models in a similar way as strong correspondences provide a proof method for observational equivalence between algebras. As discussed above, proofs of Observational Equivalence can be difficult because the requirements of Definition 1 do not involve non observable objects. However, these objects often occur in intermediate steps of inductive proofs making it impossible to apply the inductive hypothesis directly. Model Correspondences solve this problem since Definition 3(2.1) allows non observable objects to be related in H. In Section 3.5, we illustrate the use of Model Correspondences.

3.4 Observational Equivalence of Relational Specifications

In this section we apply the results of Section 3.3 to the equivalence of Relational Specifications. The objective is the definition of a relation of Observational Equivalence between Relational Specifications. For this definition we have a choice between a relation parametric on an observation signature, like the $\equiv_{\Omega_{OBS}}$ relation, or to include an observation signature as a component of the Relational Specification and then define the equivalence relation on the class of Relational Specifications with the same observation signature.

Both choices lead to essentially the same notion of equivalence. The first choice is more flexible in the sense that it directly allows two Relational Specifications to be compared using different observation signatures; the second choice

provides simpler and more concise definitions. Furthermore, we can indirectly achieve the flexibility of the first approach in the second approach by considering different classes of specifications, one for each observation signature. Therefore, we choose the second approach for its simplicity.

We extend a Relational Specification (Definition 2.2) with an observation signature such that whenever $(\Omega, \phi, \mathrm{A})$ is a Relational Specification and $\Omega_{\mathrm{OBS}} \sqsubseteq \Omega$ is an observation signature of Ω, we write the Relational Specification as $(\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$. The class of Relational Specifications with observation signature Ω_{OBS} is denoted by $Spec(\Omega_{\mathrm{OBS}})$. Notice that in general the specifications in $Spec(\Omega_{\mathrm{OBS}})$ have different signatures. The notion of Observational Equivalence between Relational Specifications will account for this.

We now define a relation of Observational Equivalence on $Spec(\Omega_{\mathrm{OBS}})$. To motivate this definition let us consider two Relational Specifications of a simple functional language without recursion, which we call *Fun*. The following example shows a *substitution-model* semantics for *Fun*, which is just an extension to the Relational Specification *Exp* defined in examples 2.1 and 2.2.

Example 1 This example presents a complete definition of a Relational Specification $Sub = (\Omega^{Sub}, \phi^{Sub}, \mathrm{A}^{Sub}, \Omega^{SFun}_{\mathrm{OBS}})$ that defines a semantic of *Fun*. Here, we have called the observation signature $\Omega^{SFun}_{\mathrm{OBS}}$ instead of $\Omega^{Sub}_{\mathrm{OBS}}$ because it will be used in the Relational Specification of the next example as well. The components S^{Sub} and F^{Sub} of Ω^{Sub} are defined by the following BNF rules:

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{var} \mid \mathit{nat} \mid \mathit{exp} + \mathit{exp} \mid \textbf{let } \mathit{var} = \mathit{exp} \textbf{ in } \mathit{exp} \\
 & & \textbf{fn } \mathit{var} \,.\, \mathit{exp} \mid \mathit{exp}(\mathit{exp}) \\
\mathit{var} & ::= & \mathbf{x} \mid \mathbf{y} \mid \cdots \\
\mathit{nat} & ::= & \textbf{plus}(\mathit{nat}, \mathit{nat}) \mid 0 \mid 1 \mid \cdots \\
\mathit{funval} & ::= & \textbf{fn } \mathit{var} \,.\, \mathit{exp} \\
\mathit{val} & ::= & \mathit{nat} \mid \mathit{funval}
\end{array}
$$

where an object of sort *funval* is *function value*, i.e., the value that results from the evaluation of an expression of the form **fn** *var* . *exp*. The meta-variables

used to construct terms and formulae are those in X^E, defined in Example 2.1, to which we add the component X^E_{val} generated by v and n. The set of rules ϕ^{Sub} defines three relations. The relations $\Rightarrow_S$ and $\longrightarrow_S$ have the same intuitive meaning as in Example 2.1, extended to the new language constructs.

The definition of the relation $\Rightarrow$ makes sure that only expressions without free-variables, i.e., *closed expressions*, are evaluated using the relation $\Rightarrow_S$. The evaluation of expressions with free variables may produce incorrect results due to the capture of the free variables in the substitution. This restriction is achieved by using the function FV that, when applied to an expression, returns the set of its free variables. The definition of FV can be given by a simple set of Relational Rules. However, its meaning is standard and we therefore omit its definition from this presentation.

Rule (1) defines the relation $\Rightarrow$ using the relation $\Rightarrow_S$. Rules (2)-(4) and (7)-(12) were defined in Example 2.1 and are also presented here to make this example self-contained.

Evaluation of Expressions $\boxed{exp \Rightarrow val}$

$$\frac{\mathrm{FV}(e) = \{\,\} \qquad e \Rightarrow_S v}{e \Rightarrow v} \tag{1}$$

Evaluation of Closed Expressions $\boxed{exp \Rightarrow_S val}$

$$\frac{}{n \Rightarrow_S n} \tag{2}$$

$$\frac{e_1 \Rightarrow_S n_1 \qquad e_2 \Rightarrow_S n_2}{e_1 + e_2 \Rightarrow_S \mathbf{plus}(n_1, n_2)} \tag{3}$$

$$\frac{e_1 \Rightarrow_S v_1 \qquad (v_1, id, e_2) \longrightarrow_S e_2' \qquad e_2' \Rightarrow_S v_2}{\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \Rightarrow_S v_2} \tag{4}$$

$$\frac{}{\mathbf{fn}\ id\ .\ e \Rightarrow_S \mathbf{fn}\ id\ .\ e} \tag{5}$$

$$\frac{e_1 \Rightarrow_S \mathbf{fn}\ id\ .\ e \qquad e_2 \Rightarrow_S v_2 \qquad (v_2, id, e) \longrightarrow_S e' \qquad e' \Rightarrow_S v}{e_1(e_2) \Rightarrow_S v} \tag{6}$$

Substitution $\boxed{(val, var, exp) \longrightarrow_S exp}$

$$\frac{}{(v, id, n) \longrightarrow_S n} \tag{7}$$

$$\frac{}{(v, id, id) \longrightarrow_S v} \tag{8}$$

$$\frac{id \neq id'}{(v, id, id') \longrightarrow_S id'} \tag{9}$$

$$\frac{(v, id, e_1) \longrightarrow_S e_1' \qquad (v, id, e_2) \longrightarrow_S e_2'}{(v, id, e_1 + e_2) \longrightarrow_S e_1' + e_2'} \tag{10}$$

$$\frac{(v, id, e_1) \longrightarrow_S e_1'}{(v, id, \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2) \longrightarrow_S \mathbf{let}\ id = e_1'\ \mathbf{in}\ e_2} \tag{11}$$

$$\frac{id \neq id' \qquad (v, id, e_1) \longrightarrow_S e_1' \qquad (v, id, e_2) \longrightarrow_S e_2'}{(v, id, \mathbf{let}\ id' = e_1\ \mathbf{in}\ e_2) \longrightarrow_S \mathbf{let}\ id' = e_1'\ \mathbf{in}\ e_2'} \tag{12}$$

$$\frac{}{(v, id, \mathbf{fn}\ id\ .\ e) \longrightarrow_S \mathbf{fn}\ id\ .\ e} \tag{13}$$

$$\frac{id \neq id' \qquad (v, id, e) \longrightarrow_S e'}{(v, id, \mathbf{fn}\ id'\ .\ e) \longrightarrow_S \mathbf{fn}\ id'\ .\ e'} \tag{14}$$

$$\frac{(v, id, e_1) \longrightarrow_S e_1' \qquad (v, id, e_2) \longrightarrow_S e_2'}{(v, id, e_1(e_2)) \longrightarrow_S e_1'(e_2')} \tag{15}$$

The Σ^{Sub}-algebra A^{Sub} is a simple extension to A^{Exp} of Example 2.2 in which $\mathrm{A}^{Sub}_{funval} = T(\Sigma^{Sub})_{funval}$, $\mathrm{A}^{Sub}_{val} = \mathrm{A}^{Sub}_{nat} + \mathrm{A}^{Sub}_{funval}$, and the new language constructs and new values of sort *funval* are assigned the term algebra interpretation.

Finally, the observation signature $\Omega^{SFun}_{\mathrm{OBS}}$ is defined as follows. The set $\Pi^{SFun}_{\mathrm{OBS}}$ has only the relation $\Rightarrow$, and the sets $\mathrm{S}^{SFun}_{\mathrm{OBS}}$ and $\mathrm{F}^{SFun}_{\mathrm{OBS}}$ are defined by the following BNF rules:

$$
\begin{array}{lll}
\mathit{exp} & ::= & \mathit{var} \mid \mathit{nat} \mid \mathit{exp} + \mathit{exp} \mid \textbf{let } \mathit{var} = \mathit{exp} \textbf{ in } \mathit{exp} \\
 & & \textbf{fn } \mathit{var} \,.\, \mathit{exp} \mid \mathit{exp}(\mathit{exp}) \\
\mathit{var} & ::= & \mathbf{x} \mid \mathbf{y} \mid \cdots \\
\mathit{nat} & ::= & 0 \mid 1 \mid \cdots \\
\mathit{funval} & ::= & \\
\mathit{val} & ::= & \mathit{nat}
\end{array}
$$

This finishes the complete definition of *Sub* and Example 3.1. □

We now define another Relational Specification for *Fun*. This definition uses an environment-model semantics in which the variables of expression and their values are kept in an environment, and looked up when needed. For the purpose of establishing their equivalence, an important difference between this semantics and *Sub* is in the representation of function values.

Example 2 In this example we present an *environment-model* semantics for *Fun* by defining a Relational Specification $SEnv = (\Omega^{SEnv}, \phi^{SEnv}, \mathrm{A}^{SEnv}, \Omega^{SFun}_{\mathrm{OBS}})$. In this specification, the values of the free variables of an expression are kept in an environment. Moreover, function values are represented as triples of the form (E, id, e), where E is an environment, id is the formal parameter, and e is the body of the function. This representation for function values is normally called a *closure* in the literature (e.g., in [FH88, page 200]).

The following BNF rules define the components S^{SEnv} and F^{SEnv} of Ω^{SEnv}.

$$
\begin{array}{lll}
\mathit{funval} & ::= & (\mathit{env}, \mathit{var}, \mathit{exp}) \\
\mathit{val} & ::= & \mathit{nat} \mid \mathit{funval} \\
\mathit{env} & ::= & \varepsilon_E \mid \mathit{var} \mapsto \mathit{val} \cdot \mathit{env}
\end{array}
$$

where *exp*, *var*, *nat* were defined in Example 1. We first extend X^E with X^E_{env} generated by E. The Relational Rules below define ϕ^{SEnv}. Rule (1) defines the relation $\Rightarrow$ in terms of the relation $\_ \vdash \_ \Rightarrow \_$. Since in *Sub* only the evaluation of closed expressions is defined, in *SEnv* the evaluation of expressions starts with

an empty environment ε_E, as defined by rule (1). Rules (2) to (7) define the evaluation of expressions in an arbitrary environment E. Rules (8) and (9) define the result of looking up the value of a variable in the environment.

Evaluation of Expressions $\boxed{exp \Rightarrow val}$

$$\frac{\varepsilon_E \vdash e \Rightarrow v}{e \Rightarrow v} \tag{1}$$

Evaluation on an Environment $\boxed{env \vdash exp \Rightarrow val}$

$$\frac{}{E \vdash n \Rightarrow n} \tag{2}$$

$$\frac{(E, id) \longrightarrow_L v}{E \vdash id \Rightarrow v} \tag{3}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad E \vdash e_2 \Rightarrow v_2}{E \vdash e_1 + e_2 \Rightarrow \mathbf{plus}(v_1, v_2)} \tag{4}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad id \mapsto v_1 \cdot E \vdash e_2 \Rightarrow v_2}{E \vdash \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \Rightarrow v_2} \tag{5}$$

$$\frac{}{E \vdash \mathbf{fn}\ id\ .\ e \Rightarrow (E, id, e)} \tag{6}$$

$$\frac{E \vdash e_1 \Rightarrow (E', id, e') \qquad E \vdash e_2 \Rightarrow v_2 \qquad id \mapsto v_2 \cdot E' \vdash e' \Rightarrow v'}{E \vdash e_1(e_2) \Rightarrow v'} \tag{7}$$

Variable Lookup $\boxed{(env, var) \longrightarrow_L val}$

$$\frac{}{(id \mapsto v \cdot E, id) \longrightarrow_L v} \tag{8}$$

$$\frac{id \neq id' \qquad (E, id) \longrightarrow_L v}{(id' \mapsto v' \cdot E, id) \longrightarrow_L v} \tag{9}$$

The Σ^{SEnv}-algebra A^{SEnv} is a simple extension to A^{Exp} of Example 2.2 in which , $\mathrm{A}^{SEnv}_{funval} = T(\Sigma^{SEnv})_{funval}$, $\mathrm{A}^{SEnv}_{val} = \mathrm{A}^{SEnv}_{nat} + \mathrm{A}^{SEnv}_{funval}$, and the new language constructs and new values of sort *funval* are assigned the term algebra interpretation.

The observation signature $\Omega_{\mathrm{OBS}}^{SFun}$ was already defined in Example 1. This finishes Example 3.2. □

We now have two Relational Specifications for the programming language *Fun*: *Sub* and *SEnv*. Moreover, these Relational Specifications belong to the same class $Spec(\Omega_{\mathrm{OBS}}^{SFun})$; thus the next question we want to answer is whether *Sub* and *SEnv* are equivalent. To answer this question we first have to define a notion of equivalence between Relational Specifications in terms of the relation $\equiv_{\Omega_{\mathrm{OBS}}}$.

Definition 4 (Equivalence of Relational Specifications) Two Relational Specifications $\mathcal{S} = (\Omega^{\mathcal{S}}, \phi^{\mathcal{S}}, \mathrm{A}^{\mathcal{S}}, \Omega_{\mathrm{OBS}})$ and $\mathcal{R} = (\Omega^{\mathcal{R}}, \phi^{\mathcal{R}}, \mathrm{A}^{\mathcal{R}}, \Omega_{\mathrm{OBS}})$, with Declarative Semantics $\mathrm{M}^{\mathcal{S}}$ and $\mathrm{M}^{\mathcal{R}}$ respectively, are *observationally equivalent*, written $\mathcal{S} \cong \mathcal{R}$, if and only if:

$$\mathrm{M}^{\mathcal{S}}/\Omega_{\mathrm{OBS}} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{M}^{\mathcal{R}}/\Omega_{\mathrm{OBS}}$$ □

This definition says that $\mathcal{S}$ and $\mathcal{R}$ are equivalent if the reduct of their Declarative Semantics by observation signature are observationally equivalent.

Fact 3 The relation $\cong \subseteq Spec(\Omega_{\mathrm{OBS}}) \times Spec(\Omega_{\mathrm{OBS}})$ is an equivalence relation on $Spec(\Omega_{\mathrm{OBS}})$, for any observation signature Ω_{OBS}. □

Therefore, to establish whether $Sub \cong SEnv$ we must establish whether:

$$\mathrm{M}^{Sub}/\Omega_{\mathrm{OBS}}^{SFun} \equiv_{\Omega_{\mathrm{OBS}}^{SFun}} \mathrm{M}^{SEnv}/\Omega_{\mathrm{OBS}}^{SFun}$$

where M^{Sub} and M^{SEnv} are the Declarative Semantics of *Sub* and *SEnv* respectively. The proof of this equivalence is done in the next section.

3.5 A Proof of Equivalence

In this section we prove that *Sub* and *SEnv* are observationally equivalent using the notion of Model Correspondence of Definition 3. The initial proposition we want to prove is stated below.

Proposition 2 The Relational Specifications *Sub* and *SEnv* are observationally equivalent, i.e., $Sub \cong SEnv$. □

According to Definition 4 the proof of the above proposition requires a proof that $\mathrm{M}^{Sub}/\Omega^{SFun}_{\mathrm{OBS}}$ and $\mathrm{M}^{SEnv}/\Omega^{SFun}_{\mathrm{OBS}}$ are observationally equivalent with respect to $\Omega^{SFun}_{\mathrm{OBS}}$. Therefore, we will define a pair of relations (G, H) and prove this pair is an $\Omega^{SFun}_{\mathrm{OBS}}$-Model Correspondence between the two models. Then, Proposition 2 follows from Theorem 1.

An important advantage of using Model Correspondence as a proof method, rather than an ad hoc approach, is that this proof method structures the proofs of equivalence. This structure occurs in other proofs in this thesis, for instance, in the compiler correctness proof in Section 4.4. This suggests that this structure can be exploited, for instance, by defining semi-automatic tools to assist in the equivalence proofs. For this reason, we will present this proof in some detail so that the presentation of following proofs will be simplified.

To simplify the presentation let us abbreviate $\mathrm{M}^{Sub}/\Omega^{SFun}_{\mathrm{OBS}}$ by MS, $\mathrm{M}^{SEnv}/\Omega^{SFun}_{\mathrm{OBS}}$ by MSE, $\mathrm{A}^{Sub}/\Sigma^{SFun}_{\mathrm{OBS}}$ by AS, and $\mathrm{A}^{SEnv}/\Sigma^{SFun}_{\mathrm{OBS}}$ by ASE.

For the definition of the relation G we need to define how to close an expression with respect to an environment, i.e., how to substitute every occurrence of every free variable in an expression by its value in the environment. This closure operation is defined by the recursive function $close : exp \times env \rightarrow exp$, as follows:

$$close(e, \varepsilon_E) = e$$

$$close(n, E) = n$$

$$close(id, id' \mapsto n \cdot E) = \text{if } id \neq id' \text{ then } close(id, E) \text{ else } n$$

$$close(id, id' \mapsto (E, id'', e) \cdot E) = \text{if } id \neq id' \text{ then } close(id, E) \text{ else } \mathbf{fn}\ id''\ .\ close(e, remove(id'', E))$$

$$close(e_1 + e_2, E) = close(e_1, E) + close(e_2, E)$$

$$close(\mathbf{fn}\ id\ .\ e, E) = \mathbf{fn}\ id\ .\ close(e, remove(id, E))$$

$close(e_1(e_2), E) = close(e_1, E)(close(e_2, E))$

$close(\textbf{let } id = e_1 \textbf{ in } e_2, E) = \textbf{let } id = close(e_1, E) \textbf{ in } close(e_2, remove(id, E))$

Here, $remove(id, E)$ is an operation that returns an environment that is the same as E except that all binds $id \mapsto v$ were removed. It is easy to verify that *close* is a total function. Furthermore, the algebras AS and ASE are total. These facts will simplify the following definitions and the subsequent proofs.

Definition 5 (The Relation G) Let us define a S_{OBS}^{SFun}-sorted relation G such that each $G_s \subseteq AS_s \times ASE_s$, $s \in S_{OBS}^{SFun}$, is defined as follows:

For $s \in \{exp, var, nat\}$, G_s is defined as the identity relation on $AS_s \times ASE_s$.

$G_{funval} = \{(\textbf{fn } id \,.\, e', (E, id, e)) : e' = close(e, remove(id, E))\}$

$G_{val} = G_{funval} + G_{nat}$ □

The component G_{funval} relates function values, which are non observable according to Ω_{OBS}^{SFun}. This component is essential in the proof of Lemma 3.

Definition 6 (The Relation H) The relation name $\Rightarrow$ is the only relation in Π_{OBS}^{SFun}. Therefore, we define $H_{\Rightarrow} \subseteq (AS_{Exp} \times AS_{val}) \times (ASE_{exp} \times ASE_{val})$ to be the least relation (with respect to set inclusion) such that:

1. For all $e \in T(\Sigma_{OBS}^{SFun})_{exp}$ and $v \in T(\Sigma_{OBS}^{SFun})_{val}$:

$$((\psi_{exp}^{AS}(e), \psi_{val}^{AS}(v)), (\psi_{exp}^{ASE}(e), \psi_{val}^{ASE}(v))) \in H_{\Rightarrow_S}$$

2. For all $(e, e) \in G_{exp}$ and $(v, v') \in G_{val}$ if $(e, v) \in \Rightarrow_S{}^{MS}$ and $(e, v') \in \Rightarrow^{MSE}$ then $((e, v), (e, v')) \in H_{\Rightarrow}$.

□

Definition 6(2) includes the (non observable) function values in $H_{\Rightarrow}$. This is necessary for the proof of the following proposition.

Proposition 3 The pair of relations (G,H) is an $\Omega_{\mathrm{OBS}}^{SFun}$-Model Correspondence between MS and MSE, i.e., $(\mathrm{G}, \mathrm{H}) : \mathrm{MS} \longleftrightarrow_{\Omega_{\mathrm{OBS}}^{SFun}} \mathrm{MSE}$ □

To simplify the following presentation, whenever $e \in \mathrm{A}_{exp}^{Sub}$ and $v \in \mathrm{A}_{val}^{Sub}$, we will use $e \Rightarrow v$ for $(e, v) \in \Rightarrow^{\mathrm{MS}}$, if it is not ambiguous to do so. The same convention applies to the other relations in *Sub* and to every relation in *SEnv*.

The proof of Proposition 3 requires the proof of the requirements of Definitions 3(1) and 3(2). It is trivial to prove that Definition 3(1) is satisfied since AS and ASE give the same interpretation for all function names in $\mathrm{F}_{\mathrm{OBS}}^{SFun}$. Therefore, we will assume hereafter that $\mathrm{G} : \mathrm{AS} \Longrightarrow\!\!\times_{\Sigma_{\mathrm{OBS}}^{SFun}} \mathrm{ASE}$.

The proof of the requirements in Definition 3(2) becomes easier if we prove the following lemmas first. The proof of the Lemma 3 uses Lemmas 1 and 2 stated below, whose proofs are simpler than the proof of Lemma 3 and are carried out in a similar way. Therefore, we omit the proofs of these Lemmas 1 and 2 from this presentation.

Lemma 1 For all $E \in \mathrm{A}_{env}^{SEnv}$, $(id, id) \in \mathrm{G}_{var}$, $(e, e) \in \mathrm{G}_{exp}$, and $(v_1, v_1') \in \mathrm{G}_{val}$:

1. If there exists $v_2' \in \mathrm{ASE}_{val}$ such that $id \mapsto v_1' \cdot E \vdash e \Rightarrow v_2'$, there exists $v_2 \in \mathrm{AS}_{val}$ such that $(v_1, id, close(e, remove(id, E))) \longrightarrow_S e'$ and $e' \Rightarrow_S v_2$ and $(v_2, v_2') \in \mathrm{G}_{val}$.

2. If there exists $v_2 \in \mathrm{AS}_{val}$ such that $\mathrm{FV}(v_1) = \{\,\}$, $(v_1, id, close(e, remove(id, E))) \longrightarrow_S e'$, and $e' \Rightarrow_S v_2$, there exists $v_2' \in \mathrm{ASE}_{val}$ such that $id \mapsto v_1' \cdot E \vdash e \Rightarrow v_2'$ and $(v_2, v_2') \in \mathrm{G}_{val}$.

□

The next lemma relates the environment lookup in *SEnv* with the *close* function defined above.

Lemma 2 For all $E \in \mathrm{A}_{env}^{SEnv}$, $(id, id) \in \mathrm{G}_{var}$, and $(e, e) \in \mathrm{G}_{exp}$:

1. If there exists $v' \in \mathrm{ASE}_{val}$, such that $(E, id) \longrightarrow_L v'$,
   there exists $v \in \mathrm{AS}_{val}$, such that $close(id, E) \Rightarrow_S v$ and $(v, v') \in \mathrm{G}_{val}$
2. If there exists $v \in \mathrm{AS}_{val}$, such that $close(id, E) \Rightarrow_S v$,
   there exists $v' \in \mathrm{ASE}_{val}$, such that $(E, id) \longrightarrow_L v'$ and $(v, v') \in \mathrm{G}_{val}$

□

The following lemma relates the evaluation of expressions in *Sub* and *Env*.

Lemma 3 For all $E \in \mathrm{A}_{env}^{SEnv}$, $(e, e) \in \mathrm{G}_{exp}$:

1. If there exists $v' \in \mathrm{ASE}_{val}$ such that $E \vdash e \Rightarrow v'$,
   there exists $v \in \mathrm{AS}_{val}$ such that $close(e, E) \Rightarrow_S v$ and $(v, v') \in \mathrm{G}_{val}$
2. If there exists $v \in \mathrm{AS}_{val}$ such that $\mathrm{FV}(close(e, E)) = \{\ \}$,
   and $close(e, E) \Rightarrow_S v$, there exists $v' \in \mathrm{ASE}_{val}$ such that
   $E \vdash e \Rightarrow v'$ and $(v, v') \in \mathrm{G}_{val}$

Proof We prove 3(1) and 3(2) separately.

1(1) Recall that Proposition 2.3 states that $E \vdash e \Rightarrow v'$ if and only if there exists a proof tree for the formula $E \vdash e \Rightarrow v$ in M; the same is valid for the other relations. Therefore, we will prove the following statement:

$$\text{If } \exists\ v' \text{ and } \frac{PT'}{E \vdash e \Rightarrow v'} \text{ then } \exists\ v \text{ and } \frac{PT}{close(e, E) \Rightarrow_S v} \text{ and } (v, v') \in \mathrm{G}_{val}$$

where PT' and PT represent proof trees of the formulae below the line. This proof is by induction on the height of PT'. We present one case for each rule of the definition of $\_ \vdash \_ \Rightarrow \_$ (Example 2) used to construct the proof tree.

Rule 2(2) In this case $e = n$ and $v = n^{\mathrm{ASE}}$. Since $close(n, E) = n$ and $n \Rightarrow_S n^{\mathrm{AS}}$, the proof trees are as follows:

$$\frac{}{E \vdash n \Rightarrow n^{\mathrm{ASE}}} \qquad \frac{}{close(n, E) \Rightarrow_S n^{\mathrm{AS}}}$$

Since $n^{\mathrm{AS}} = n^{\mathrm{ASE}}$, for all $n \in T(\Sigma_{\mathrm{OBS}}^{SFun})_{nat}$, then $(n^{\mathrm{AS}}, n^{\mathrm{ASE}}) \in \mathrm{G}_{nat}$ and therefore $(n^{\mathrm{AS}}, n^{\mathrm{ASE}}) \in \mathrm{G}_{val}$.

Rule 2(3) In this case $e = id$. If there exist v' and a proof tree for $E \vdash id \Rightarrow v'$ this proof tree must be of the following form:

$$\frac{\dfrac{PT'}{(E, id) \longrightarrow_L v'}}{E \vdash id \Rightarrow v'}$$

From Lemma 2(1) we have that there exists v and a proof tree PT such that:

$$\frac{PT}{close(id, E) \Rightarrow_S v}$$

and $(v, v') \in \mathrm{G}_{val}$.

Rule 2(4) In this case $e = e_1 + e_2$. If there exists v' and a proof tree for $E \vdash e_1 + e_2 \Rightarrow v'$ this proof tree must be of the following form:

$$\frac{\dfrac{PT_1'}{E \vdash e_1 \Rightarrow v_1'} \qquad \dfrac{PT_2'}{E \vdash e_2 \Rightarrow v_2'}}{E \vdash e_1 + e_2 \Rightarrow \mathbf{plus}(v_1', v_2')}$$

where $v' = \mathbf{plus}(v_1', v_2')$. From the definition of *close* we have that $close(e_1 + e_2, E) = close(e_1, E) + close(e_2, E)$. From the inductive hypothesis there exist v_1, v_2, PT_1, and PT_2 such that:

$$\frac{\dfrac{PT_1}{close(e_1, E) \Rightarrow_S v_1} \qquad \dfrac{PT_2}{close(e_2, E) \Rightarrow_S v_2}}{close(e_1 + e_2, E) \Rightarrow_S \mathbf{plus}(v_1, v_2)}$$

and $(v_1, v_1'), (v_2, v_2') \in \mathrm{G}_{val}$. From the fact that $\mathrm{G} : \mathrm{AS} \Longrightarrow\!\!\!\!\times_{\Sigma_{\mathrm{OBS}}^{SFun}} \mathrm{ASE}$, we have that $(\mathbf{plus}^{\mathrm{AS}}(v_1, v_2), \mathbf{plus}^{\mathrm{ASE}}(v_1', v_2')) \in \mathrm{G}_{val}$.

Rule 2(5) In this case $e = \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2$. If there is a v_2' and a proof tree for $E \vdash \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \Rightarrow v_2'$, this proof tree must be of the following form:

$$\frac{\dfrac{PT_1'}{E \vdash e_1 \Rightarrow v_1'} \qquad \dfrac{PT_2'}{id \mapsto v_1' \cdot E \vdash e_2 \Rightarrow v_2'}}{E \vdash \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \Rightarrow v_2'}$$

From the definition of *close* we have that $close(\textbf{let } id = e_1 \textbf{ in } e_2, E)$ is equal to $\textbf{let } id = close(e_1, E) \textbf{ in } close(e_2, remove(id, E))$. From the inductive hypothesis there exist v_1 and PT_1 such that $(v_1, v_1') \in G_{val}$, and from Lemma 1(1) there exist v_2, PT_2, and PT_3 such that:

$$\frac{\dfrac{PT_1'}{close(e_1, E) \Rightarrow_S v_1} \quad \dfrac{PT_2'}{(v_1, id, close(e_2, remove(id, E))) \longrightarrow_S e_2'} \quad \dfrac{PT_3'}{e_2' \Rightarrow_S v_2}}{\textbf{let } id = close(e_1, E) \textbf{ in } close(e_2, remove(id, E)) \Rightarrow_S v_2}$$

and $(v_2, v_2') \in G_{val}$.

Rule 2(6) In this case $e = \textbf{fn } id \,.\, e'$. If there exists v' and a proof tree for the formula $E \vdash \textbf{fn } id \,.\, e' \Rightarrow v'$ this proof tree must be of the following form:

$$\frac{}{E \vdash \textbf{fn } id \,.\, e' \Rightarrow (E, id, e')}$$

where $v' = (E, id, e')$. Since $close(\textbf{fn } id \,.\, e', E) = \textbf{fn } id \,.\, close(e', remove(id, E))$, then:

$$\frac{}{close(\textbf{fn } id \,.\, e', E) \Rightarrow_S \textbf{fn } id \,.\, close(e', remove(id, E))}$$

From the definition of G_{funval}, $(\textbf{fn } id \,.\, close(e', remove(id, E)), (E, id, e')) \in G_{val}$.

Rule 2(7) In this case $e = e_1(e_2)$. If there exists v' and a proof tree for the formula $E \vdash e_1(e_2) \Rightarrow v'$ this proof tree must be of the following form:

$$\frac{\dfrac{PT_1'}{E \vdash e_1 \Rightarrow (E', id', e')} \quad \dfrac{PT_2'}{E \vdash e_2 \Rightarrow v_2'} \quad \dfrac{PT_3'}{id' \mapsto v_2' \cdot E' \vdash e' \Rightarrow v'}}{E \vdash e_1(e_2) \Rightarrow v'}$$

From the definition of *close* we have $close(e_1(e_2), E) = close(e_1, E)(close(e_2, E))$. If there exists a proof tree for $close(e_1(e_2), E) \Rightarrow_S v$ it must be of the following form:

$$\frac{\dfrac{PT_1}{close(e_1, E) \Rightarrow_S \textbf{fn } id \,.\, e} \quad \dfrac{PT_2}{close(e_2, E) \Rightarrow_S v_2} \quad \dfrac{PT_3}{(v_2, id, e) \longrightarrow_S e''} \quad \dfrac{PT_4}{e'' \Rightarrow_S v}}{close(e_1(e_2), E)}$$

From the inductive hypothesis the proof trees PT_1 and PT_2 exist. Furthermore:

- $(\textbf{fn } id \,.\, e, (E', id', e')) \in G_{funval}$ and from the definition of G_{funval}, we have that $close(e', remove(id', E')) = e$ and $id = id'$.

- $(v_2, v_2') \in \mathrm{G}_{val}$.

Then applying Lemma 1(1) to the formulae $id' \mapsto v_2' \cdot E' \vdash e' \Rightarrow v'$, $(v_2, id, close(e', remove(id, E'))) \longrightarrow_S e''$, and $e'' \Rightarrow_S v$, we obtain $(v, v') \in \mathrm{G}_{val}$.

This completes the proof of Lemma 3(1).

3(2) (sketch): We must prove that:

$$\text{If } \exists\, v,\ \mathrm{FV}(close(e, E)) = \{\,\} \text{ and } \frac{PT'}{close(e, E) \Rightarrow_S v} \text{ then}$$
$$\exists\, v' \text{ and } \frac{PT}{E \vdash e \Rightarrow v'} \text{ and } (v, v') \in \mathrm{G}_{val}$$

This proof is by rule induction on the definition of $\Rightarrow_S$ of Example 1. The proof is analogous to the proof Lemma 3(1).

This finishes the proof of Lemma 3. □

Using Lemma 3 we can give proof for Proposition 3.

Proof (of Proposition 3) We already argued that the proof of the requirement of Definition 3(1) is trivial, and therefore $\mathrm{G} : \mathrm{AS} \Longrightarrow\!\!\!\!\times_{\Sigma_{\mathrm{OBS}}^{SFun}} \mathrm{ASE}$. It remains to prove the requirements of Definition 3(2) are satisfied.

This proof is clearer if we first restate Lemma 3 for the case when $E = \varepsilon_E$. Lemma 3(1) then becomes: for all $(e, e) \in \mathrm{G}_{exp}$,

1. if there exists $v' \in \mathrm{ASE}_{val}$ such that $\varepsilon_E \vdash e \Rightarrow v'$
   then there exists $v \in \mathrm{AS}_{val}$ such that $close(e, \varepsilon_E) \Rightarrow_S v$ and $(v, v') \in \mathrm{G}_{val}$

Notice that applying rule (1) of Example 2 to the proof tree for the formula $\varepsilon_E \vdash e \Rightarrow v'$ we obtain $e \Rightarrow v'$ in *SEnv*. It is easy to check that $\mathrm{FV}(e) = \{\,\}$. Furthermore, $close(e, \varepsilon_E) = e$ and by applying rule (1) of Example 1 we obtain the following statement for Lemma 3(1): for all $(e, e) \in \mathrm{G}_{exp}$,

1. if there exists $v' \in \mathrm{ASE}_{val}$ such that $(e, v') \in \Rightarrow^{\mathrm{MSE}}$
   then there exists $v \in \mathrm{AS}_{val}$ such that $(e, v) \in \Rightarrow^{\mathrm{MS}}$ and $(v, v') \in \mathrm{G}_{val}$

Using a similar argument we can rewrite Lemma 3(2) as follows: for all $(e, e) \in G_{exp}$,

2. if there exists $v \in AS_{val}$ such that $FV(e) = \{\}$ and $(e, v) \in \Rightarrow^{MS}$
   then there exists $v' \in ASE_{val}$ such that $(e, v') \in \Rightarrow^{MSE}$ and $(v, v') \in G_{val}$

Let us prove first that Definition 3(2.i) is satisfied. Any pair $((e, v), (e', v'))$ can only be in $H_{\Rightarrow}$ because of either Definition 6(1) or Definition 6(2). Suppose it is because of Definition 6(1), then from Proposition 1 it follows that $(e', e) \in G_{exp}$ and $(v, v') \in G_{val}$. Suppose now that it is because of Definition 6(2). It is then clear that $e = e'$ and $(e, e') \in G_{exp}$. It follows then from Lemma 3 that $(v, v') \in G_{val}$. We then conclude that Definition 3(2.i) is satisfied

Definition 3(2.ii) follows trivially from Definition 6(1). It remains to show that $\Rightarrow$ is compatible with $H_{\Rightarrow}$ in the sense of Definition 3(2). Therefore, we must prove the following statement:

$$\text{whenever } ((e, v), (e, v')) \in H_{\Rightarrow} \text{ then:}$$
$$(e, v) \in \Rightarrow^{MS} \text{ if and only if } (e, v') \in \Rightarrow^{MSE}$$

Let us prove the left to right implication first. Suppose that $(e, v) \in \Rightarrow_S{}^{MS}$ and let us analyse the possible values for v.

1. If $v = n$ since $(v, v') \in G_{val}$ then $v = v'$. In this case it follows from Lemma 3 that there exists v'' such that $(e, v'') \in \Rightarrow^{MSE}$ and $(v, v'') \in G_{val}$. Since $v = n$ then $v = v''$, therefore $v' = v''$.

2. If $v = \textbf{fn}\ id\ .\ e$, then $((e, v), (e, v'))$ is in $H_{\Rightarrow}$ because of Definition 6(2) since there is no visible term of sort *funval*. In this case, $((e, v), (e, v')) \in H_{\Rightarrow}$ only if $(e, v) \in \Rightarrow^{MS}$ and $(e, v') \in \Rightarrow^{MSE}$, and the statement holds.

The proof of the right to left implication follows similarly. Therefore, we have proved that the requirements of Definitions 3(1) and 3(2) are satisfied, concluding that $(G, H) : MS \longleftrightarrow_{\Omega_{OBS}^{SFun}} MSE$. From Theorem 1 it follows that $MS \equiv_{\Omega_{OBS}^{SFun}}$

MSE, which according to Definition 4 establishes that $Sub \cong SEnv$. This finishes the proof of equivalence between *Sub* and *SEnv*. □

3.6 Summary and Conclusions

In this chapter we defined an equivalence relation between Relational Specifications and used this equivalence as a correctness criterion between two Relational Specifications. This equivalence relation is based on an observational equivalence in the algebraic specification sense. Moreover, the particular notion of observational equivalence used in this section is that of [ST87].

This definition provides a general criterion for correctness; the equivalence relation between specifications is (obviously) transitive, which is a desirable property of a correctness criterion. Moreover, this equivalence comes equipped with a proof method based on correspondence relations. We extended the definition of strong correspondence of [Sch87, page 241] to a relation between first order models we called Model Correspondence.

In our approach to correctness, non-deterministic programming languages are accounted for. In [Nip86] Nipkow proposes a notion of *simulation* of an algebra by another which also accounts for non-deterministic languages Our solution differs from Nipkow's simulation relation in that $\equiv_{\Omega_{OBS}}$, and therefore $\cong$, are symmetric relations. The symmetry between alternative definitions of a language's semantics expresses that *essential features* of the language (in the sense of Section 3.3, page 69) must be preserved across alternative definitions. On the other hand, non-essential characteristics may be made non-observable for correctness purposes.

An important application of the definitions of this chapter is in the problem of compiler correctness. This problem will be studied in Chapter 4 where we show that the use of Observational Equivalence extends and improves previous approaches to this problem.

Chapter 4

Compiler Correctness

In this chapter we study the problem of compiler correctness in the framework of Relational Semantics. This problem consists in establishing whether a compiler for a programming language generates "correct machine code" for programs in the language. The key aspect of this problem is to define a natural and formal meaning for "correct machine code".

We propose a solution to this problem by applying the concept of Observational Equivalence of Relational Specifications developed in Chapter 3. The initial aspect of this solution is the characterisation of the process that we call *Evaluation by Compilation*, or simply Compilation. In a Compilation, the semantics of a program is defined in three stages: the program is *compiled* into machine code which is loaded and *executed* on the machine; if this execution is successful then the result of the evaluation is *unloaded* from the machine and given as the result of the program.

Once we characterise a Compilation we show how it may be defined by a Relational Specification. The last aspect of our approach to compiler correctness is to use Observational Equivalence as the equivalence between a (standard) semantics and a Compilation of the programming language. This equivalence defines a criterion for compiler correctness and we shall argue why this is a suitable criterion.

The compiler correctness problem has received a great deal of attention in the literature. In Section 4.1 we summarise the main approaches to this problem and compare their solutions to the approach we propose in this chapter. In Section 4.3 we give an abstract characterisation of the Compilation process in terms of signatures and first order models; then we use an example to show how a concrete Compilation may be defined by a Relational Specification. In Section 4.4 we prove that the Compilation defined in Section 4.3 and a Relational Specification derived from *SEnv* (see Example 3.2) are observationally equivalent.

4.1 Introduction

In [MP67], McCarthy and Painter presented one of the earliest approaches to compiler correctness. This work consists of a proof of correctness of an algorithm for compiling arithmetic expressions into an abstract machine. In [BL69], Burstall and Landin introduced the use of algebraic methods in the compiler correctness problem. The use of an algebraic approach introduced structure on the objects involved in the correctness problem: programming language semantics, machine semantics, and the definition of the compiler.

The algebraic approach was further developed in [Mor73], where compiler correctness is characterised as the commutativity of the diagram in Figure 1 (page 94), known as the Morris Diagram. In that diagram, the nodes are algebras and the arrows are homomorphisms. The π arrow denotes the semantics of the programming language, while the μ arrow denotes the semantics of the machine language.

In the Morris Diagram, the algebra $\underline{L}$ is the initial term algebra $T(\Sigma)$ for a signature Σ which defines the programming language. Therefore, γ and π are unique homomorphisms by initiality. The proof method is also based on initiality, for if μ and δ are also homomorphisms then the commutativity of the Morris Diagram follows by uniqueness. Therefore, a proof of compiler correctness consists of proving that the arrows of the diagram are homomorphisms.

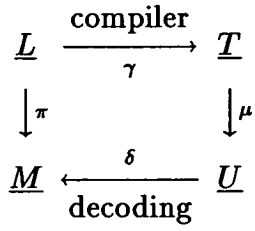

Figure 1: The Morris Diagram

One limitation of Morris' approach is that the correctness criterion requires the existence of a homomorphism δ from the algebra of the machine values into the algebra of the programming language values. To understand why this is a limitation let us consider a practical example. In the language *Fun* of Example 3.2 the expression **fn x . x** + 1 evaluates to the function value $(\varepsilon_E, \mathbf{x}, \mathbf{x}+1)$, and **fn y . y** + 1 evaluates to $(\varepsilon_E, \mathbf{y}, \mathbf{y}+1)$. It is conceivable that in a machine implementation these two expressions evaluate to the same machine value that "represents" all α-conversions of a function value like $(\varepsilon_E, \mathbf{x}, \mathbf{x}+1)$. In this case there is no homomorphism δ that makes the Morris Diagram commute.

The approach initiated by Morris inspired several investigations which set out to extend and improve the ideas presented in [Mor73]. We now discuss some of these investigations. In [TWW81], the ADJ group proposes the use of a homomorphism $\epsilon : \underline{M} \to \underline{U}$ (of encoding) to replace the homomorphism $\delta : \underline{U} \to \underline{M}$ in the Morris Diagram. A motivation for using ϵ is to overcome the limitations of the original diagram for cases like the function values discussed above.

However, the use of an encoding arrow in the correctness diagram is problematic in various ways. First, the commutativity of the diagram with $\epsilon : \underline{M} \to \underline{U}$ is not a sufficient criterion for correctness. For instance, in the case where $\underline{T}$ and $\underline{U}$ are one-point algebras and γ, μ, and ϵ are the unique homomorphism to these algebras, the diagram commutes trivially, as mentioned in [TWW81].

Another reason why ϵ does not give a sufficient correctness criterion is illustrated by a simple example. Suppose γ compiles every program $\underline{l}$ in $\underline{L}$ into the

(fixed) code sequence $\underline{t}$ in $\underline{T}$. Therefore, since μ is a function, every program has the (fixed) meaning $\mu(\underline{l})$ in $\underline{U}$. Furthermore, for every $\underline{l}$ in $\underline{L}$ suppose that ϵ maps $\pi(\underline{l})$ into (the fixed) $\mu(\underline{l})$. The diagram then commutes trivially, although we intuitively would not regard this compiler as being correct.

The degenerate case of one-point algebras seems irrelevant in practice since we expect the machine language $\underline{T}$ never to be one-point. However, the second problem discussed above suggests that errors in the compiler arrow γ can be hidden by a suitable choice of the encoding arrow ϵ. Therefore, the use of the encoding arrow in the Morris Diagram is not adequate for compiler correctness.

Furthermore, the use of an encoding arrow suffers from a pragmatic problem. In practice, we use a compiler to translate a program into machine code; we then execute the code on the machine and, if a result is produce by the execution, we expect to obtain its source level representation as the result of the program evaluation. In other words, we are interested in the results as they are represented in the algebra $\underline{M}$.

However, the existence of an encoding ϵ that makes the correctness diagram commute is not sufficient to guarantee we can (uniquely) convert from the machine representation of a result to its source language representation. In fact, a diagram with an encoding arrow only guarantees that there exists at least one result in $\underline{M}$ that corresponds to the result in $\underline{U}$ obtained from the execution of the program's code. We argue this is not sufficient from a pragmatic point of view.

A major limitation of the aforementioned algebraic approaches is that the algebraic semantics used to define the semantics of programming languages is functional. Therefore, these approaches do not directly deal with non-deterministic languages. In the approach initiated by Despeyroux [Des86] and followed by Simpson [Sim90], the nodes of the Morris Diagram become term algebras and the arrows become inductively defined relations between these algebras. Therefore, these approaches model non-deterministic languages naturally. The correctness diagrams used in [Des86,Sim90] use the encoding arrow ϵ. Therefore, both

approaches suffer from the problems discussed above.

Some authors divide the compiler correctness problem into compiler *specification* correctness and compiler *implementation* correctness [Pol81,CM86]. The former refers to the correctness of a compiler specification with respect to the programming language semantics. This category includes all above cited works. Compiler implementation correctness refers to the correctness of a compiler implementation with respect to its specification.

Various aspects distinguish specification and implementation of compilers in this context. For instance, a specification is usually defined in terms of abstract syntax of the program while the implementation may involve lexical analysis, parsing, and so forth. Furthermore, a specification does not need to be executable, and even when it is executable, it is too inefficient to be used in practical applications. On the other hand, an implementation is necessarily executable and often robust for real applications.

This distinction was first addressed in [Pol81]. In [CM86] Chirica and Martin show how to apply the ideas of the Morris Diagram to prove correctness of a compiler implementation. Although we believe that compiler implementation correctness is an important problem, it is not addressed in this thesis.

Summarising the above discussion, we have seen that most approaches to the compiler correctness problem are based on the ideas proposed in [Mor73, TWW81]. However, the original Morris Diagram (see Figure 1) is too restrictive for some practical applications. Furthermore, the use of an encoding arrow ϵ replacing the decoding arrow δ cannot be considered a sufficient criterion for compiler correctness. A natural question at this point is whether there is a suitable generalisation of the Morris Diagram which is an (intuitively) sufficient criterion and yet is general enough to address cases such the function value.

Clearly, to require the encoding arrow ϵ to be injective gives a sufficient correctness criterion in the sense that it does not suffer from the problems addressed above. However, this restriction means that any two distinct program phrases

with distinct semantics must have distinct target semantics. The example of the function values presented above shows that this restriction is too strong in some practical cases.

A less restrictive solution would be to use Hoare's idea of a representation relation [Hoa72] between the algebras $\underline{M}$ and $\underline{U}$. Another solution would be to compare the algebras $\underline{M}$ and $\underline{U}$ under observational equivalence [Rei81,ST87, NO88,Sch90]. The advantages of using observational equivalence over representation relation were discussed in Chapter 3. Moreover, in [Sch87, page 255] Schoett gives a proof that observational equivalence is more general than representation relation.

In Chapter 3, we successfully applied Observational Equivalence to the problem of equivalence between Relational Specifications. If we consider the problem of compiler correctness as an instance of equivalence between Relational Specifications we have a criterion for compiler correctness based on Observational Equivalence. It is our objective in this chapter to show how the problem of compiler correctness can be formulated in the framework developed in Chapter 3.

The use of Observational Equivalence as the criterion for compiler correctness is an improvement on previous approaches for two clear reasons. First, it is more general than previous approaches to the problem while it is still an (intuitively) sufficient criterion for correctness. Second, it is based on a formal definition which can reasoned about at the meta-level. This level of reasoning is important since it is now possible to state and prove properties about the correctness criterion, for instance, those stated in Facts 3.1, 3.2, and 3.3.

It is generally agreed that a major contribution of the ideas in [Mor73, TWW81,Pol81,CM86,Des86,Sim90] is that they present methodologies to structure the compiler and other semantic objects involved in the compiler correctness problem. However, this structure does not directly extend to the proofs of correctness which remains an ad hoc process. Various approaches have proposed ways of structuring the correctness proofs by using semi-automatic theorem

provers [MW72,Coh78,Joy89,Sim90].

Another advantage of formulating the compiler correctness problem in the framework of Chapter 3 is that we can use Model Correspondence as a proof method. As stressed in Chapter 3, Model Correspondence is an improvement over ad hoc approaches because, besides being consistent with respect to Observational Equivalence, it introduces structure into the proofs of equivalence. This structure may suggest ways in which proofs can be semi-automated, contributing to the use of this framework in practical applications.

Summarising, our approach to compiler correctness affirms the ideas proposed in previous approaches and improves these ideas in various aspects. First, it gives a more general and yet (intuitively) sufficient criterion for correctness. Second, it provides a proof method which is consistent with respect to the correctness criterion. Finally, this proof method suggests a methodology to structure the proofs of correctness which complements previous advice on how to structure the other objects involved in the compiler correctness problem. It is the main objective of this chapter to illustrate these improvements.

4.2 Definitional Preliminaries

The definitions in this, and subsequent chapters, use the concept of *directed first order signatures* in the Relational Specifications. In such a signature the sorts and function names are as in a (standard) first order signature and the relation names have an *input/output direction* assigned to them. Intuitively, the inputs of a relation characterise the programs in the programming language and the outputs characterise their results. Furthermore, a directed signature has a distinguished relation symbol, called the *initial relation* of the signature, which defines the meaning of programs by relating programs to their results.

In this chapter we need to distinguish between programs and results so that we can characterise compilers from source language programs into target language programs. In Chapter 5, we will motivate the need for directed relations in the

definition of program evaluation.

A *directed first order signature* (or simply a *directed signature*) is a quadruple (S, F, Π, π) such that (S, F) is an algebraic signature, Π is a $S^+ \times S^*$-sorted family of sets, and $\pi \in \Pi$ is a relation name called the *initial relation* of the signature. A directed signature (S, F, Π, π) is a sub-signature of (S', F', Π', π') if $(S, F, \Pi) \sqsubseteq (S', F', \Pi')$ and $\pi = \pi'$. Hereafter, the signature of any Relational Specification is a directed signature.

A relation name $\pi \in \Pi_{w \times w'}$, for some $w \in S^+$ and $w' \in S^*$, is called a *directed relation name* and is written $\pi : w \leftrightarrow w'$. If $w = s_1 \ldots s_n$ and $w' = s_{n+1} \ldots s_m$, for $n > 0$, $m \geq n$, and $t_i \in T_X(\Sigma)_{s_i}$, for $i \in [m_+]$, then t_w is an abbreviation for $t_1, \ldots, t_n$ and t'_w is an abbreviation for $t_{n+1}, \ldots, t_m$. The formula $\pi(t_1, \ldots, t_n, t_{n+1}, \ldots, t_m)$ in $F_X(\Omega)_\pi$ is then abbreviated by $\pi(t_w, t_{w'})$.

As usual, we will treat w and w' as single sorts and t_w and $t_{w'}$ as single terms whenever it does not introduce ambiguities. Therefore, $t_w \in T_X(\Sigma)$ is an abbreviation for $t_{s_i} \in T_X(\Sigma)_{s_i}$ for $i \in [n_+]$. The application of a S-sorted function to t_w or $t_{w'}$ is interpreted as the componentwise application; the same applies for substitutions or other S-sorted operations. In a formula $\pi(t_w, t_{w'})$, t_w is called the *input*, and $t_{w'}$ is called the *output*.

The following terminology is used if $\pi \in \Pi_{w \times w'}$ is the initial relation of a directed signature. A *program formula* is a formula $\pi(\overline{t_w}, t_{w'})$, where $\overline{t_w} \in T(\Sigma)$ and $t_{w'} \in T_X(\Sigma)$. The (ground) term $\overline{t_w}$ is called the *program* and, whenever $t_{w'}$ is ground, it is called the *result* of the program formula. Notice that this definition of the term "program" is different from its common usage. A program in our terminology contains an actual program together with the context in which this program evaluates.

We denote the set of program formulae in a first order signature Ω by $P_X(\Omega)$; thus $P(\Omega)$ denotes the set of ground program formulae. Clearly, $P_X(\Omega) \subset F_X(\Omega)_\pi$ whenever $X \neq \{\}$, and $P(\Omega) = F(\Omega)_\pi$.

For example, we can define a directed signature $\Omega = (S, F, \Pi, \_ \vdash \_ \Rightarrow \_)$ based on the signature Ω^{SEnv}, defined in Example 3.2, as follows: $S = S^{SEnv}$,

$\mathrm{F} = \mathrm{F}^{SEnv}$, $\Rightarrow_S$: $exp \leftrightarrow val$, and $\_ \vdash \_ \Rightarrow \_ : env \times exp \leftrightarrow val$. A program in Ω is a pair containing a ground environment and a ground *Fun* expression, e.g., $(\varepsilon_E, \mathbf{num}(1) + \mathbf{num}(2))$. A program formula in $P_{\mathrm{X}^E}(\Omega)$ is, for instance, a formula $\varepsilon_E \vdash \mathbf{num}(1) + \mathbf{num}(2) \Rightarrow v$, for some $v \in \mathrm{X}_{val}$. Hereafter, whenever it is necessary to refer to Ω we will use $\Omega^{Env} = (\mathrm{S}^{Env}, \mathrm{F}^{Env}, \Pi^{Env}, \_ \vdash \_ \Rightarrow \_)$. The directed signature Ω^{Env} will be used in Section 4.4 in the example of a compiler correctness proof.

4.3 Evaluation by Compilation

In this section we study compiler correctness in the context of the theory of Observational Equivalence developed in Chapter 3. The generality of the framework developed in Chapter 3 makes it simple to formulate the compiler correctness problem as an instance of equivalence between Relational Specifications. However, the compiler correctness problem possesses particular aspects which require specific treatment by a correctness theory. In this sense, compiler correctness is not just an instance of equivalence between Relational Specifications but has extra requirements that account for these particular aspects. It is our objective in this section to characterise these extra-requirements.

Our approach to structuring the compiler correctness problem is inspired by the early algebraic approaches [Mor73,TWW81]. However, our interpretation of the Morris Diagram (Figure 1, page 94) is similar to the approaches in [Des86, Sim90] in which the nodes of the diagram are term algebras and the arrows are inductively defined relations between the carriers of these algebras. We illustrate our interpretation of compiler correctness using the diagrams in Figure 2.

In Figure 2, the nodes $\underline{L}$, $\underline{M}$, $\underline{T}$, $\underline{U}$, and $\underline{M'}$ are term algebras, the double arrows are directed relations and the single arrow is a partial function. Both diagrams describe first order models of some directed signature Ω. The left diagram describes an Ω-model $\underline{LM}$ in which the programming language is defined by a directed relation $\pi : \underline{L} \leftrightarrow \underline{M}$ between programs and results. The right diagram

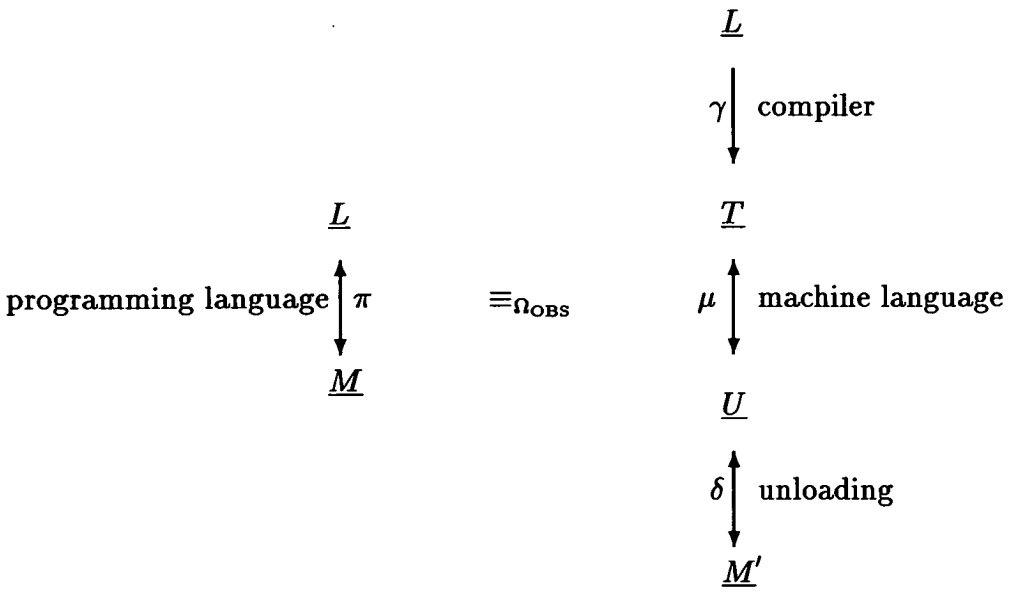

Figure 2: Evaluation by Compilation

describes what we call an *Evaluation by Compilation*, or simply a *Compilation*, which is an Ω-model $\underline{LTUM'}$ in which the semantics of a program is defined by the composition of the *compiler* $\gamma : \underline{L} \to \underline{T}$, the *machine language semantics* $\mu : \underline{T} \leftrightarrow \underline{U}$, and the *unloading* of results $\delta : \underline{U} \leftrightarrow \underline{M'}$.

The *conditions for compiler correctness* are also expressed by the diagrams in Figure 2. Let $\Omega_{\mathrm{OBS}} \sqsubseteq \Omega$ be an observation signature. The Compilation $\underline{LTUM'}$ is correct with respect to $\underline{LM}$ and Ω_{OBS} if and only if γ is a (partial) function and $\underline{LM} \equiv_{\Omega_{\mathrm{OBS}}} \underline{LTUM'}$. Whenever the Ω-models $\underline{LM}$ and $\underline{LTUM'}$ are defined by Relational Specifications in $Spec(\Omega_{\mathrm{OBS}})$, say $\mathcal{S}$ and $\mathcal{C}$ respectively, the second requirement becomes $\mathcal{S} \cong \mathcal{C}$.

In the above sense, compiler correctness should be actually called *Compilation correctness*. In fact, the question of what it means for a compiler to be correct with respect to the semantics of the programming language is vacuous if asked in isolation of the other components of the Compilation. Nevertheless, we prefer to keep the more traditional terminology and use "compiler correctness" in this presentation.

Compiler correctness is not just an instance of Observational Equivalence

between Relational Specifications because we require the compiler γ to be a partial function instead of an arbitrary directed relation. In the rest of this section we will use examples to illustrate the above ideas. However, before we move on to the examples, let us discuss the motivations for requiring the compiler to be a (partial) function.

This requirement is motivated by the behaviour we expect from the Compilation process when the language is non-deterministic. Whenever a programming language is non-deterministic, π will be a relation in which any program in $\underline{L}$ may be related to more than one result in $\underline{M}$. In a Compilation, we have the freedom to simulate this non-determinism in either of the three stages, provided the overall non-deterministic behaviour is equivalent to that of π.

However, this freedom is misleading if we consider this problem from a pragmatic point of view. In practice, whenever we need all possible results of the evaluation of a (non-deterministic) program we expect to be able to compile the program once and for all and then run the generated code as many times as necessary. However, if the non-determinism of the Compilation is produced by a non-deterministic compiler while the machine evaluation of the generated code is deterministic, we will have to recompile the program before each re-evaluation. This is clearly not what we expect in practice. The compiler correctness condition makes sure that we obtain the behaviour we expect from a non-deterministic Compilation by requiring γ to be a (partial) function.

Let us now use an example to illustrate how to define Compilations using Relational Specifications. This and following examples have some similarities with the work of Despeyroux [Des86]. Throughout the rest of this chapter we discuss the main similarities and differences between Despeyroux's work and our approach to the definition of Compilations and to compiler correctness proofs.

In agreement with previous approaches, we advocate that the starting point in the design of a Compilation is the formal semantics of the programming language. Therefore, we start by defining a Relational Specification derived from *SEnv* (see

Example 3.2) for the language *Fun* of previous examples.

First, let Ω^{Env} be the directed signature defined in Section 4.2. Let us now define a Relational Specification $Env = (\Omega^{Env}, \phi^{Env}, \mathrm{A}^{Env}, \Omega^{Fun}_{\mathrm{OBS}})$ by making ϕ^{Env} and A^{Env} to be the same as ϕ^{SEnv} and A^{SEnv} respectively. Let $\Omega^{Fun}_{\mathrm{OBS}}$ be the directed signature that has $\_ \vdash \_ \Rightarrow \_$ as the initial relation, Π^{Fun}_{OBS} has only the symbol $\_ \vdash \_ \Rightarrow \_$, and $\mathrm{S}^{Fun}_{\mathrm{OBS}}$ and $\mathrm{F}^{Fun}_{\mathrm{OBS}}$ are defined by the following BNF rules:

$$
\begin{array}{lll}
\mathit{exp} & ::= & \mathit{var} \mid \mathit{nat} \mid \mathit{exp} + \mathit{exp} \mid \textbf{let } \mathit{var} = \mathit{exp} \textbf{ in } \mathit{exp} \\
 & & \textbf{fn } \mathit{var} \,.\, \mathit{exp} \mid \mathit{exp}(\mathit{exp}) \\
\mathit{env} & ::= & \varepsilon_E \\
\mathit{var} & ::= & \mathbf{x} \mid \mathbf{y} \mid \cdots \\
\mathit{nat} & ::= & 0 \mid 1 \mid \cdots \\
\mathit{funval} & ::= & \\
\mathit{val} & ::= & \mathit{nat}
\end{array}
$$

The observable terms according to these BNF rules are the expressions of sort *exp*, the empty environment, variables, and values of sort *nat* of the form 0, 1, and so forth. The sort *funval* is observable, but there is no constructor for building observable terms of this sort. Hereafter, let M^{Env} denote the Declarative Semantics of *Env*. Notice that the Declarative Semantics M^{SEnv} of *SEnv* is equal to M^{Env}.

Once we have the programming language semantics, the next stage is to define the Compilation process. In practice, this stage starts with the definition of the machine language semantics. The next example defines the semantics of the Categorical Abstract Machine (CAM) [CCM84] by a Relation Specification.

The definition of the CAM in the following example is the first difference between our approach and Despeyroux's approach in [Des86]. Despeyroux defines the CAM by a relation that describes the entire evaluation of a sequence of machine instructions into a final result. We define a transition relation that describes the evaluation of a single machine instruction and then use the transitive-reflexive closure of this relation to evaluate sequences of instructions.

We chose this approach to define the CAM because this style of defining an abstract machine will be used in the characterisation of Compiler-debuggers in Chapter 7. Furthermore, the definition of the CAM given below will also be used in various examples in Chapter 7.

Example 1 In this example we present a complete definition of a subset of the CAM that is sufficient for the Compilation of *Fun* programs. This definition is given by a Relational Specification $Cam = (\Omega^{Cam}, \phi^{Cam}, \mathrm{A}^{Cam}, \Omega^{Cam}_{\mathrm{OBS}})$. The grammar below defines the sets S^{Cam} and F^{Cam} of Ω^{Cam}:

$$
\begin{array}{lcl}
\mathit{state} & ::= & (\mathit{stack}, \mathit{code}) \\
\mathit{stack} & ::= & \varepsilon_S \mid \mathit{val} \cdot \mathit{stack} \\
\mathit{code} & ::= & \varepsilon_C \mid \mathit{inst} \cdot \mathit{code} \\
\mathit{val} & ::= & \mathit{nat} \mid \mathit{funval} \mid (\mathit{val}, \mathit{val}) \mid ( ) \\
\mathit{nat} & ::= & \mathbf{plus}(\mathit{nat}, \mathit{nat}) \mid 0 \mid 1 \mid \cdots \\
\mathit{funval} & ::= & [\mathit{val}, \mathit{code}] \\
\mathit{inst} & ::= & \mathbf{quote}(\mathit{nat}) \mid \mathbf{push} \mid \mathbf{car} \mid \mathbf{cdr} \mid \mathbf{cons} \\
 & & \mid \mathbf{swap} \mid \mathbf{cur}(\mathit{code}) \mid \mathbf{app} \mid \mathbf{add}
\end{array}
$$

where ε_S denotes the empty stack, ε_C denotes the empty sequence of machine instructions, and ( ) denotes an empty pair. A *stack* is a sequence of machine values and a *code* is a sequence of machine instructions. We denote the concatenation of two code sequences c and c' as $c@c'$. The *val* component of a function value holds a machine environment, which is an encoding of a source language environment obtained using de Bruijn's method [dB72]. We extend X^E with the following components:

X^E_{state}	generated by st	X^E_{stack}	generated by S
X^E_{code}	generated by c	X^E_{inst}	generated by op

We present the rules that define the directed relation $\xrightarrow{\mathrm{CAM}}$: $\mathit{state} \leftrightarrow \mathit{state}$. This relation defines the evaluation of a single CAM instruction. The transitive-reflexive

closure of $\xrightarrow{\text{CAM}}$, written $\xrightarrow{\text{CAM}^*}$, defines the evaluation of arbitrary sequences of instructions. The definition of $\xrightarrow{\text{CAM}^*}$ by a set of Relational Rules is straightforward and is omitted in this presentation.

The Transition Relation $\boxed{state \xrightarrow{\text{CAM}} state}$

$$\frac{}{(v \cdot S, \mathbf{quote}(n) \cdot c) \xrightarrow{\text{CAM}} (n \cdot S, c)} \quad (1)$$

$$\frac{}{(v \cdot S, \mathbf{push} \cdot c) \xrightarrow{\text{CAM}} (v \cdot v \cdot S, c)} \quad (2)$$

$$\frac{}{((v_1, v_2) \cdot S, \mathbf{car} \cdot c) \xrightarrow{\text{CAM}} (v_1 \cdot S, c)} \quad (3)$$

$$\frac{}{((v_1, v_2) \cdot S, \mathbf{cdr} \cdot c) \xrightarrow{\text{CAM}} (v_2 \cdot S, c)} \quad (4)$$

$$\frac{}{(v_1 \cdot v_2 \cdot S, \mathbf{cons} \cdot c) \xrightarrow{\text{CAM}} ((v_2, v_1) \cdot S, c)} \quad (5)$$

$$\frac{}{(v_1 \cdot v_2 \cdot S, \mathbf{swap} \cdot c) \xrightarrow{\text{CAM}} (v_2 \cdot v_1 \cdot S, c)} \quad (6)$$

$$\frac{}{(v \cdot S, \mathbf{cur}(c') \cdot c) \xrightarrow{\text{CAM}} ([v, c'] \cdot S, c)} \quad (7)$$

$$\frac{}{(([v, c], v') \cdot S, \mathbf{app} \cdot c') \xrightarrow{\text{CAM}} ((v, v') \cdot S, c@c')} \quad (8)$$

$$\frac{}{(v_1 \cdot v_2 \cdot S, \mathbf{add} \cdot c) \xrightarrow{\text{CAM}} (\mathbf{plus}(v_2, v_1) \cdot S, c)} \quad (9)$$

The directed signature Ω^{Cam} is defined by $(\mathrm{S}^{Cam}, \mathrm{F}^{Cam}, \Pi^{Cam}, \xrightarrow{\text{CAM}})$, where $\xrightarrow{\text{CAM}}$: *state* × *state*. For all $s \in \mathrm{S}^{Cam}$, $\mathrm{A}_s^{Cam} = T(\Sigma^{Cam})_s$, except that $\mathrm{A}_{nat}^{Cam} = \{0, 1, \ldots\}$. Moreover, the Σ^{Cam}-algebra A^{Cam} interprets **plus** as the standard sum operation on the natural numbers and all other function names are given the term algebra interpretation.

The observational signature $\Omega_{\mathrm{OBS}}^{Cam}$ of *Cam* is just the signature Ω^{Cam}. Therefore, every object in this Relational Specification is observable. This finishes Example 4.1. □

Once the machine language semantics is defined, the next step in the definition of a Compilation process is to define the compiler.

Example 2 In this example we define a compiler for *Fun* into CAM code by presenting a Relational Specification $Trans = (\Omega^{Trans}, \phi^{Trans}, \mathrm{A}^{Trans}, \Omega_{\mathrm{OBS}})$. The signature Ω^{Trans} is the union of Ω^{Cam} with the relation names defined in the Relational Rules below and the algebraic signature defined by the following BNF rules:

$$
\begin{array}{lcl}
sfunval & ::= & (env, var, exp) \\
sval & ::= & nat \mid sfunval \\
env & ::= & \varepsilon_E \mid var \mapsto sval \cdot env \\
comp\_env & ::= & \varepsilon_{CE} \mid (comp\_env, var)
\end{array}
$$

where *exp* and *var* are defined in Example 3.1. In the grammar above *sval* and *sfunval* are the same as *val* and *funval* of Example 3.2. The sort names must be changed to distinguish them from *val* and *funval* of the signature Ω^{Cam}.

The compiler generates code for pairs of the form (E, e) where E is an environment and e is a *Fun* expression. In fact, the compiler constructs a machine state of the form (S, c) where c is the code generated for e with respect to the environment E and S is a compiled version of E, called the *machine environment*. Such an environment is a pair in which values are accessed using a sequence of **car** and **cdr** instructions. This sequence is a form of de Bruijn encoding of variables [dB72]. In the rest of this section, we will write a sequence of CAM code $op_1 \cdot \ldots \cdot op_n \cdot \varepsilon_C$ using the usual sequence notation $\langle op_1, \ldots, op_n \rangle$, to improve readability.

We extend X^E with the component $\mathrm{X}^E_{comp\_env}$ generated by *CE*. The rules below define the compiler $\xrightarrow{Comp}$: $env \times exp \leftrightarrow state$; this definition uses three

auxiliary relations. The relation $\longrightarrow_E$: $env \leftrightarrow comp\_env \times val$ compiles an environment into a compilation-environment and a machine environment, i.e., a pair of values as described above. The compilation-environment is necessary in the generation of lookup code of variables. The relation $\longrightarrow_{LC}$: $comp\_env \times var \leftrightarrow code$ generates lookup code for variables, and $\_ \vdash \_ \overset{Comp}{\longrightarrow} \_$: $env \times exp \leftrightarrow code$ generates code for expressions.

Environment-Compiler $\boxed{env \longrightarrow_E (comp\_env, val)}$

$$\frac{}{\varepsilon_E \longrightarrow_E (\varepsilon_{CE}, ())} \tag{1}$$

$$\frac{\mathrm{E} \longrightarrow_E (\mathrm{CE}, v)}{id \mapsto n \cdot \mathrm{E} \longrightarrow_E ((\mathrm{CE}, id), (v, n))} \tag{2}$$

$$\frac{\mathrm{E}' \longrightarrow_E (\mathrm{CE}', v') \qquad (\mathrm{CE}', id') \vdash e' \overset{Comp}{\longrightarrow} c \qquad \mathrm{E} \longrightarrow_E (\mathrm{CE}, v)}{id \mapsto (\mathrm{E}', id', e') \cdot \mathrm{E} \longrightarrow_E ((\mathrm{CE}, id), (v, [v', c]))} \tag{3}$$

Lookup Code $\boxed{(comp\_env, var) \longrightarrow_{LC} code}$

$$\frac{}{((\mathrm{CE}, id), id) \longrightarrow_{LC} \langle \mathbf{cdr} \rangle} \tag{4}$$

$$\frac{id' \neq id \qquad (\mathrm{CE}, id) \longrightarrow_{LC} c}{((\mathrm{CE}, id'), id) \longrightarrow_{LC} \langle \mathbf{car} \rangle @ c} \tag{5}$$

The Code Generator $\boxed{comp\_env \vdash exp \overset{Comp}{\longrightarrow} code}$

$$\frac{}{\mathrm{CE} \vdash n \overset{Comp}{\longrightarrow} \langle \mathbf{quote}(n) \rangle} \tag{6}$$

$$\frac{(\mathrm{CE}, id) \longrightarrow_{LC} c}{\mathrm{CE} \vdash id \overset{Comp}{\longrightarrow} c} \tag{7}$$

$$\frac{\mathrm{CE} \vdash e_1 \overset{Comp}{\longrightarrow} c_1 \qquad \mathrm{CE} \vdash e_2 \overset{Comp}{\longrightarrow} c_2}{\mathrm{CE} \vdash e_1 + e_2 \overset{Comp}{\longrightarrow} \langle \mathbf{push} \rangle @ c_1 @ \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{add} \rangle} \tag{8}$$

$$\frac{\mathrm{CE} \vdash e_1 \overset{Comp}{\longrightarrow} c_1 \qquad (\mathrm{CE}, id) \vdash e_2 \overset{Comp}{\longrightarrow} c_2}{\mathrm{CE} \vdash \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \overset{Comp}{\longrightarrow} \langle \mathbf{push} \rangle @ c_1 @ \langle \mathbf{cons} \rangle @ c_2} \tag{9}$$

$$\frac{(\mathrm{CE}, id) \vdash e \overset{Comp}{\longrightarrow} c}{\mathrm{CE} \vdash \mathbf{fn}\ id\ .\ e \overset{Comp}{\longrightarrow} \langle \mathbf{cur}(c) \rangle} \tag{10}$$

$$\frac{\mathrm{CE} \vdash e_1 \overset{Comp}{\longrightarrow} c_1 \qquad \mathrm{CE} \vdash e_2 \overset{Comp}{\longrightarrow} c_2}{\mathrm{CE} \vdash e_1(e_2) \overset{Comp}{\longrightarrow} \langle \mathbf{push} \rangle @ c_1 @ \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{cons,app} \rangle} \tag{11}$$

The Compiler $\boxed{(env, exp) \overset{Comp}{\longrightarrow} state}$

$$\frac{\mathrm{E} \longrightarrow_E (\mathrm{CE}, v) \qquad \mathrm{CE} \vdash e \overset{Comp}{\longrightarrow} c}{(\mathrm{E}, e) \overset{Comp}{\longrightarrow} (v \cdot \varepsilon_S, c)} \tag{12}$$

The algebra A^{Trans} is the trivial term algebra on all function names. The observational signature $\Omega^{Trans}_{\mathrm{OBS}}$ is just the signature Ω^{Trans}. This finishes Example 4.2. □

There are still two remaining stages in the definition of a Compilation. First we must define the unload relation and then to compose the compiler, machine semantics, and unload relation to obtain the semantics *Fun* expressions.

Example 3 This example defines the remaining components of a Compilation for *Fun* programs by presenting a Relational Specification

$$Comp = (\Omega^{Comp}, \phi^{Comp}, \mathrm{A}^{Comp}, \Omega^{Fun}_{\mathrm{OBS}})$$

The BNF rules which define the sets S^{Comp} and F^{Comp} of Ω^{Comp} are the same as in Example 2. The rules below define the unloading relation $\longrightarrow_U$: $state \leftrightarrow val$ and the evaluation of *Fun* expressions given by the relation $\_ \vdash \_ \Rightarrow \_ : env \times exp \leftrightarrow val$. The entire set of rules ϕ^{Comp} is the union of the rules below and the rules defined in Examples 1 and 2.

Unloading of Results $\boxed{state \longrightarrow_U val}$

$$\frac{}{(v \cdot S, \varepsilon_C) \longrightarrow_U v} \tag{1}$$

The Compilation $\boxed{env \vdash exp \Rightarrow val}$

$$\frac{(\mathrm{E}, e) \xrightarrow{Comp} st \qquad st \xrightarrow{\mathrm{CAM}^*} (S, \varepsilon_C) \qquad (S, \varepsilon_C) \longrightarrow_U v}{\mathrm{E} \vdash e \Rightarrow v} \tag{2}$$

In rule (1), ε_C indicates we only unload results from successful states. Rule (2) is what we call a *Compilation rule*; it defines the evaluation of e in E by compiling e into code for the CAM, executing this code by using the transitive-reflexive closure of $\xrightarrow{\mathrm{CAM}}$, and unloading the result from the final CAM state.

The Σ-algebra A^{Comp} agrees with the algebras defined in Examples 1 and 2 on all function names. The observation signature $\Omega^{Fun}_{\mathrm{OBS}}$ is the same as for the Relational Specification *Env* defined above. Hereafter, let M^{Comp} denote the Declarative Semantics of *Comp*.

This finishes Example 4.3. □

In Examples 1, 2, and 3 we illustrated how to structure the design of a Compilation in Relational Specification. We started by defining the abstract machine which gives the target language for the compiler. We then defined the compiler and the unloading relation, and composed them using a *Compilation rule*. We believe this is a pattern which frequently occurs in the design of compilers in practice.

There are other possibilities in the definition of a Compilation of *Fun* into CAM. For instance, it is straightforward to define a Compilation that uses the definition of the CAM given in [Des86].

4.4 A Proof of Compiler Correctness

In Section 4.3 we presented a criterion for compiler correctness and then used an example to illustrate how to design a Compilation in Relational Semantics. We now have two definitions of the language *Fun* given by the Relational Specifications *Env* and *Comp*. In this section we will prove the *correctness* of *Comp* with respect to *Env*.

In the framework of Chapter 3, this "correctness" would mean establishing whether $Env \cong Comp$. However, *Comp* defines a Compilation and we are interested in a *compiler correctness proof* of *Comp* with respect to *Env*. Therefore, we must prove two conditions. First, we must show that the rules for $\overset{Comp}{\longrightarrow}$ defined in Example 2 define a (partial) function. Then, we prove that $Env \cong Comp$.

Our motivation in this sections is to emphasise that there are more requirements in a compiler correctness proof than in a proof of equivalence between Relational Specifications. Furthermore, we want illustrate the use of Model Correspondence in compiler correctness proofs.

Following the same conventions used in Section 3.5, whenever $E \in \mathrm{A}_{env}^{Env}$, $e \in \mathrm{A}_{exp}^{Env}$, and $v \in \mathrm{A}_{val}^{Env}$ we write $E \vdash e \Rightarrow v$ for $(E, e, v) \in (\_ \vdash \_ \Rightarrow \_)^{\mathrm{M}^{Env}}$ when it does not introduce ambiguities; the same convention applies to other relations in *Env* and for every relation in *Comp*.

The first step in the compiler correctness proof of *Comp* with respect to *Env* is to establish whether $\overset{Comp}{\longrightarrow}$ defines a function. This is stated in the following proposition.

Proposition 1 For all $E \in \mathrm{A}_{env}^{Comp}$ and $e \in \mathrm{A}_{exp}^{Comp}$ if there exist $st, st' \in \mathrm{A}_{state}^{Comp}$ such that $(E, e) \overset{Comp}{\longrightarrow} st$ and $(E, e) \overset{Comp}{\longrightarrow} st'$ then $st = st'$.

Proof (Sketch) The definition of $\overset{Comp}{\longrightarrow}$ in rule 2(12) is given in terms of the relations $\longrightarrow_E$ and $\_ \vdash \_ \overset{Comp}{\longrightarrow} \_$. This proof follows easily by induction on the height of the proof trees for the relation $\_ \vdash \_ \overset{Comp}{\longrightarrow} \_$. We omit the details of the proof from this presentation. □

The next stage in the compiler correctness proof is to establish the equivalence between *Env* and *Comp*.

Proposition 2 The Relational Specifications *Env* and *Comp* are observationally equivalent, i.e., $Env \cong Comp$. □

Hereafter, we drop the superscript *Fun* from the signature $\Omega_{\mathrm{OBS}}^{Fun}$ whenever it does not introduce ambiguities. The proof of Proposition 2 follows the structure of the proof of Proposition 3.3 that established the Observational Equivalence of *Sub* and *SEnv*. For this proof we must establish whether:

$$\mathrm{M}^{Env}/\Omega_{\mathrm{OBS}} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{M}^{Comp}/\Omega_{\mathrm{OBS}}$$

For this proof we define a pair of relations (G, H) and prove that this pair is an Ω_{OBS}-Model Correspondence between $\mathrm{M}^{Env}/\Omega_{\mathrm{OBS}}$ and $\mathrm{M}^{Comp}/\Omega_{\mathrm{OBS}}$. To simplify the presentation of the proof let ME denote $\mathrm{M}^{Env}/\Omega_{\mathrm{OBS}}$, MC denote $\mathrm{M}^{Comp}/\Omega_{\mathrm{OBS}}$, AE denote $\mathrm{A}^{Env}/\Sigma_{\mathrm{OBS}}$, and AC denote $\mathrm{A}^{Comp}/\Sigma_{\mathrm{OBS}}$. Clearly, the algebras AE and AC are total algebras. This fact will simplify the presentation of the following proofs.

Definition 1 (The Relation G) Let us define a $\mathrm{S}_{\mathrm{OBS}}$-sorted relation G such that each $\mathrm{G}_s \subseteq \mathrm{AE}_s \times \mathrm{AC}_s$, $s \in \mathrm{S}_{\mathrm{OBS}}$, is defined as follows

For $s \in \{exp, env, var, nat\}$, G_s is the identity relation on $\mathrm{AE}_s \times \mathrm{AC}_s$.

$$\mathrm{G}_{funval} = \{((\mathrm{E}, id, e), [S, c]) : E \longrightarrow_E (CE, \mathrm{S}) \text{ and } (CE, id) \vdash e \xrightarrow{Comp} c\}$$

$$\mathrm{G}_{val} = \mathrm{G}_{funval} + \mathrm{G}_{nat}$$ □

Definition 2 (The Relation H) The relation name $\_ \vdash \_ \Rightarrow \_$ is the only relation in Π_{OBS}. Therefore, we define $\mathrm{H}_{\_\vdash\_\Rightarrow\_}$ as the smallest (with respect to set inclusion) such that:

1. For all $E \in T(\Sigma_{\mathrm{OBS}})_{env}$, $e \in T(\Sigma_{\mathrm{OBS}})_{exp}$, and $v \in T(\Sigma_{\mathrm{OBS}})_{val}$ then:

$$((\psi^{\mathrm{AE}}_{env}(E), \psi^{\mathrm{AE}}_{exp}(e), \psi^{\mathrm{AE}}_{val}(v)), (\psi^{\mathrm{AC}}_{env}(E), \psi^{\mathrm{AC}}_{exp}(e), \psi^{\mathrm{AC}}_{val}(v))) \in \mathrm{H}_{\_\vdash\_\Rightarrow\_}$$

2. For all $(E, E) \in \mathrm{G}_{env}$, $(e, e) \in \mathrm{G}_{exp}$, and $(v, v') \in \mathrm{G}_{val}$,
   if $(E, e, v) \in \_ \vdash \_ \Rightarrow \_^{\mathrm{ME}}$ and $(E, e, v') \in \_ \vdash \_ \Rightarrow \_^{\mathrm{MC}}$ then:

$$((E, e, v), (E, e, v')) \in \mathrm{H}_{\_\vdash\_\Rightarrow\_}$$ □

We now have to prove Proposition 3, and the proof of Proposition 2 then follows from Theorem 3.1.

Proposition 3 The pair (G, H) of Definitions 1 and 2 is an Ω_{OBS}-Model Correspondence between ME and MC, i.e., (G, H) : ME $\longleftrightarrow_{\Omega_{\mathrm{OBS}}}$ MC. □

The proof of this proposition is easier if we prove the following lemmas first. The first lemma relates the evaluation of variables in *Env* and *Comp*. The proof of this lemma is simpler than the proof of Lemma 2 and carried out in the same way. Therefore, we omit the proof of Lemma 1 from this presentation.

Lemma 1 For all $(\mathrm{E}, \mathrm{E}) \in \mathrm{G}_{env}$, $(id, id) \in \mathrm{G}_{var}$, $S' \in \mathrm{A}_{stack}$, and $c' \in \mathrm{A}_{code}$:

1. If there exists $v \in \mathrm{AE}_{val}$ such that $(E, id) \longrightarrow_L v$
   then there exists $v', v'' \in \mathrm{AC}_{val}$, $\mathrm{CE} \in \mathrm{AC}_{comp\_env}$, $c \in \mathrm{A}_{code}$
   such that $E \longrightarrow_E (CE, v'')$, $CE \vdash id \stackrel{Comp}{\longrightarrow} c$,
   $(v'' \cdot S', c@c') \stackrel{\mathrm{CAM}^*}{\longrightarrow} (v' \cdot S', c')$ and $(v, v') \in \mathrm{G}_{val}$
2. and conversely □

The next lemma relates the evaluation of arbitrary expressions in *Env* and *Comp*. The reader familiar with the proof of correctness given in [Des86] will notice that Lemma 2 is similar to the central correctness theorem of that proof. In this sense, our proof of correctness in this example is similar to the proof in [Des86]. However, one of the main advantages of using Model Correspondence is that this proof method is formally consistent with respect to Observational Equivalence.

Lemma 2 For all $(\mathrm{E}, \mathrm{E}) \in \mathrm{G}_{env}$, $(e, e) \in \mathrm{G}_{exp}$, $S' \in \mathrm{AC}_{stack}$, and $c' \in \mathrm{A}_{code}$:

1. If there exists $v \in \mathrm{AE}_{val}$ such that $E \vdash e \Rightarrow v$
   then there exists $v', v'' \in \mathrm{AC}_{val}$, $\mathrm{CE} \in \mathrm{AC}_{comp\_env}$, $c \in \mathrm{A}_{code}$
   such that $E \longrightarrow_E (CE, v'')$, $CE \vdash e \stackrel{Comp}{\longrightarrow} c$,
   $(v'' \cdot S', c@c') \stackrel{\mathrm{CAM}^*}{\longrightarrow} (v' \cdot S', c')$, and $(v, v') \in \mathrm{G}_{val}$
2. And conversely

Proof We prove 2(1) and 2(2) separately.

2(1) This proof is by induction on the height of the proof tree for $E \vdash e \Rightarrow v$ in *Env*.

Rule 3.2(2) In this case $e = n$, and the proof tree in *Env* is as follows:

$$\frac{}{\mathrm{E} \vdash n \Rightarrow n^{\mathrm{AE}}}$$

In *Comp* we have the following proof trees:

$$\frac{}{\mathrm{E} \longrightarrow_E (\mathrm{CE}, v'')} \qquad \frac{}{\mathrm{CE} \vdash n \overset{Comp}{\longrightarrow} \langle \mathbf{quote}(n) \rangle}$$

and the following single CAM transition:

$$(v'' \cdot S', \langle \mathbf{quote}(n) \rangle @ c') \overset{\mathrm{CAM}}{\longrightarrow} (n^{\mathrm{AC}} \cdot \mathrm{S}', c')$$

Therefore, $n^{\mathrm{AE}} = n^{\mathrm{AC}}$ and it is trivial to conclude that $(n^{\mathrm{AE}}, n^{\mathrm{AC}}) \in \mathrm{G}_{val}$.

Rule 3.2(3) The proof of this case follows immediately from Lemma 1.

Rule 3.2(4) The proof of this case is simpler than for the other cases and we omit it here.

Rule 3.2(5) The proof of this case is similar to the case of rule 3.2(7). It is therefore omited.

Rule 3.2(6) In this case $e = \mathbf{fn}\ id\ .\ e'$, and the proof tree for this expression is as follows:

$$\frac{}{E \vdash \mathbf{fn}\ id\ .\ e' \Rightarrow (E, id, e')}$$

In *Comp* the corresponding proof trees are as follows:

$$\frac{}{\mathrm{E} \longrightarrow_E (\mathrm{CE}, v'')} \qquad \frac{\dfrac{PT}{(\mathrm{CE}, id) \vdash e' \overset{Comp}{\longrightarrow} c}}{\mathrm{CE} \vdash \mathbf{fn}\ id\ .\ e' \overset{Comp}{\longrightarrow} \langle \mathbf{cur}(c) \rangle}$$

and we have the following single CAM transition:

$$(v'' \cdot S', \langle \mathbf{cur}(c) \rangle @ c') \xrightarrow{\text{CAM}} ([v'', c] \cdot \mathrm{S}', c')$$

It follows from the definition of G_{funval} that $((E, id, e'), [v'', c]) \in \mathrm{G}_{val}$.

Rule 3.2(7) In this case $e = e_1(e_2)$. If there exists v and a proof tree for $E \vdash e_1(e_2) \Rightarrow v$ in *Env* this proof tree must be of the following form:

$$\frac{\dfrac{PT_1}{E \vdash e_1 \Rightarrow (E', id, e')} \quad \dfrac{PT_2}{E \vdash e_2 \Rightarrow v_2} \quad \dfrac{PT_3}{id \mapsto v_2 \cdot E' \vdash e' \Rightarrow v}}{E \vdash e_1(e_2) \Rightarrow v}$$

Therefore, in *Comp* we must have the following proof trees:

$$\frac{}{\mathrm{E} \longrightarrow_E (\mathrm{CE}, v'')} \qquad \frac{\dfrac{PT_1'}{\mathrm{CE} \vdash e_1 \xrightarrow{Comp} c_1} \quad \dfrac{PT_2'}{\mathrm{CE} \vdash e_2 \xrightarrow{Comp} c_2}}{\mathrm{CE} \vdash e_1(e_2) \xrightarrow{Comp} \langle \mathbf{push} \rangle @ c_1 @ \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{cons,app} \rangle}$$

and the following CAM transition sequence:

$$\begin{array}{ll}
(v'' \cdot S', \langle \mathbf{push} \rangle @ c_1 @ \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{cons,app} \rangle @ c') \xrightarrow{\text{CAM}} & \\
(v'' \cdot v'' \cdot S', c_1 @ \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{cons,app} \rangle @ c') \xrightarrow{\text{CAM}^*} & (1) \\
(v_1' \cdot v'' \cdot S', \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{cons,app} \rangle @ c') \xrightarrow{\text{CAM}} & \\
(v'' \cdot v_1' \cdot S', c_2 @ \langle \mathbf{cons,app} \rangle @ c') \xrightarrow{\text{CAM}^*} & (2) \\
(v_2' \cdot v_1' \cdot S', \langle \mathbf{cons,app} @ c' \rangle) \xrightarrow{\text{CAM}} & \\
((v''', c''], v_2') \cdot S', \langle \mathbf{app} \rangle @ c') \xrightarrow{\text{CAM}} & \\
((v''', v_2') \cdot S', c'' @ c') \xrightarrow{\text{CAM}^*} & (3) \\
(v' \cdot S', c') &
\end{array}$$

And the existence of the above transition sequence is justified as follows. From the inductive hypothesis applied to PT_1, PT_1', and (1) we have $((E', id, e'), v_1') \in \mathrm{G}_{val}$. Therefore, $v_1 = [v''', c'']$ where $E' \longrightarrow_E (CE', v''')$ and $(CE', id) \vdash e' \xrightarrow{Comp} c''$. From the inductive hypothesis applied to PT_2, PT_2', and (2) we obtain that $(v_2, v_2') \in \mathrm{G}_{val}$. Moreover, $id \mapsto v_2 \cdot E' \longrightarrow_E ((CE', id), (v''', v_2'))$; thus we can apply the inductive hypothesis on PT_3 and (3) and obtain that $(v, v') \in \mathrm{G}_{val}$. This finishes the proof of 2(1).

2(2) (Sketch): This proof is analogous to the proof of part 2(1), and follows by induction on the length of the CAM evaluation of the code sequence of the expressions.

This finishes the proof of Lemma 2. □

Proof (of Proposition 3) We must prove that (G, H) obeys the requirements of Definitions 3.3(1) and 3.3(2). Since AE and AC give the same interpretation for each symbol in Σ_{OBS} the proof of Definition 3.3(1) is trivial. Therefore, we can assume that $\mathrm{G} : \mathrm{AE} \Longrightarrow\!\!\!\!\times_{\Sigma_{\mathrm{OBS}}} \mathrm{AC}$. It remains to prove Definition 3.3(2).

First, let us prove Definition 3.3(2.i) is satisfied. Any pair $((E, e, v), (E', e', v'))$ can only be in $\mathrm{H}_{\_\vdash\_\Rightarrow\_}$ because of either Definition 2(1) or Definition 2(2). Suppose it is because of Definition 2(1), then from Proposition 3.1 it follows that $(E, E') \in \mathrm{G}_{env}$, $(e, e') \in \mathrm{G}_{exp}$, and $(v, v') \in \mathrm{G}_{val}$. Suppose now that it is because of Definition 2(2). It is then clear that $E = E'$, $e = e'$, $(E, E') \in \mathrm{G}_{env}$, and $(e, e') \in \mathrm{G}_{exp}$, and it follows from Lemma 2 that $(v, v') \in \mathrm{G}_{val}$. We then conclude that Definition 3.3(2.i) is satisfied

Definition 3.3(2.ii) follows trivially from Definition 2(1). It remains to show that $\_ \vdash \_ \Rightarrow \_$ is compatible with $\mathrm{H}_{\_\vdash\_\Rightarrow\_}$ in the sense of Definition 3.3(2). i.e., we must prove the following statement:

$$\text{whenever } ((E, e, v), (E, e, v')) \in \mathrm{H}_{\_\vdash\_\Rightarrow\_} \text{ then:}$$

$$(E, e, v) \in \_ \vdash \_ \Rightarrow \_^{\mathrm{ME}} \text{ if and only if } (E, e, v') \in \_ \vdash \_ \Rightarrow \_^{\mathrm{MC}}$$

The proof, similarly to that of Proposition 3.3 presented in Section 3.5, follows by analysing the possibilities of v and v'. We omit the details from this presentation.

Therefore, we have proved that (G, H) satisfies Definitions 3.3(1) and 3.3(2), concluding that $(\mathrm{G}, \mathrm{H}) : \mathrm{ME} \longleftrightarrow_{\Omega_{\mathrm{OBS}}^{SFun}} \mathrm{MC}$. From Theorem 3.1 it follows that $\mathrm{ME} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{MC}$ and from Definition 3.4 we obtain that *Env* $\cong$ *Comp*. This finishes the proof of equivalence between *Env* and *Comp*. □

Comment

There is no representation relation from MC to ME because there is no homomorphism from a subset of A^{Comp}_{funval} to A^{Env}_{funval} that will make the diagram of Figure 1 commute. For instance, the machine closure [( ), 3] is the result of the evaluation of the *Fun* expression **fn** *id* . 3 on the empty environment for all formal parameters *id*. Therefore, this is a practical example in which the use of Observational Equivalence is more general than the use of representation relations in compiler correctness. Furthermore, it is possible to find examples in which there is no representation relation in either direction between two algebras, as demonstrated in [Sch87, page 255].

Let us now establish the equivalence between *Comp* and the Relational Specification *Sub* defined in Example 3.1. We will assume that the signatures of *Sub* and *SEnv* were modified to be directed signatures in the obvious way. Furthermore, let the relation $\Rightarrow$: $exp \leftrightarrow val$ be the initial relation of those signatures. The problem in establishing the equivalence between *Sub* and *Comp* is that *Comp* does not define the relation $\Rightarrow$. This relation is the initial relation in *Sub* and belongs to its observation signature. A simple solution is to add the following rule to the set ϕ^{Comp}:

$$\frac{\varepsilon_E \vdash e \Rightarrow v}{e \Rightarrow v}$$

This rule is the same as rule 3.2(1) and serves the purpose of defining the relation $\Rightarrow$ in terms of the relation $\_ \vdash \_ \Rightarrow \_$. Then we build a Relational Specification $SComp = (\Omega^{SComp}, \phi^{SComp}, A^{SComp}, \Omega^{SFun}_{OBS})$ where Ω^{SComp} is equal to Ω^{Comp} except that the initial relation is $\Rightarrow$ instead of $\_ \vdash \_ \Rightarrow \_$. The set of rules ϕ^{SComp} is ϕ^{Comp} with the above rule, and A^{SComp} is the same as A^{Comp}. Therefore, $SComp \in Spec(\Omega^{SFun}_{OBS})$.

Let MSC denote $M^{SComp}/\Omega^{SFun}_{OBS}$ where M^{SComp} denotes the Declarative Semantics of *SComp*. If *SEnv* is the specification of Example 3.2 we can state the following proposition.

Proposition 4 The Relational Specifications *SComp* and *SEnv* are observationally equivalence, i.e, $SComp \cong SEnv$.

Proof (Sketch) Let G be the relation of Definition 1 and J be the relation obtained from H of Definition 2 as follows:

$$((E, e, v), (E', e', v')) \in \mathrm{H} \text{ if and only if } ((e, v), (e', v')) \in \mathrm{J}$$

Now, in both *SComp* and *SEnv* the relation $\Rightarrow$ is defined by the following rule:

$$\frac{\varepsilon_E \vdash e \Rightarrow v}{e \Rightarrow v}$$

Therefore, since $(\mathrm{G}, \mathrm{H}) : \mathrm{ME} \longleftrightarrow_{\Omega_{\mathrm{OBS}}^{Fun}} \mathrm{MC}$, it is easy to prove that $(\mathrm{G}, \mathrm{J}) : \mathrm{MSE} \longleftrightarrow_{\Omega_{\mathrm{OBS}}^{SFun}} \mathrm{MSC}$. Therefore, we conclude that $SComp \cong SEnv$. □

Corollary 1 The Relational Specifications *SComp* and *Sub* are observationally equivalent, i.e, $SComp \cong Sub$.

Proof Immediate consequence of Propositions 3.2 and 4 and the transitivity of the relation $\cong$. □

4.5 Summary and Conclusions

In this chapter we defined a notion of compiler correctness by applying the results of Chapter 3. This notion was achieved by first characterising the process we call a Compilation, in which a program is evaluated in three stages: *translation* into machine code, *execution* of the code on the machine, and *unloading* of the results from the machine state. The criterion for compiler correctness defined in this chapter is based on Observational Equivalence, as defined in Chapter 3. This criterion is more general than the previous approaches to the compiler correctness problem. In this sense, we argue that the results of this chapter improve on previous work.

Furthermore, our approach does not suffer from the problems found in other approaches (e.g., in [TWW81,Des86,Sim90]), where an encoding arrow ϵ is used to replace the arrow δ of Figure 1 (page 94). Using this approach compilers that we would intuitively regard as being incorrect can be proved correct by a suitable choice of an encoding arrow, as discussed in Section 4.1.

Finally, we argue that the use of Model Correspondence in the proofs of compiler correctness is also an improvement on previous approaches. We believe that the structure introduced by Model Correspondence in the correctness proofs can be exploited to make our methods to scale up to practical examples. A possible investigation in this direction is to study the use of semi-automatic theorem provers in the proofs of correctness. Semi-automatic proof by induction is being investigated, for instance, in the work of the DReaM group [Bun88, BvHHS91]. Since the bulk of the proofs of compiler correctness using the Model Correspondence are inductive proofs, we believe that such proofs are amenable to semi-automatic treatment by theorem provers. Moreover, in [HP92] Hannan and Pfenning showed a proof of correctness of a compiler, from a simple functional programming language to CAM, using the LF Logical Framework [HHP87]. This also reinforces our belief that such proofs can be semi-automated.

Chapter 5

Evaluation of Programs Based on Relational Semantics

In this chapter we define *program evaluation* and a notion of *evaluation step* based on a Relational Specification. Although semantics-based program evaluation has been investigated as a problem on its own, our investigation is motivated and guided towards the definition of an evaluation step to be used in debuggers. The ability to evaluate a program through a *step by step* evaluation of its sub-programs is essential for debugging programs. Therefore, a notion of an evaluation step is the most basic and fundamental component in the specification of debuggers.

We first define the *Computational Semantics* of a Relational Specification, by a (non-deterministic) procedure that searches for a satisfying substitution for arbitrary formulae in the specification. We then define *program evaluation* as the search for a satisfying substitution for a program formula. The Computational Semantics is a terminal transition system inspired by the stack-of-stack (SOS) operational semantics of logic programming defined in [And91]. The transitions of this system provide an *operational interpretation* for the Relational Rules which complements the abstract interpretation given by the Declarative Semantics.

The first technical result in this chapter is a proof of the *soundness* of the Computational Semantics with respect to the Declarative Semantics of Defini-

tion 2.3. This result is important since it guarantees that the notions of program evaluation and evaluation step defined in this chapter are consistent with with respect to the declarative interpretation of a Relational Specification.

Not all Relational Specifications have an operational interpretation that agrees with our intuition about the programming language, and we use some examples to illustrate this problem. We then define the class of *Data-driven Relational Specifications* with the objective of characterising the specifications that have an intuitive operational interpretation. The second major result of this chapter is the proof of the *completeness* of the Computational Semantics of a Data-driven Specification.

Although we argue that the operational interpretation of a Data-driven Specification is intuitive, the transition of its Computational Semantics is non-deterministic and therefore unsuitable for debugging. We then characterise a subclass of Data-driven Specifications which we call *deterministic* and prove the Computational Semantics of a Deterministic Specification is *monogenic*. Finally, we define a *program evaluation step* to be a transition of the this monogenic Computational Semantics. The class of Deterministic Specification and this notion of an evaluation step will be used in Chapter 6 in the specification of debuggers.

5.1 Introduction

In this section we discuss other approaches to program evaluation based on the semantics of a programming language, and compare some aspects of those approaches to our own. Some authors have proposed program evaluation based on a denotational semantics of the programming language, for instance in the PSG system [BS86]. However, we concentrate our presentation on the approaches that use a semantic formalism closer to Relational Semantics.

CENTAUR [CIK89] is a system that, among other features, generates interpreters from Natural Semantics specifications of programming languages. The

semantic specifications are written in the meta-language TYPOL [Des88], which the system compiles into a Mu-PROLOG [Nai83b] program that may be executed. Running the TYPOL specification with a program as its input yields program evaluation. One difference between TYPOL and our approach is that there is no formal proof of the correctness of the compilation of TYPOL into Mu-PROLOG with respect to some underlying mathematical meaning of Natural Semantics. Furthermore, the definition of a notion of evaluation step is not taken into consideration in the CENTAUR approach to program evaluation. For this reason, certain characteristics of the evaluation of TYPOL programs, like backtracking and the use of control strategies of Mu-PROLOG [Nai83a,Nai85] make the definition of an intuitive notion of evaluation step very difficult.

The Animator Generator [Ber91a] is a system that generates program animators from a Structural Operational Semantics of the programming language. Program Animators may be used for several purposes, including debugging. Because of the similarity between Relational Semantics and the formalism used in the Animator Generator, one would imagine that it would not be difficult to use the Animator Generator's concept of "compilation step" in this thesis. However, an animation step is unnecessarily complex for our needs and this extra-complexity would make reasoning about debuggers more difficult.

Moreover, in Berry's approach there is no definition of a concept similar to the Declarative Semantics. His definition of program evaluation is the meta-semantics of the formalism used in the Animator Generator. Therefore, to use the notion of program evaluation of the Animator Generator in the framework of this thesis would not be as trivial as one first imagined.

The parallel between Relational Specifications and Definite Clause Programs (DCP) that we made in Section 2.5 extends to the problem of program evaluation. The Computational Semantics of a Relational Specification is the counterpart of an operational semantics for DCP. The latter is usually defined by SLD-resolution or some of its variants; other authors have proposed the use of proof tree construc-

tion in the definition of the operational semantics of DCP [DF87]. Furthermore, the concept of a *data-driven* DCP defined in [DM85] is similar to the notion of *directed relations* introduced in Section 4.2.

Therefore, it is not a complete surprise that some concepts defined in this chapter for a Relational Specification have been addressed in the context of DCP. Our approach differs from the logic programming approach in two major aspects. First, we formalise the use of term evaluation in a Relational Specification. Although recently some authors have proposed a theoretical account of term evaluation in logic programming [Boy91a,Boy91b], this problem has often been left as an implementation issue. For instance, in PROLOG term evaluation is usually performed using the "IS" predicate [CM87].

The second and major aspect that distinguishes our approach from approaches related to logic programming is that our main objective is the definition of a notion of evaluation step based on the Computational Semantics. In this sense, SLD-resolution and proof tree construction do not provide a suitable notion of Computational Semantics since they yield definitions of an evaluation step that are difficult to reason about.

We present a Computational Semantics based on the SOS semantics of [And91] extended with term evaluation, and derive from this semantics a notion of evaluation step that agrees with our intuition about the language. Furthermore, this definition of evaluation step is simpler to present and easier to reason about than the approaches based on proof tree construction or SLD-resolution.

Finally, the concept of directed relations is essential to our definition of an evaluation step since it is used in the definition of a Data-driven Relational Specification. A more restricted form of directed relations was used in [Chi89] also with the objective of defining semantic based program evaluation.

5.2 Definitional Preliminaries

As discussed before, this chapter defines a computational semantics for the Relational specifications. This computational semantics computes substitutions for meta-variables in formulae using unification, e.g., Robinson's algorithm [Rob65]. In this section we present some definitions related to substitution and unification that will be used in the following sections.

Let (S, F, Π, π) be a directed signature, θ be a substitution as defined in Section 2.3 (page 43), and X be an S-sorted set of meta-variables.

A substitution θ is *variable pure* if ran $\theta \subseteq X$. If $t \in T_X(\Sigma)$ then θ is a *renaming* for t if θ is variable-pure and for each $x \in \mathcal{V}(t)$, $\theta(x) \notin \mathcal{V}(t)$. If θ is a renaming for t then $t\theta$ is the *variant* of t by θ. If $t, t' \in T_X(\Sigma)_s$, then t and t' *unify* if there exists a substitution θ such that $t\theta = t'\theta$; in this case θ is called the *unifier* for t and t'. A substitution θ is called the *most general unifier* (m.g.u.) for t and t' if for each unifier θ' for t and t' there exists a substitution θ'' such that $\theta' = \theta\theta''$. Hereafter we assume that *Unify* is an algorithm that takes two terms as arguments and returns their m.g.u. if the terms unify.

The definitions in this chapter use the concept of *simplification* of terms by evaluation. For this definition, let A be a Σ-algebra and X be a S-sorted set of meta-variables, such that $A_s \cap X_s = \{\ \}$, for each $s \in S$. Intuitively, a simplification of a term $t \in T_X(\Sigma)$ with respect to a Σ-algebra A is a term in $t \in T_{(X+A)}(\Sigma)$ in which the evaluation function ψ^A is applied to all sub-terms of t (including t) that can be evaluated, leaving the sub-terms with meta-variables in X unaltered.

Let A be a Σ-algebra. The *simplification* of a Σ-term is a S-sorted partial function $\chi = \{\chi_s\}$, such that $\chi_s : T_{(X+A)}(\Sigma)_s \rightarrow T_{(X+A)}(\Sigma)_s$ is defined as follows:

1. if $t = a$, and $a \in A_s$, then $\chi_s(t) = a$.

2. if $t = x$, and $x \in X_s$, then $\chi_s(t) = x$.

3. if $t = \sigma(t_1, \ldots, t_n)$, $t \in T_{\mathrm{A}}(\Sigma)_s$, and $t \in \mathrm{dom}\ \psi^{\mathrm{A}}$ then $\chi_s(t) = \psi^{\mathrm{A}}(t)$.

4. if $t = \sigma(t_1, \ldots, t_n)$, $\sigma : s_1 \times \ldots \times s_n \rightarrow s$, and $t \notin T_{\mathrm{A}}(\Sigma)_s$, then $\chi_s(t) = \sigma(\chi_{s_1}(t_1), \ldots, \chi_{s_n}(t_n))$.

From the above definition, it is easy to conclude that whenever $\mathrm{X} = \{\ \}$, χ is the same as the evaluation function ψ. Simplification of Σ-terms extends naturally to simplification of Ω-formulae. We also use χ to denote simplification of formulae. Hereafter we write χ^{A} for χ to emphasise its dependency on the Σ-algebra A.

The following paragraphs introduce some standard concepts about transition systems. A *transition system* is a pair $(\Gamma, \rightarrow)$ where Γ is the set of *states* and $\rightarrow \subseteq \Gamma \times \Gamma$ is the *transition relation*. For $(st, st') \in \Gamma \times \Gamma$, $(st, st') \in \rightarrow$ is written $st \rightarrow st'$.

A *terminal transition system* is a triple $(\Gamma, \rightarrow, \mathcal{T})$ where $(\Gamma, \rightarrow)$ is a transition system and $\mathcal{T} \subseteq \Gamma$ is a set of *terminal* states that satisfy: for each $st \in \mathcal{T}$ and all $st' \in \Gamma$, $st \not\rightarrow st'$. We also use the term *abstract machine* to refer to a terminal transition system.

We use $\rightarrow^*$ to denote the *transitive-reflexive closure* of $\rightarrow$, and $\rightarrow^+$ to denote the *transitive closure*. Let $st \in \Gamma$; *st succeeds* in $(\Gamma, \rightarrow, \mathcal{T})$ if and only if there exists a state $st' \in \mathcal{T}$ such that $st \rightarrow^* st'$; *st fails* if and only if *st* does not succeed and there exists $st' \in \Gamma \backslash \mathcal{T}$ such that $st \rightarrow^* st'$, and for all $st'' \in \Gamma$, $st' \not\rightarrow st''$; *st terminates* if *st* either succeeds or fails; and *st diverges* if *st* neither succeeds nor fails. A transition system is *monogenic* if and only if for all $st \in \Gamma$ there exists at most one $st' \in \Gamma$ such that $st \rightarrow st'$.

5.3 Computational Semantics of Relational Specifications

In this section we define the Computational Semantics of a Relational Specification. From a general perspective, this semantics defines an operational interpret-

ation of the Relational Specification which we may use to understand the programming language in a concrete and evaluation-oriented way. In this sense, the Computational Semantics provides an interpretation of the Relational Specification that complements the more abstract interpretation given by the Declarative Semantics defined in Chapter 2.

These two interpretations have different applications. On the one hand, it is simpler to understand a Relational Specification of a programming language by referring to its operational interpretation, as defined by the Computational Semantics. On the other hand, it is easier to formally reason about the Relational Specification by referring to its Declarative Semantics.

From the perspective of this thesis in particular, the Computational Semantics is used in the definition of program evaluation and a notion of an evaluation step based on a Relational Specification. Therefore, the Computational Semantics is a fundamental component of the formal specification of debuggers, discussed in Chapter 6.

The definition of a notion of evaluation step is the main motivation for the definition of the Computational Semantics in this thesis. For this reason, the Computational Semantics is defined as a transition system whose transition relation is then interpreted as the steps of the program evaluation. This transition system is inspired by the SOS semantics of logic programming defined in [And91].

There exist other possibilities for a Computational Semantics different from the SOS semantics, for instance proof tree construction [DF87] or some variant of SLD-resolution [Llo87]. A particular definition of proof tree construction is used in the Animator Generator for evaluation of programs based on Structural Operational Semantics [Ber91a]. However, when compared to different notions of evaluation step based on those approaches, the notion of evaluation step defined in this chapter is simpler to define, more natural to understand, and easier to reason about.

Let $\mathcal{S} = (\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Relational specification where $\Omega = (\mathrm{S}, \mathrm{F}, \Pi, \pi)$,

and X be an S-sorted set of meta-variables, such that $X_s \cap A_s = \{\ \}$ for each $s \in S$.

The basic objects of the Computational Semantics of the Relational Specification $\mathcal{S}$ are goal stacks and evaluation states. A *goal stack* is a sequence of formulae in $F_X(\Omega)$, and the empty goal stack is denoted by ε. We use α with indexes to stand for an arbitrary goal stack, and write the concatenation of goal stacks α_1 and α_2 as $\alpha_1 \cdot \alpha_2$. If $f \in F_X(\Omega)$, we write f to stand for the goal stack $\langle f \rangle$, such that $\alpha_1 \cdot \langle f \rangle \cdot \alpha_2$ is written $\alpha_1 \cdot f \cdot \alpha_2$. The context and the convention on the names used to denote formulae and goal stacks will be enough to resolve ambiguities.

An *evaluation state* (or simply a state) is either the constant [ ] (the *failure state*), or a term $[\theta : \alpha]$, where α is a goal stack, and θ is a substitution of terms in $T_{(X+A)}(\Sigma)$ for meta-variables in X. Notice that a term in $T_{(X+A)}(\Sigma)$ has sub-terms from the carriers of the Σ-algebra A.

Informally, the Computational Semantics of $\mathcal{S}$ is a non-deterministic terminal transition system whose states contain the Ω-formulae for which we are trying to find a satisfying substitution. At each state of this system one formula in the goal stack is *selected*. If there exists a rule in ϕ whose conclusion unifies with this selected goal, then in the following state the selected goal is replaced by the premisses of the rule, which become new goals. The substitutions resulting from the unifications at each transition are kept in the state. If a successful state is reached, i.e., a state with an empty goal stack, the substitution in this state is a satisfying substitution for each formulae in the goal stack of predecessor states.

Moreover, if $[\theta_0 : f]$ is the initial state for some Ω-formula f, and $[\theta : \varepsilon]$ is a state after various transitions of the Computational Semantics, then θ is a satisfying substitution for f.

Definition 1 (Computational Semantics) The *Computational Semantics* of a Relational Specification $\mathcal{S} = (\Omega, \phi, A, \Omega_{OBS})$ is a terminal transition system $\mathcal{CS} = (\Gamma_{CS}, \overset{CS}{\Rightarrow}, \mathcal{T}_{CS})$, where Γ_{CS} is the set of evaluation states whose formulae are

in $F_{\mathbf{X}}(\Omega)$, $\mathcal{T}_{\mathrm{CS}}$ is the set of (terminal) states of the form $[\theta : \varepsilon]$, and $\overset{\mathrm{CS}}{\Rightarrow}$ is the transition relation defined by the following rewrite rules:

1. Goal Expansion

$$[\theta : \alpha_1 \cdot f \cdot \alpha_2] \overset{\mathrm{CS}}{\Rightarrow} [\theta\theta' : \alpha_1 \cdot f_1 \cdot \ldots \cdot f_n \cdot \alpha_2]$$

where:

- $(\{f_1, \ldots, f_n\}, ce)$, $n \geq 0$, is a variant of a (non-deterministically chosen) rule $\mathcal{R} \in \phi$, such that no meta-variable in the rule appears on the left of the $\overset{\mathrm{CS}}{\Rightarrow}$ arrow.
- $f\theta \in \mathrm{dom}\ \chi^{\mathbf{A}}$ and $\theta' = \mathit{Unify}(\chi^{\mathbf{A}}(f\theta), ce)$,

2. Failure

$$[\theta : \alpha_1 \cdot f \cdot \alpha_2] \overset{\mathrm{CS}}{\Rightarrow} [\,]$$

whenever Goal Expansion does not apply.

□

In the definition of the transition relation $\overset{\mathrm{CS}}{\Rightarrow}$, the state on the left of the arrow is called the *current state*. At the current state, the formula f is the *selected goal* and $\mathcal{R} \in \phi$ is the *selected rule*. The Relational Rule $(\{f_1, \ldots, f_n\}, ce)$ is a suitable variant of $\mathcal{R}$ that does not contain any meta-variables already appearing in the current state.

The renaming of meta-variables is (deterministically) achieved by subscribing the meta-variables of $\mathcal{R}$ with a counter (initially 0) which indicates the number of transitions from an initial state to the current state at which $\mathcal{R}$ is the selected rule. This process of renaming the meta-variable is known in the literature as *standardising the meta-variables apart* [Llo87, page 41]. Definition 6.2 in Chapter 6 and the results in Chapter 7 depend on a deterministic choice for the fresh meta-variables.

Some points in the above definition need clarification. In contrast to the SOS semantics of [And91], there is no *backtrack stack* in the states of the above transition system; instead, the rules are non-deterministically chosen. Therefore, only one rule is *selected* at each transition of $\mathcal{CS}$. Another possibility would be to use the idea of a backtrack stack, i.e., a sequence of evaluation states, and change the transition relation $\overset{\text{CS}}{\Rightarrow}$ to be a relation between backtrack stacks. With this alteration, at each transition of the new $\overset{\text{CS}}{\Rightarrow}$ every rule in ϕ whose conclusion unifies with the selected goal is selected, and an evaluation state is added to the current backtrack stack for each selected rule. Therefore, each possibility for the search is "tried in parallel". However, for the Relational Specifications that are suitable for debugging, both alternatives to define the Computational Semantics are equivalent. We therefore preferred the version without backtrack stack because it is simpler to present and easier to reason about.

In the Goal Expansion rule, the simplification function χ^{A} is used to evaluate as many sub-terms of $f\theta$ as possible. The objective is to mimic the use of term evaluation in the derivation of the Declarative Semantics of a Relational Specification in Section 2.4 (page 52). Intuitively, the use of simplification makes terms like 5 and $\mathbf{plus}(3,2)$ unify.

The use of an explicit failure state in the transition system is unusual. However, when debugging a program, using the Computational Semantics for its evaluation, we do want to know that a failure state is reached and also to be able to syntactically recognise this state. This necessity for the failure state shall become clearer when we treat the specification of debuggers in Chapter 6.

The concepts and terminology defined in Section 5.2 for an arbitrary terminal transition system also apply to the $\mathcal{CS}$ system; thus, a state $st \in \Gamma$ *succeeds* in $\mathcal{CS}$ if and only if $st \overset{\text{CS}}{\Rightarrow}^{*} [\theta : \varepsilon]$; st *fails* if and only if st does not succeed and $st \overset{\text{CS}}{\Rightarrow}^{*} [\,]$; and st *diverges* if it neither succeeds nor fails. We say that a formula $f \in F_{\text{X}}(\Omega)$ succeeds (fails, diverges) in $\mathcal{CS}$ if the *initial state* $[\theta_0 : f]$ succeeds (fails, diverges); in an initial state $[\theta_0 : f]$ the formula f is called the *query*. If a

formula f succeeds in $\mathcal{CS}$ the finite transition sequence:

$$[\theta_0 : f] \overset{\mathcal{CS}}{\Rightarrow} [\theta_0\theta_1 : \alpha_1] \overset{\mathcal{CS}}{\Rightarrow} \ldots \overset{\mathcal{CS}}{\Rightarrow} [\theta_0\theta_1 \ldots \theta_n : \epsilon]$$

is called a *$\mathcal{CS}$-sequence* of f with length n; the composition of substitutions $\theta_0\theta_1 \ldots \theta_n$ with domain restricted to the variables of f is called an *answer* for f in $\mathcal{CS}$.

Hereafter, $\mathcal{CS}$ stands for the Computational Semantics of an arbitrary Relational Specification. Whenever we want to refer to the Computational Semantics of a particular Relational Specification $\mathcal{S}$ we write $\mathcal{CS}^{\mathcal{S}}$.

We shall prove the Computational Semantics of $\mathcal{S}$ is sound with respect to the Declarative Semantics of $\mathcal{S}$. This means that if an answer θ is found for f in $\mathcal{CS}$ then all ground instantiations of $f\theta$ are satisfiable in the Declarative Semantics of $\mathcal{S}$. The following lemma relates simplification and evaluation of terms. Intuitively, it states that simplification followed by a substitution and an evaluation is the same as the evaluation of the instantiated term. The lemma is used in the proof of the Soundness Theorem.

Lemma 1 For all terms $t \in T_{(\mathbf{X}+\mathbf{A})}(\Sigma)_s$ and substitutions θ and θ', the following holds:

$$\chi^{\mathbf{A}}(t\theta)\theta' \in \text{dom } \psi^{\mathbf{A}} \quad \text{if and only if} \quad t\theta\theta' \in \text{dom } \psi^{\mathbf{A}}$$

and if both sides of this equivalence are true then $\psi^{\mathbf{A}}(\chi^{\mathbf{A}}(t\theta)\theta') = \psi^{\mathbf{A}}(t\theta\theta')$.

Proof (Sketch) This follows by simple structural induction on t. □

Lemma 1 generalises trivially to the evaluation of formulae using Ψ. Now we are ready to prove the first main result about the Computational Semantics: its soundness with respect to the Declarative Semantics defined in Section 2.4 (Definition 2.3).

Theorem 1 (Soundness of the Computational Semantics)
Let $(\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Relational Specification and $\mathcal{CS}$ be its Computational Semantics. For all $f \in F_{\mathrm{X}}(\Omega)_\pi$, and $\pi \in \Pi$:

if there exists a $\mathcal{CS}$-sequence of f of the form $[\theta_0 : f] \overset{\mathrm{CS}}{\Rightarrow}^* [\theta : \varepsilon]$ then for all ground instantiation $\overline{\theta}$ of $f\theta$, if $f\theta\overline{\theta} \in \mathrm{dom}\ \Psi^{\mathrm{A}}$ then $\Psi^{\mathrm{A}}(f\theta\overline{\theta}) \in \pi^{\mathrm{A}}$.

Proof This proof consists of building a ϕ^{A}-tree for $\Psi^{\mathrm{A}}(f\theta\overline{\theta})$ using the $\mathcal{CS}$-sequence of f. The proof is by induction on the length of the $\mathcal{CS}$-sequence. As it is often the case in proofs by induction, we shall prove a stronger proposition of which the desired theorem is a trivial corollary.

For all $f \in F_{\mathrm{X}}(\Omega)_\pi$, goal stacks α_1 and α_2, and substitution θ:

if there exists a transition sequence of the form $[\theta : \alpha_1 \cdot f \cdot \alpha_2] \overset{\mathrm{CS}}{\Rightarrow}^* [\theta' : \varepsilon]$ then for all ground instantiation $\overline{\theta}$ of $f\theta'$, if $f\theta'\overline{\theta} \in \mathrm{dom}\ \Psi^{\mathrm{A}}$ then $\Psi^{\mathrm{A}}(f\theta'\overline{\theta}) \in \pi^{\mathrm{A}}$.

Base Case For a single transition of the form:

$$[\theta : \alpha_1 \cdot f \cdot \alpha_2] \overset{\mathrm{CS}}{\Rightarrow} [\theta\theta' : \varepsilon]$$

α_1 and α_2 are empty and $(\{\ \}, ce)$ is a variant of the selected rule in the transition such that $\theta' = \mathit{Unify}(\chi^{\mathrm{A}}(f\theta), ce)$. Therefore, every ground instantiation $\overline{\theta}$ of $ce\theta'$ is a ground instantiation of $(\chi^{\mathrm{A}}(f\theta))\theta'$, such that $ce\theta'\overline{\theta} = \chi^{\mathrm{A}}(f\theta)\theta'\overline{\theta}$. If $\chi^{\mathrm{A}}(f\theta)\theta'\overline{\theta} \in \mathrm{dom}\ \Psi^{\mathrm{A}}$ then from Lemma 1 we have $f\theta\theta'\overline{\theta} \in \mathrm{dom}\ \Psi^{\mathrm{A}}$ and $\Psi^{\mathrm{A}}(\chi^{\mathrm{A}}(f\theta)\theta'\overline{\theta}) = \Psi^{\mathrm{A}}(f\theta\theta'\overline{\theta})$; thus $\Psi^{\mathrm{A}}(ce\theta'\overline{\theta}) = \Psi^{\mathrm{A}}(f\theta\theta'\overline{\theta})$ and

$$\overline{\Psi^{\mathrm{A}}(ce\theta'\overline{\theta})}$$

is a ϕ^{A}-tree of $\Psi^{\mathrm{A}}(f\theta\theta'\overline{\theta})$. Therefore, from Proposition 2.3 $\Psi^{\mathrm{A}}(f\theta\overline{\theta}) \in \pi^{\mathrm{A}}$, which proves the base case.

Inductive Step For a transition sequence of length $m > 1$ of the form:

$$[\theta : \alpha_1 \cdot f \cdot \alpha_2] \overset{\mathrm{CS}}{\Rightarrow} [\theta\theta_1 : \alpha_1 \cdot f_1 \cdot \ldots \cdot f_n \cdot \alpha_2] (\overset{\mathrm{CS}}{\Rightarrow})^{m-1} [\theta'' : \varepsilon]$$

such that $(\{f_1, \ldots, f_n\}, ce)$, $n \geq 0$, is a variant of the selected rule in the first transition. By the inductive hypothesis for every ground instantiation $\overline{\theta}$ of $f_i\theta''$, for $i \in [n]$, $\Psi^{\mathrm{A}}(f_i\theta''\overline{\theta}) \in \pi^{\mathrm{A}}$, therefore there is a ϕ^{A}-tree T_i with root $\Psi^{\mathrm{A}}(f_i\theta''\overline{\theta})$ for each $i \in [n]$. Moreover, $\theta_1 = \mathit{Unify}(\chi^{\mathrm{A}}(f\theta), ce)$.

Therefore, every ground instantiation $\overline{\theta}$ of $ce\theta''$ is a ground instantiation of $\chi^{\mathrm{A}}(f\theta)\theta''$, such that $ce\theta''\overline{\theta} = \chi^{\mathrm{A}}(f\theta)\theta''\overline{\theta}$. If $\chi^{\mathrm{A}}(f\theta)\theta''\overline{\theta} \in \mathrm{dom}\ \Psi^{\mathrm{A}}$ then from Lemma 1 we have that $\Psi^{\mathrm{A}}(\chi^{\mathrm{A}}(f\theta)\theta''\overline{\theta}) = \Psi^{\mathrm{A}}(f\theta\theta''\overline{\theta})$; thus $\Psi^{\mathrm{A}}(ce\theta''\overline{\theta}) = \Psi^{\mathrm{A}}(f\theta\theta''\overline{\theta})$ and

$$\frac{\mathrm{T}_1 \ldots \mathrm{T}_n}{\Psi^{\mathrm{A}}(ce\theta''\overline{\theta})}$$

is a Ψ^{A}-tree for $\Psi^{\mathrm{A}}(f\theta\theta''\overline{\theta})$; thus $\Psi^{\mathrm{A}}(f\theta\theta''\overline{\theta}) \in \pi^{\mathrm{A}}$.

This finishes the proof of the stronger proposition. For $\theta = \theta_0$ and α_1 and α_2 equal to ε this proposition is exactly the Soundness theorem. □

In the rest of this section we shall discuss the completeness of the Computational Semantics of $\mathcal{S}$ with respect to its Declarative Semantics. Informally, completeness means that if there exists a substitution θ for a formula f such that $f\theta$ belongs to the Declarative Semantics, then f succeeds in $\mathcal{CS}$ with answer θ. This notion of Completeness is often used as the completeness of SLD-resolution, e.g., in [Llo87].

Because of the use of term evaluation in the Declarative Semantics, the Computational Semantics is not complete in general. To illustrate this problem, let us consider the Relational Rule 2.1(2) of the Relational Specification $Exp = (\Omega^{Exp}, \phi^{Exp}, \mathrm{A}^{Exp})$ in Example 2.1 (page 48):

$$\frac{e_1 \Rightarrow_S n_1 \qquad e_2 \Rightarrow_S n_2}{e_1 + e_2 \Rightarrow_S \mathbf{plus}(n_1, n_2)} \qquad 2.1(2)$$

The derivation of the inductive system from the Relational Inductive Definition ϕ_{Exp} using A^{Exp} is achieved by first obtaining all possible ground instantiations of each rule in ϕ_{Exp}, and then applying the evaluation of terms to the instantiated

rules to obtain the rules of the inductive system. Let us denote this inductive system by $\phi^{\mathrm{A}^{Exp}}$. For instance, the following rule is a ground instance of the Relational Rule 2.1(2):

$$\frac{\mathbf{num}(1) \Rightarrow_S 1 \qquad \mathbf{num}(2) \Rightarrow_S 2}{\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow_S \mathbf{plus}(1,2)} \qquad (*)$$

Applying term evaluation to the above rule we obtain:

$$\frac{\mathbf{num}(1) \Rightarrow_S 1 \qquad \mathbf{num}(2) \Rightarrow_S 2}{\mathbf{num}(1) + \mathbf{num}(2) \Rightarrow_S 3} \qquad (**)$$

where the term $\mathbf{plus}(1,2)$ is evaluated to 3 and the other terms are evaluated to themselves because of their term interpretation in A^{Exp}. The rule $(*)$ does not belong to $\phi_{Exp}^{\mathrm{A}^{Exp}}$ whereas the rule $(**)$ does. Therefore, $\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow_S 3$ is obviously valid in *Exp* because it belongs to the inductively defined set $\mathrm{I}(\phi^{\mathrm{A}^{Exp}})$.

However, the Computational Semantics $\mathcal{CS}^{Exp}$ fails to find a satisfying substitution for this formula (in this case the identity substitution) because the term 3 in the formula does not unify with the term $\mathbf{plus}(v_1, v_2)$ in the conclusion of the Relational Rule 2.1(2). In this sense, $\mathcal{CS}^{Exp}$ is not complete with respect to Declarative Semantics of *Exp*.

There are various solutions for this incompleteness problem in logic programming. For instance, it is possible to use strategies in the Computational Semantics to delay the unification until all meta-variables in terms like $\mathbf{plus}(v_1, v_2)$ are instantiated. Some of these strategies are discussed in [Nai83a,Nai85]. Alternatively, we can use other search strategies to define $\mathcal{CS}$, for instance, S-SLD-resolution [Boy91b]. These general solutions are not suitable in the context of this thesis because they would make the notion of evaluation step more difficult to define and reason about.

The objective of the aforementioned solutions is to improve the completeness of the resulting Computational Semantics for an arbitrary Relational Specification. However, as we will demonstrate in the next section, some Relational Specifications are not suitable as the basis for the specification of debuggers.

Therefore, such Relational Specifications are excluded from the definition of an evaluation step and from the investigations of Chapters 6 and 7.

Since our main goal in this chapter is to achieve a definition of an evaluation step, we can approach the incompleteness problem from another direction. We first characterise the subset of Relational Specifications that will be used to define an evaluation step. Then, we establish the completeness of the Computational Semantics of specifications in this subset.

Our approach departs from the logic programming approach in the sense that we do not improve the completeness of the $\mathcal{CS}$ system, but characterise the class of Relational Specification for which $\mathcal{CS}$ is complete. This is sufficient for our objectives since this class contains the Relational Specifications we are interested in as the basis for the specification of debuggers.

In our approach, an (indirect) solution to the incompleteness of the $\mathcal{CS}^{Exp}$ system is to replace rule 2.1(2) by the following rule:

$$\frac{e_1 \Rightarrow_S v_1 \qquad e_2 \Rightarrow_S v_2 \qquad \mathbf{num}(\mathbf{plus}(v_1, v_2)) \Rightarrow_S v}{e_1 + e_2 \Rightarrow_S v} \qquad 2.1(2)'$$

Intuitively, this rule is semantically equivalent to rule 2.1(2) and *Exp* using this rule fits in the class of specifications that are suitable as the basis of specifications of debuggers. We conjecture that *Exp* with this new rule is complete under the condition that, after a goal expansion using this rule, the premisses $e_1 \Rightarrow_S v_1$ and $e_2 \Rightarrow_S v_2$ are selected before $\mathbf{num}(\mathbf{plus}(v_1, v_2)) \Rightarrow_S v$.

The reason for requiring the two leftmost premisses to be selected before the rightmost one is to guarantee that when the the rightmost premiss is selected in a state of $\mathcal{CS}^{Exp}$ the instance of the term $\mathbf{plus}(v_1, v_2)$ at this state is ground; thus, its simplification yields a *nat* number, and the premiss unifies with the conclusion of the Relational Rule 2.1(1) that defines constant expressions. A term of the form $\mathbf{plus}(v_1, v_2)$ does not unify with the conclusion of that rule. The generalisation of this discussion is studied in the next section.

5.4 Program Evaluation and Evaluation Step

In this section we define the notion of evaluation step that will be used in the specification of debuggers in Chapter 6. It was mentioned above that the Computational Semantics gives an operational interpretation for the Relational Specifications. For this interpretation to be suitable in a debugger it must agree with our intuition about the programming language, as discussed in Chapter 1.

Initially, we describe what we think is a natural and intuitive operational interpretation of a Relational Specification. Not all Relational Specifications have such an intuitive operational interpretation, and we use some examples to illustrate this problem. We then characterise a Data-driven Relational Specification with the objective of obtaining a class of specifications that has an intuitive operational interpretation, and prove the completeness of the Computational Semantics of an arbitrary specification in this class.

Although we argue the operational interpretation given by the Computational Semantics of a Data-driven Specification is intuitive, the transitions of this Computational Semantics are non-deterministic, and therefore unsuitable for debugging. We then characterise a subclass of Data-driven Specifications which we call deterministic and prove the Computational Semantics of a Deterministic Specification is monogenic. Finally, we define a program evaluation step to be a transition of the this monogenic Computational Semantics.

The diagram in Figure 1 shows the inclusions of the subclasses of Relational Specifications we define in this section.

Let us now describe an operational interpretation for Relational Rules which we consider intuitive. To illustrate this notion we use the rule for sum expressions defined in the previous section:

$$\frac{e_1 \Rightarrow_S v_1 \qquad e_2 \Rightarrow_S v_2 \qquad \mathbf{num}(\mathbf{plus}(v_1, v_2)) \Rightarrow_S v}{e_1 + e_2 \Rightarrow_S v} \qquad 2.1(2)'$$

An operational interpretation of the above rule says that the evaluation of an expression $e_1 + e_2$ is composed of the steps of the evaluation of e_1, plus the steps

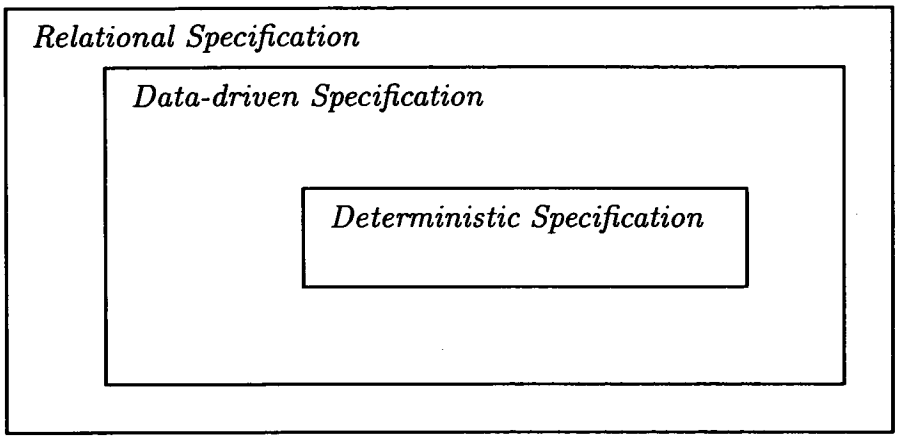

Figure 1: Inclusions of Classes of Relational Specifications

of the evaluation of e_2 (not necessarily in this order), plus the step necessary to perform the sum operation. Let us now consider, as another example, the rule 2.1(3) in *Exp*:

$$\frac{e_1 \Rightarrow_S v_1 \qquad (v_1, id, e_2) \longrightarrow_S e_2' \qquad e_2' \Rightarrow_S v_2}{\textbf{let } id = e_1 \textbf{ in } e_2 \Rightarrow_S v_2} \qquad 2.1(3)$$

The operational interpretation of this rule is as follows: the evaluation of the expression **let** $id = e_1$ **in** e_2 is composed of the steps of the evaluation of e_1 into v_1, followed by the steps of the substitution of the value v_1 for every free occurrence of id in e_2 resulting in an expression e_2', followed by the steps of the evaluation of e_2' into the result v_2, which is the result of the original expression.

In the interpretation of the rule 2.1(3) there is an implicit order on the evaluation; for instance, we evaluate e_1 before making the substitution of v_1 for id in e_2. This ordering is derived from an input/output dependency among the premisses. For instance, the input of premiss $(v_1, id, e_2) \longrightarrow_S e_2'$ *depends on* the output v_1 of the premiss $e_1 \Rightarrow_S v_1$. However, in the interpretation of rule 2.1(2)$'$ there is no such an ordering, since the premiss are independent. This suggests we use a partial order on the premisses in our operational interpretation of the rules.

The definition of Relational Specifications (Definition 2) allows inference rules

which do not have the operational interpretation discussed above. For instance, the two rules below do not have the above operational interpretation.

$$\frac{id \mapsto \tau' \cdot TE \vdash e : \tau}{TE \vdash \mathbf{fn}\ id\ .\ e : \tau' \rightarrow \tau}$$

$$\frac{\mathbf{num}(n_1) \Rightarrow_S n_2 \qquad \mathbf{num}(n_2) \Rightarrow_S n_1}{\mathbf{num}(n_1) = \mathbf{num}(n_2)}$$

The first rule defines type inference for functional abstractions. If we try to interpret this rule as we did above for rules 2.1(2) and 2.1(3) we have problems in "guessing" the value of the input meta-variable τ' in the premiss. The second rule defines equality between two *nat* numbers using the rules that define the relation $\Rightarrow_S$ in Example 2.1. The operational interpretation of this rule is problematic because there is no intuitive criterion for choosing a premiss to start the evaluation.

It may be possible to give another operational interpretation for the above rules. However, this would make the notion of an evaluation step more difficult to reason about, e.g., in the proof of debugger correctness in Chapter 7. Therefore, we prefer to keep our basic framework simpler and suggest the investigation of an alternative operational interpretation to account for the above rules as an extension to this framework.

The definitions below characterise a Data-driven Relational Specification (or simply a Data-driven Specification). The main characteristic of such a specification is that the premisses of Data-driven Relational Rules are ordered by a partial order that reflects a dependency of the inputs of a premiss on the outputs of other premisses, as illustrated above. The motivation for this definition is to characterise the Relational Specification which have the operational interpretation discussed in the above examples.

The definition of a Data-driven Specification begins with the definition of structural terms and structural formulae. Intuitively, the evaluation of a structural term does not change its syntactical structure, hence the name structural.

This property is essential in the proof of completeness of the Computational Semantics of a Data-drive Specification.

A term $t \in T_X(\Sigma)_s$ is *structural* with respect to a Σ-algebra A if and only if either $t \in X_s$, i.e., t is a meta-variable, or $t = \sigma(t_1, \ldots, t_n)$, σ has the term algebra interpretation in A, and each t_i, for $i \in [n_+]$, is a structural term. A formula $\pi(t_w, t_{w'}) \in F_X(\Omega)_\pi$ is *structural* if and only if each term in $t_{w'}$ is structural. A *structural program formula* is a structural formula whose inputs are ground terms. Hereafter, $P_X(\Omega)$ denotes the set of structural program formulae over Ω with meta-variables in X, unless indicated otherwise. For instance, the formula $\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow_S \mathbf{plus}(v_1, v_2)$ is not a structural formula, whereas $\mathbf{num}(1)+\mathbf{num}(2) \Rightarrow_S v$ is structural.

Now, we define a (formulae) dependency relation on the set $F_X(\Omega)$. Intuitively, a formula f depends on another formula f' if there are meta-variables in the input terms of f that also occurs in some output term of f'. Let $f = \pi(t_w, t_{w'})$ and $f' = \pi'(t_u, t_{u'})$ be formulae in $F_X(\Omega)$. The *predecessor relation* $\prec$ on $F_X(\Omega)$ is defined as follows: $f \prec f'$ (read as f is a predecessor of f') if and only if $\mathcal{V}(t_{w'}) \cap \mathcal{V}(t_u) \neq \{\ \}$. The *formula dependency relation* $\preceq$ on $F_X(\Omega)$ is taken to be the transitive and reflexive closure of $\prec$.

Definition 2 (Data-driven Specification) A Relational Rule (Pre, *ce*) with formulae in $F_X(\Omega)$, and $ce = \pi(t_w, t_{w'})$, is *Data-driven* if all conditions below hold:

1. The relation $\preceq$ is anti-symmetric in Pre, i.e., the relation $\preceq$ is a partial order in Pre.

2. Each $f \in$ Pre is a structural formula.

3. For each $\pi'(t_u, t_{u'}) \in$ Pre, for all $x \in \mathcal{V}(t_u)$ either $x \in \mathcal{V}(t_w)$, or $x \in \mathcal{V}(t_{v'})$ for some $\pi''(t_v, t_{v'}) \in \text{Pre}\backslash\{\pi'(t_u, t_{u'})\}$.

4. The conclusion *ce* is structural, and furthermore every term in t_w is structural.

5. For *ce* as above and for all $x \in \mathcal{V}(t_{w'})$, either $x \in \mathcal{V}(t_w)$ or $x \in \mathcal{V}(t_{u'})$ for some $\pi'(t_u, t_{u'}) \in \text{Pre}$.

A Relational Specification $(\Omega, \phi, \text{A}, \Omega_{\text{OBS}})$ is *Data-driven* if every Relational Rule in ϕ is Data-driven □

Definition 2(2) rules out premisses of the form $e \Rightarrow_S \mathbf{plus}(n_1, n_2)$. An interpretation of this premiss, in a context in which n_1 and n_2 are not instantiated, would be as follows: if e evaluates to some number n then find two numbers n_1 and n_2 such that $n = \mathbf{plus}(n_1, n_2)$. This is clearly undesirable from an operational point of view because it involves reversing the evaluation of a sum operation into the values of its operands. Definition 2(4) guarantees that term evaluation does not change the syntactical structure of the conclusion of the Data-driven rules. The motivations are the same as for Definition 2(2), here applied also to the inputs of the conclusion. Similar restrictions on the form of Natural Semantics rules were defined in [AFZ88,AC90] to allow evaluation of TYPOL programs without using unification.

The motivation for the restrictions introduced by items 2(1), 2(3), and 2(5) is that they allow a formalisation of an operational interpretation of the rules that agrees with the informal interpretation presented at the beginning of this section. This operational interpretation defines a *flow of evaluation* that goes from the inputs of the conclusion of a rule, through the premisses (in any total order consistent with the relation $\preceq$), to the outputs of the conclusion. This interpretation was informally introduced by Berry in [Ber91a, page 48], and is formalised in our framework in Theorem 2.

We now formalise a notion of program evaluation.

Definition 3 (Program Evaluation) Let $\mathcal{S} = (\Omega, \phi, \text{A}, \Omega_{\text{OBS}})$ be a Data-driven Specification and $P_{\text{X}}(\Omega)$ be the set of structural program formulae over Ω. *Program evaluation* is the search for a satisfying substitution for a query $p \in P_{\text{X}}(\Omega)$ using $\mathcal{CS}^{\mathcal{S}}$. □

We now define the concepts of goal choice and expansion order for premisses in the context of the $\mathcal{CS}$ system. A *goal choice* in the $\mathcal{CS}$ system is a (total) function whose input is a goal stack and the output is a formulae called the *selected goal.* In logic programming, the term "computation rule" is often used for the term "goal choice", e.g., in [Llo87, pages 50,60]. An *expansion order* is a total order on the set of premisses of a Data-driven Rule that defines the order in which the premisses are added on the goal stack. In the definition of $\mathcal{CS}$ the goal choice is non-deterministic; thus, the particular expansion order is not relevant. However, the next definitions and theorems depend on the particular goal choice and expansion order defined below.

The *Standard Goal Choice* is the goal choice that always chooses the left most element of the goal stack. Any expansion rule that is consistent with the partial order $\preceq$ is called a $\preceq$-*expansion.* The $\mathcal{CS}$ system of a Data-driven Specification that uses Standard Goal Choice and $\preceq$-expansion is complete with respect to the Declarative Semantics of the Data-driven Specification.

We now prove two important theorems: the first formalises an operational interpretation for Data-driven Specifications; the second proves the aforementioned completeness result. Informally, the first theorem states that in a $\mathcal{CS}$ system of a Data-driven Specification that uses standard goal choice and $\preceq$-expansion, if a program formula p has a $\mathcal{CS}$-sequence with answer θ then the instantiation of the outputs of p by θ are ground terms.

Theorem 2 (Operational Interpretation)
Let $(\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Data-driven Relational Specification and $\mathcal{CS}$ be its Computational Semantics that uses Standard Goal Choice and $\preceq$-expansion. For all $p \in P_{\mathrm{X}}(\Omega)$, if there exists a $\mathcal{CS}$-sequence for p of the form $[\theta_0 : p] \overset{\mathrm{CS}}{\Rightarrow}^{*} [\theta : \varepsilon]$ then $p\theta \in P(\Omega)$, i.e., θ is a ground instantiation for p.

Proof This proof is by induction on the length of the $\mathcal{CS}$-sequence of p.

Base Case For a single transition, the selected rule is of the form $(\{\ \}, ce)$. All terms in the inputs of p are ground by the definition of $P_{\mathrm{X}}(\Omega)$. Therefore, every

term in $ce\theta$ is ground because $\theta = Unify(\chi^{A}(p\theta_0), ce)$ and every meta-variable of the output of ce must occur in the input of ce. Therefore, $ce\theta \in P(\Omega)$, and since $ce\theta = p\theta$ then $p\theta \in P(\Omega)$.

Inductive Step For a transition sequence of length $m > 1$ the rule selected in the first transition must be of the form $(\{f_1, \ldots, f_n\}, ce)$ such that:

$$[\theta_0 f] \overset{CS}{\Rightarrow} [\theta_0\theta_1 : f_1 \cdot \ldots \cdot f_n]$$

where $f_1, \ldots, f_n$ is in a total order consistent with $\preceq$. Now we prove by induction on $n \geq 1$ that if $[\theta_0\theta_1 : f_1 \cdot \ldots \cdot f_n] \overset{CS}{\Rightarrow}^{*} [\theta : \varepsilon]$ then $f_i\theta \in P(\Omega)$ for $i \in [n_+]$. We call this induction the *inner induction.*

From the definition of Data-driven Rules, every meta-variable of the inputs of f_1 must occur also in the inputs of ce; thus, $f_1\theta_0\theta_1 \in P_X(\Omega)$. Now, we apply the outer inductive hypothesis on the state $[\theta_0 : f_1\theta_1]$ and obtain the following transition:

$$[\theta_0 : f_1\theta_1] \overset{CS}{\Rightarrow}^{*} [\theta_0\theta_2 : \varepsilon]$$

and $f_1\theta_1\theta_2 \in P(\Omega)$. The substitution θ_1 agrees with θ_2 on the meta-variables of f_1 in the following sense: for all $x \in \mathcal{V}(f_1)$, either $\theta_1(x) = x$ or $\theta_1(x) = \theta_2(x)$. Therefore,

$$[\theta_0\theta_1 : f_1] \overset{CS}{\Rightarrow}^{*} [\theta_0\theta_1\theta_2 : \varepsilon]$$

is also a transition sequence in CS. Therefore, we can assemble the following sequence:

$$[\theta_0\theta_1 : f_1 \cdot \ldots \cdot f_n] \overset{CS}{\Rightarrow}^{*} [\theta_0\theta_1\theta_2 : f_2 \cdot \ldots \cdot f_n]$$

which, together with the fact that $f_1\theta_1\theta_2 \in P(\Omega)$, proves the base case for the inner induction. From the definition of Data-driven Rules, every meta-variable of the inputs of f_2 must occur also either in the inputs of ce or in the outputs of f_1; thus, $f_2\theta_1\theta_2 \in P_X(\Omega)$. Therefore, we can apply the inner inductive hypothesis and obtain:

$$[\theta\theta_1\theta_2 : f_2 \cdot \ldots \cdot f_n] \overset{CS}{\Rightarrow}^{*} [\theta : \varepsilon]$$

Therefore, $f_i\theta \in P(\Omega)$, for every $i \in [n_+]$. Each meta-variable of the *ce* occurs either in the inputs of *ce*, in which case the instantiation by θ is ground because of the unification with p, or else it occurs in the output of some premiss, and in this case its instantiation by θ is ground by the proof of the inner induction. Therefore, $ce\theta \in P(\Omega)$ and $p\theta \in P(\Omega)$. □

The next theorem states the completeness of the Computation Semantics of a Data-driven Specification. This theorem uses the concept of a structural substitution. A substitution θ is *structural* if and only if every term in ran θ is structural. Informally, the Completeness Theorem states that if there is ground structural substitution $\overline{\theta}$ which is a satisfying substitution for $p \in P_X(\Omega)$ then $\mathcal{CS}$ finds a satisfying substitution θ for p such that $\Psi(p\overline{\theta}) = \Psi(p\theta)$.

In the following proposition we show that a completeness theorem stated in terms of an arbitrary ground substitution for p is no more general than Theorem 3 which is stated in terms of a ground structural substitution. The proposition states that any ground substitution can be replaced by a ground structural substitution in the context of an evaluation function.

Proposition 1 If $\overline{\theta}$ is a ground substitution for $t \in T_X(\Sigma)$ and $t\overline{\theta} \in \text{dom } \psi^A$ then there exists a ground structural substitution $\overline{\theta}'$ for t such that $t\overline{\theta}' \in \text{dom } \psi^A$ and $\psi^A(t\overline{\theta}) = \psi^A(t\overline{\theta}')$.

Proof (Sketch) Let $\overline{\theta}'$ be the same as $\overline{\theta}$ except that for all $x \in \text{dom } \overline{\theta}/\mathcal{V}(t)$, $\overline{\theta}'(x) = \psi^A(\overline{\theta}(x))$. Notice that $\overline{\theta}(x) \in \text{dom } \psi^A$ otherwise $t\overline{\theta} \notin \text{dom } \psi^A$. Since all non-structural terms in ran $\overline{\theta}/\mathcal{V}(t)$ were replaced by the result of their evaluation, $\overline{\theta}'$ is structural on ran $\overline{\theta}/\mathcal{V}(t)$. Furthermore, it is clear $\psi^A(t\overline{\theta}) = \psi^A(t\overline{\theta}')$. □

The proof of Theorem 3 uses the following lemmas about the evaluation of structural terms and structural formulae, which are stated without proof. The first lemma is a trivial consequence of the fact that each function symbol in a structural term has the term algebra interpretation in A.

Lemma 2 For all ground structural terms $\overline{t} \in T(\Sigma)_s$,
if $\overline{t} \in \text{dom } \psi^{\text{A}}$ then $\psi^{\text{A}}(\overline{t}) = \overline{t}$. □

Using Lemma 2, it is trivial to prove the following lemma about simplification of structural terms.

Lemma 3 For all structural term $t \in T_{\text{X}}(\Sigma)_s$ and ground structural substitution $\overline{\theta}$ if $t\overline{\theta} \in \text{dom } \chi^{\text{A}}$ then $\chi^{\text{A}}(t\overline{\theta}) = \chi^{\text{A}}(t)\overline{\theta}$. □

Using the fact that χ^{A} is the same as ψ^{A} for ground terms and generalising the above lemmas to program formulae, we obtain the following lemma.

Lemma 4 For all structural program formulae $p \in P_{\text{X}}(\Omega)$ and ground structural substitution $\overline{\theta}$, if $p\overline{\theta} \in \text{dom } \Psi^{\text{A}}$ then $\Psi^{\text{A}}(p\overline{\theta}) = \chi^{\text{A}}(p)\overline{\theta}$. □

We now state and prove the Completeness Theorem.

Theorem 3 (Completeness of the Computational Semantics)
Let $(\Omega, \phi, \text{A}, \Omega_{\text{OBS}})$ be a Data-driven Relational Specification and $\mathcal{CS}$ be its Computational Semantics that uses Standard Goal Choice and $\preceq$-expansion. For all $p \in P_{\text{X}}(\Omega)$, if there exists a ground structural substitution $\overline{\theta}$ for p such that $p\overline{\theta} \in \text{dom } \Psi^{\text{A}}$ and $\Psi^{\text{A}}(p\overline{\theta}) \in \pi^{\text{A}}$ then there exists a $\mathcal{CS}$-sequence of p of the following form $[\theta_0 : p] \overset{\text{CS}}{\Rightarrow}^* [\theta : \varepsilon]$ such that $\Psi^{\text{A}}(p\theta) = \Psi^{\text{A}}(p\overline{\theta})$.

Proof If $p\overline{\theta} \in \text{dom } \Psi^{\text{A}}$ and $\Psi^{\text{A}}(p\overline{\theta}) \in \pi^{\text{A}}$ then, from Proposition 2.3, there is a ϕ^{A}-tree of $\Psi^{\text{A}}(p\overline{\theta})$ whose root is an instance of the conclusion of a rule $\mathcal{R} = (\{f_1, \ldots, f_n\}, ce)$, $n \geq 0$, in ϕ, by a ground structural substitution $\overline{\theta}'$ such that:

$$\frac{\text{PT}(\Psi^{\text{A}}(f_1\overline{\theta}')) \ldots \text{PT}(\Psi^{\text{A}}(f_n\overline{\theta}'))}{\Psi^{\text{A}}(ce\overline{\theta}')}$$

and because every Relational Rule in ϕ is Data-driven, from Definition 2(4) every term in ce is structural. From Lemma 2, generalised to the evaluation of formulae,

it follows that $\Psi^{\mathrm{A}}(ce\overline{\theta}') = ce\overline{\theta}'$. The proof of Theorem 3 is by induction on the height of the ϕ^{A}-tree and builds a $\mathcal{CS}$-sequence for p using this ϕ^{A}-tree.

Base Case If the ϕ^{A}-tree has height 1 the rule $\mathcal{R}$ is of the form $(\{\,\}, ce)$ and $ce\overline{\theta}' = \Psi^{\mathrm{A}}(p\overline{\theta})$. From Lemma 4, $\Psi^{\mathrm{A}}(p\overline{\theta}) = \chi^{\mathrm{A}}(p)\overline{\theta}$. Therefore, $\overline{\theta}'\overline{\theta}$ is a unifier for ce and $\chi^{\mathrm{A}}(p)$. Let θ be the m.g.u. of ce and $\chi^{\mathrm{A}}(p)$, therefore the single transition $[\theta_0 : p] \overset{\mathrm{CS}}{\Rightarrow} [\theta : \varepsilon]$ is a $\mathcal{CS}$-sequence for p.

It remains to prove that $\Psi^{\mathrm{A}}(p\theta) = \Psi^{\mathrm{A}}(p\overline{\theta})$. Since θ is the m.g.u. of ce and $\chi^{\mathrm{A}}(p)$ then there exists a substitution θ' such that $\overline{\theta}'\overline{\theta} = \theta\theta'$; in particular, $\Psi^{\mathrm{A}}(p\overline{\theta}'\overline{\theta}) = \Psi^{\mathrm{A}}(p\theta\theta')$. Since the meta-variable in ce are fresh $\overline{\theta}'$ does not change p and we can write $\Psi^{\mathrm{A}}(p\overline{\theta}'\overline{\theta}) = \Psi^{\mathrm{A}}(p\overline{\theta})$; thus, $\Psi^{\mathrm{A}}(p\overline{\theta}) = \Psi^{\mathrm{A}}(p\theta\theta')$, and since from Theorem 2 $p\theta$ is ground we have $\Psi^{\mathrm{A}}(p\theta\theta') = \Psi^{\mathrm{A}}(p\theta)$. Therefore, $\Psi^{\mathrm{A}}(p\overline{\theta}) = \Psi^{\mathrm{A}}(p\theta)$, finishing the proof of the base case.

Inductive Step If the ϕ^{A}-tree has height $h > 1$, then the rule $\mathcal{R}$ is of the form $(\{f_1, \ldots, f_n\}, ce)$, for $n \geq 1$, and $ce\overline{\theta}' = \Psi^{\mathrm{A}}(p\overline{\theta})$. From Lemma 4, $\Psi^{\mathrm{A}}(p\overline{\theta}) = \chi^{\mathrm{A}}(p)\overline{\theta}$. Therefore, $\overline{\theta}'\overline{\theta}$ is a unifier for ce and $\chi^{\mathrm{A}}(p)$. Let θ_1 be the m.g.u. of ce and $\chi^{\mathrm{A}}(p)$, and therefore the following transition is a transition in $\mathcal{CS}$:

$$[\theta_0 : p] \overset{\mathrm{CS}}{\Rightarrow} [\theta_0\theta_1 : f_1 \cdot \ldots \cdot f_n]$$

Now, we prove by induction on $n \geq 1$ that:

$$[\theta_0\theta_1 : f_1 \cdot \ldots \cdot f_n] \overset{\mathrm{CS}}{\Rightarrow}^{*} [\theta_0\theta_1 \ldots \theta_n : \varepsilon]$$

We call this induction the *inner induction.* For the base case of this proof, recall that the premisses of $\mathcal{R}$ are added to the goal stack in a total order consistent with the $\preceq$ relation. Since $\mathcal{R}$ is Data-driven all meta-variables in the inputs of f_1 appear in the inputs of ce. Therefore, $f_1\theta_1 \in P_{\mathrm{X}}(\Omega)$. The substitution θ_1 agrees with $\overline{\theta}'$ on the variables of f_1 in the sense that for all $x \in \mathcal{V}(f_1)$, either $\theta_1(x) = x$ or $\theta_1(x) = \overline{\theta}'(x)$. Therefore, $f_1\theta_1\overline{\theta}' = f_1\overline{\theta}'$ and the ϕ^{A}-tree for $\Psi^{\mathrm{A}}(f_1\theta_1\overline{\theta}')$ is $\mathrm{PT}(\Psi^{\mathrm{A}}(f_1\overline{\theta}'))$. Therefore, we can apply the inductive hypothesis of the outer induction to obtain the following transition sequence:

$$[\theta_0 : f_1\theta_1] \overset{\mathrm{CS}}{\Rightarrow}^{*} [\theta_2 : \varepsilon]$$

which is a $\mathcal{CS}$-sequence of $f_1\theta_1$, with answer θ_2. Moreover, since θ_1 agrees with the substitution θ_2 on the variables of f_1 then $[\theta_0\theta_1 : f_1] \overset{\mathrm{CS}}{\Rightarrow}^* [\theta_0\theta_1\theta_2 : \varepsilon]$ is also a transition sequence in $\mathcal{CS}$. Therefore, we obtain the following transition sequence:

$$[\theta_0 : p] \overset{\mathrm{CS}}{\Rightarrow} [\theta_0\theta_1 : f_1 \cdot \ldots \cdot f_n] \overset{\mathrm{CS}}{\Rightarrow}^* [\theta_0\theta_1\theta_2 : f_2 \cdot \ldots \cdot f_n]$$

It follows from Theorem 2 that $f_1\theta_0\theta_1\theta_2 \in P(\Omega)$ and from the inductive hypothesis $\Psi^{\mathrm{A}}(f_1\theta_0\theta_1\theta_2) = \Psi^{\mathrm{A}}(f_1\overline{\theta}')$; thus θ_2 agrees with $\overline{\theta}'$ on the variables of f_1. Therefore, because each meta-variable in the inputs of f_2 also occurs either in the inputs of ce or in the outputs of f_1 we have that $f_2\theta_0\theta_1\theta_2 \in P_{\mathrm{X}}(\Omega)$. Therefore, we can apply the inner inductive hypothesis and obtain the following transition:

$$[\theta_0\theta_1\theta_2 : f_2 \cdot \ldots \cdot f_n] \overset{\mathrm{CS}}{\Rightarrow}^* [\theta_0\theta_1 \ldots \theta_n : \varepsilon]$$

By appending this sequence to the previous one we obtain:

$$[\theta_0 : p] \overset{\mathrm{CS}}{\Rightarrow}^* [\theta_0\theta_1 \ldots \theta_n : \varepsilon]$$

It remains to show that $\Psi^{\mathrm{A}}(p\theta_0\theta_1 \ldots \theta_n) = \Psi^{\mathrm{A}}(p\overline{\theta})$. First, notice that each θ_i, for $i \in [n_+]$, agrees with $\overline{\theta}'$ on the meta-variables of f_i; in particular they agree on those meta-variables of f_i that occur in ce. The substitution θ_1 also agrees with $\overline{\theta}'$ on the meta-variables of the input terms of ce. Because $\mathcal{R}$ is Data-driven, the meta-variables in the outputs of ce are either in the output of some f_i or in the inputs of ce; thus $\theta_0\theta_1 \ldots \theta_n$ agrees with $\overline{\theta}'$ on all meta-variables of ce. Therefore, the following holds:

$$ce\theta_0\theta_1 \ldots \theta_n = ce\overline{\theta}'$$

since $\Psi^{\mathrm{A}}(ce\overline{\theta}') = \Psi^{\mathrm{A}}(p\overline{\theta})$ then $\Psi^{\mathrm{A}}(ce\theta_0\theta_1 \ldots \theta_n) = \Psi^{\mathrm{A}}(p\overline{\theta})$; because θ_1 is the m.g.u. of ce and $\chi^{\mathrm{A}}(p)$ we obtain $\Psi^{\mathrm{A}}(ce\theta_0\theta_1 \ldots \theta_n) = \Psi^{\mathrm{A}}(\chi^{\mathrm{A}}(p)\theta_0\theta_1 \ldots \theta_n)$. Applying Lemma 4 we have that $\Psi^{\mathrm{A}}(ce\theta_0\theta_1 \ldots \theta_n) = \Psi^{\mathrm{A}}(p\theta_0\theta_1 \ldots \theta_n)$ and therefore $\Psi^{\mathrm{A}}(p\theta_0\theta_1 \ldots \theta_n) = \Psi^{\mathrm{A}}(p\overline{\theta})$.

This finishes the proof of the Completeness theorem. □

The above theorem assures that the Computational Semantics of a Data-driven Specification with Standard Goal Choice and $\preceq$-expansion is complete for program evaluation. Together with Theorem 2, this completeness means that if a program formula $\pi(\overline{t_w}, t_{w'})$ has a $\mathcal{CS}$-sequence with answer θ then $t_{w'}\theta$ is ground, i.e., $t_{w'}\theta$ is the result of the evaluation of the program $\overline{t_w}$.

Let us now continue with the development of a notion of evaluation step. We argue that the operational interpretation given by the Computational Semantics of a Data-driven Specification is intuitive. Furthermore, it is sound and complete with respect to the Declarative Semantics and can be used as the underlying theory of a system for prototyping of programming languages as the CENTAUR system or the Animator Generator. However, the transition of this Computational Semantics is non-deterministic in general, and therefore unsuitable to be used as the evaluation step in debugging.

The theory of debugger specification developed in Chapter 6 only addresses debuggers for deterministic programming languages. Therefore, it is reasonable to restrict the possible Relational Specifications to a class in which only deterministic programming languages can be expressed. We then prove that the Computational Semantics of such a specification is monogenic; thus, this Computational Semantics yields a deterministic notion of evaluation step which is suitable for debugging.

Definition 4 (Deterministic Specification)
A Data-driven Specification $(\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ is *deterministic* if there is no pair of rules (Pre, ce) and (Pre$'$, ce') in ϕ such that, if $ce = \pi(t_w, t_{w'})$ and $ce' = \pi(t'_w, t'_{w'})$, then t_w and t'_w unify. □

The following theorem states that the Computational Semantics of a Deterministic Relational Specification is monogenic. Therefore, this system has three desirable properties: it may be given a deterministic implementation; it yields

a deterministic and intuitive notion of evaluation step; and it is complete with respect to the Declarative Semantics of the specification.

Theorem 4 (Monogenicity)
Let $(\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Deterministic Relational Specification. The Computational Semantics of this specification that uses the Standard Goal Choice and $\preceq$-expansion of goals is monogenic.

Proof (Sketch) It is clearly monogenic because the standard goal choice and the $\preceq$-expansions are deterministic, and there is at most one rule in ϕ that can be selected at each state of the Computational Semantics. □

To define the concept of an evaluation step we must solve one last problem. The above definitions and theorems were stated with respect to any $\preceq$-expansion, i.e., premisses are added to the goal stack on any total order consistent with $\preceq$. We argue that any choice for this total order will give an intuitive notion of step.

However, in an application of the $\mathcal{CS}$ system, e.g., in the specification of a debugger, it is necessary to use one particular total order. Therefore, we have to make the choice of total order an explicit parameter of $\mathcal{CS}$. Let $<$ be a total order consistent with $\preceq$, we write $\mathcal{CS}_<$ to denote the $\mathcal{CS}$ system that uses Standard Goal Choice and add the premisses on the goal stack according to $<$. We write $\stackrel{\mathrm{CS}}{\Rightarrow}_<$ for the transition relation of $\mathcal{CS}_<$.

We are now ready to define an evaluation step based on a Relational Specification.

Definition 5 (Evaluation Step) Let $(\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Deterministic Relational Specification and $<$ a total order on $F_{\mathrm{X}}(\Omega)$ consistent with $\preceq$. An *evaluation step* of a program formula $p \in P_{\mathrm{X}}(\Omega)$ is a transition of the $\mathcal{CS}_<$ system during the evaluation of p. □

Although the definition of an evaluation step is parametric on a definition of $<$, we will fix a particular total order to be used in subsequent examples. Since the

left-to-right order in which the premisses are written in the Relational Rules of all our examples is consistent with $\preceq$, we will use this order to add the premisses on the goal stack. In the rest of this thesis, we use the term *standard* to refer to a $\mathcal{CS}$ system with this total order and Standard Goal Choice.

Let us now introduce a notational simplification. Since Standard Goal Choice treats the goal stack as a list of formula we will write goal stacks in the usual list notation using the "::" as the list constructor and ε as the empty list. Therefore, in a goal stack $f :: \alpha$, the formula f is the selected goal and α is the rest of the stack.

Let us see how the transition relation of the standard $\mathcal{CS}$ behaves when compared to the intuitive notion of evaluation step discussed in the beginning of this section. For this, we consider a hypothetical evaluation of an expression $e_1 + e_2$ in $\mathcal{CS}^{Exp}$, i.e., we consider the transitions of $\mathcal{CS}^{Exp}$ for a query $e_1 + e_2 \Rightarrow_S n$. In the following examples we ignore the renaming of the meta-variables and elide some formulae from the goal stack to simplify the presentation. The initial transition of this evaluation uses rule 2.1(2)′ (page 133), and is as follows:

$$[\theta_0 : e_1 + e_2 \Rightarrow_S n] \overset{\mathcal{CS}}{\Rightarrow} [\theta_0\theta_1 : e_1 \Rightarrow_S n_1 :: e_2 \Rightarrow_S n_2 :: \mathbf{num}(\mathbf{plus}(n_1, n_2)) \Rightarrow_S v]$$

The current state after this transition indicates that the evaluation of $e_1 + e_2$ starts with the evaluation of e_1. Using enough $\mathcal{CS}^{Exp}$ transitions until the evaluation of e_1 is completed we obtain the following sequence:

$$[\theta_0 : e_1 + e_2 \Rightarrow_S n] \overset{\mathcal{CS}}{\Rightarrow} [\theta_0\theta_1 : e_1 \Rightarrow_S n_1 :: \ldots] \overset{\mathcal{CS}^*}{\Rightarrow} [\theta_0\theta_1\theta_2 : e_2 \Rightarrow_S n_2 :: \ldots]$$

such that $\theta_0\theta_1 \ldots \theta_2(n_1)$ is the result of the evaluation of e_1. The full evaluation of the expression is represented by the following transition sequence:

$$[\theta_0 : e_1 + e_2 \Rightarrow_S n] \overset{\mathcal{CS}}{\Rightarrow} [\theta_0\theta_1 : e_1 \Rightarrow_S n_1 :: \ldots] \overset{\mathcal{CS}^*}{\Rightarrow} [\theta_0\theta_1\theta_2 : e_2 \Rightarrow_S n_2 :: \ldots] \overset{\mathcal{CS}^*}{\Rightarrow}$$
$$[\theta_0\theta_1\theta_2\theta_3 : \mathbf{num}(\mathbf{plus}(n_1, n_2)) \Rightarrow v] \overset{\mathcal{CS}^*}{\Rightarrow} [\theta_0\theta_1 \ldots \theta_n : \varepsilon]$$

In this last state $\theta_0\theta_1 \ldots \theta_n(n)$ is the result of the evaluation of the expression $e_1 + e_2$. We can interpret this $\mathcal{CS}^{Exp}$-sequence by saying that the evaluation of

$e_1 + e_2$ is composed of the steps of the evaluation of e_1, followed by the steps of the evaluation of e_2, followed by the steps necessary to perform the sum operation.

Given the choice of total order on the premisses discussed above, the operational interpretation of the above transitions agrees with the informal interpretation discussed in the beginning of this section.

The definition of an evaluation step in Definition 5, together with the definition of program evaluation in Definition 3, achieves the goals of this chapter. However, the class of Deterministic Relational Specifications excludes many Relational Specifications of deterministic programming languages whose Computational Semantics also yield a natural notion of step. For instance, suppose that we add conditional expressions to the *Exp* language of Example 2.1 by the following grammar rule:

$$exp ::= \textbf{if } exp \textbf{ then } exp \textbf{ else } exp$$

Two natural Relational Rules to define the semantics of the conditional expressions could be given by:

$$\frac{e_1 \Rightarrow_S \textbf{true} \qquad e_2 \Rightarrow_S v}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow_S v} \qquad (C_1)$$

$$\frac{e_1 \Rightarrow_S \textbf{false} \qquad e_3 \Rightarrow_S v}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow_S v} \qquad (C_2)$$

These rules are not deterministic in the sense of Definition 4 since their conclusions unify. However, there exists a natural operational interpretation for the above rules: the evaluation of a conditional expression **if** e_1 **then** e_2 **else** e_3 consists in first evaluating e_1 and then checking its result; if it is true we use rule (C_1), and evaluate e_2 to obtain the result of the conditional expression; otherwise, we use rule (C_2), and evaluate e_3 to obtain the result of the conditional. This operational interpretation does not fit in our previous discussions because it somehow suspends the choice between the rules until e_1 is entirely evaluated

Nevertheless, there is at most one $\mathcal{CS}^{Exp}$-sequence for each conditional expression. The specification *Exp* extended with conditional expressions is still a

deterministic language, although we do not consider its Relational Specification as being deterministic. These facts suggest that we could search for an alternative set of Relational Rules with the following properties: it is semantically equivalent to the above set; it has the above operational interpretation; and is deterministic in the sense of Definition 4. The following rules have these properties:

$$\frac{e_1 \Rightarrow_S v'' \qquad branch(v'', e_2, e_3, v')}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow_S v'} \qquad (C'_1)$$

$$\frac{e_2 \Rightarrow_S v}{branch(\textbf{true}, e_2, e_3, v)} \qquad (C'_2)$$

$$\frac{e_3 \Rightarrow_S v}{branch(\textbf{false}, e_2, e_3, v)} \qquad (C'_3)$$

where $branch : val \times exp \times exp \leftrightarrow val$ is a fresh relation name. The operational interpretation of the evaluation of conditional expressions using these rules agrees with the interpretation we gave to the rules (C_1) and (C_2): the rules (C'_1) is the only one that can be selected to start the evaluation of a conditional; then e_1 is evaluated using this rule and its result is used to select between (C'_2) and (C'_3).

Therefore, if we can define an automatic transformation from rules in the form of the first set into rules in the form of the second set we would achieve two results. First, we would prove that the use of non-deterministic rules like the first set of conditional rules does not add any expressiveness to the Relational Specifications. Second, that rules as the first set could be converted into Deterministic Rules with the same operational interpretation.

The transformation illustrated above can be automatically performed. The first step towards the definition of this transformation is to characterise a class of semantics that contains the rules of the form of the first set of conditional rules. This is done in the following definition.

Definition 6 (Dynamically-deterministic Specification)
A Data-driven Specification $(\Omega, \phi, A, \Omega_{OBS})$ is *Dynamically-deterministic* if and only if for each pair of rules (Pre, ce) and (Pre$'$, ce') in ϕ, such that $ce = \pi(t_w, t_{w'})$ and $ce' = \pi(t'_w, t'_{w'})$, then either:

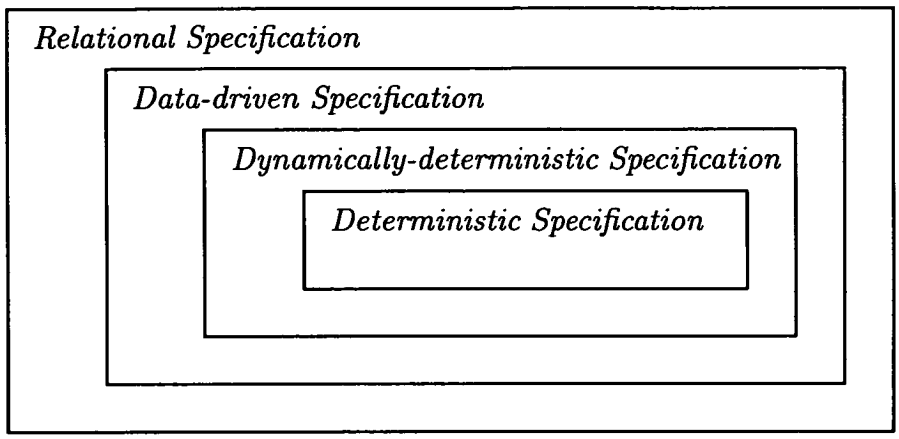

Figure 2: (More) Inclusions of Classes of Relational Specifications

- t_w and t'_w do not unify, or
- $ce = ce'$, and there exits $\pi'(t_u, t_{u'}) \in \text{Pre}$ and $\pi'(t'_u, t'_{u'}) \in \text{Pre}'$, such that $t_u = t'_u$, $t_{u'}$ and $t'_{u'}$ do not unify, and $\overline{s}(\pi(t_u, t_{u'})) = \overline{s}(\pi(t'_u, t'_{u'}))$.

where $\overline{s}(f)$, for $f \in \text{Pre}$ (respectively Pre'), denotes the premisses of Pre (respectively Pre') that are strictly smaller than $f$according to the (formulae) dependency relation $\preceq$ (page 137). □

Figure 2 extends the diagram of Figure 1 with Dynamically-deterministic Specifications. We can relax the above definition by requiring ce and ce' to be α-convertible; the same for t_u and t'_u, and each pair of corresponding formulae in $\overline{s}(\pi(t_u, t_{u'}))$ and $\overline{s}(\pi(t'_u, t'_{u'}))$. The more general definition is presented in [dS90]. However, this generalisation does not add any expressiveness and makes the definition more complex. Therefore, we favoured the simpler approach in this presentation.

In any Dynamically-deterministic Specification, whenever two rules have conclusions whose inputs unify, then there exists an initial sequence (ordered by a total order consistent with $\preceq$) of premisses in both rules equal, and there is a pair of premisses in each rule whose inputs are equal and whose outputs do not

unify. This pair of premisses is called a *distinction pair.*

A distinction pair can be used to guide the transformation of the rules as described by the above example. In that case $e_1 \Rightarrow_S$ **true** and $e_1 \Rightarrow_S$ **false** is the distinction pair; this pair is factorised from the two original rules into the single rule (C_1'). In [dS90] we presented an algorithm (called FAIR) that transforms Dynamically-deterministic Specifications into Deterministic Specifications. FAIR performs the transformation that was described in the above example. We argue that the rules generated by factorisation have the intuitive operational interpretation that we discussed for the original set of conditional rules.

Moreover, in Chapter 6 we will discuss mechanisms in which we can hide some evaluation steps in a debugger. Using such a mechanism we could hide the steps due to the evaluation of the *branch* premiss of rule (C_1'), since this premiss does not occur in the original set of rules. Consequently, we can use a Dynamically-deterministic Specification in the definition of some language, and then transform the specification into a Deterministic one for use in the specification of a debugger. Crucially, the transformed specification has the same meta-semantics and the same notion of evaluation step as the original.

The class of dynamically-deterministic rules is fairly large. The semantic aspects of most sequential programming languages can be expressed in this class of specifications. In particular, the dynamic semantics of Standard ML [HMT89] can be expressed by a Dynamically-deterministic Specification. Therefore, the theory of debugger design addresses this (large) class of Relational Specifications.

5.5 Summary and Conclusions

In this chapter we defined an operational interpretation of a Relational Specification by presenting its Computational Semantics. Theorem 1 established the soundness of the Computational Semantics with respect to the Declarative Semantics of Definition 2.3.

We then studied the problem of defining an intuitive notion of evaluation step

using the Computational Semantics. We demonstrated by examples that not all Relational Specifications yield such a notion. We characterised the class of Data-driven Relational Specifications and showed that specifications in this class yield an intuitive notion of evaluation step. The Computational Semantics of any Data-driven Specification can be given a formal operational interpretation, expressed in Theorem 2. Furthermore, such a Computational Semantics is complete as proved in Theorem 3.

The restrictions that led to the class of Data-driven Specifications are similar to the restrictions imposed by Berry on a Structural Operational Semantics to define a notion of evaluation step [Ber91a, pages 70,91]. This similarity suggests that these restrictions are not ad hoc, but necessary whenever we want to define a notion of evaluation step based on a Relational Semantics. An interesting problem for future research is to compare the notions of evaluation step of [Ber91a] and of this thesis. This comparison could include a study of the approach to the evaluation of data-driven DCP in [DM85].

Another interesting problem is to investigate whether there exists a larger class of Relational Specifications whose Computational Semantics is complete. A related problem would be to study whether the transformation of rule 2.1(2) into rule 2.1(2)$'$ given in page 133 can be done automatically. An affirmative answer to this problem would implicitly enlarge the class of specifications with complete Computational Semantics.

Chapter 6

Formal Specification of Debuggers

In this chapter we study the formal specifications of debuggers. We start by presenting an abstract characterisation of a debugger using the notions of *program evaluation* and *evaluation step* developed in Chapter 5. A debugger, according to this characterisation, evaluates programs using the Computational Semantics of a Deterministic Relational Specification.

We then study two common issues in the specification of most concrete debuggers: the granularity of the debugging step and the reference to unique program positions. Since we presented an abstract definition of debuggers, we can address these issues at the level of an arbitrary debugger. A clear advantage of this approach is that the solutions we propose for the design issues can be used in the specification of any concrete debugger that conforms with our abstract definition.

Finally, we define a notation to assist in the specification of concrete debuggers. This notation expresses a useful set of debugging operations for any programming language that can be given a Deterministic Relational Specification. We illustrate the use of this notation in Section 6.6. An interesting problem for future investigation is to provide an implementation for the specification notation. This would provide facilities for high level prototyping of debuggers.

6.1 Introduction

In this section we compare our approach to some other approaches to debugging based on formal semantics that we find in the literature. The CENTAUR system [CIK89] has some debugging facilities for the TYPOL language [Des88]. When a TYPOL specification of a programming language is being interpreted, these facilities can be used as an (indirect) debugger for programs written in the programming language.

TYPOL debuggers are LISP programs that interact with the Mu-PROLOG program generated by the TYPOL compiler. In the original description of the debugging facilities for TYPOL in [Des88], there is no mention of special features in LISP to assist in the specification of debuggers. It is suggested by the authors of [Des88] that it would be possible to use Esterel [BCG86] to specify the debuggers and then produce the LISP code using the Esterel compiler. However, Esterel does not have specific features for the specification of debuggers in terms of the programming language semantics;in this particular sense, we say that CENTAUR does not provide high-level facilities for the specification of debuggers.

The PSG system [BS86] generates an interpreter, with debugging facilities, from a denotational semantics of the programming language [BMS87]. The interpreter is generated by translating the denotational semantics into an executable functional program. Before the evaluation of any sub-phrase of a program, the interpreter calls a debugger coroutine; this receives debugging commands from the user, updates the current state of the evaluation, checks for break-points, and so on. An evaluation step in the debugger is a function call to the definition of the program phrase; thus, it cannot be changed unless the whole system is redefined to incorporate another notion of step. Some debugging operations are generated automatically; more sophisticated operations must be encoded together with the semantics of the programming language. The language used to define these debugging operations is a simple functional language with no special features for debugger specification; thus PSG offers little support for the

definition of advanced debugging operations.

Our approach to debugger specification allows the definition of debugging steps of variable size, or *granularity*, based on the notion of evaluation step defined in Chapter 5. Moreover, we shall define a specification notation with features dedicated to the definition of debuggers at a high level of abstraction.

The Animator Generator [Ber91a] is a system that generates a programming language animator from a Structural Operational Semantics. An animator is an interpreter for the programming language with a sophisticated graphical interface used to display the intermediate steps of the evaluation of the program. The animation process is interactive, allowing the user to inspect the values of variables, single step backwards and forwards, and so on. An animation step is a function between partial proof trees of a program formula. Its definition is more complex and difficult to reason about than our definition of an evaluation step, because animation has other needs that are not important for the approach of this thesis, e.g., displaying and highlighting of (parts of) the proof tree of the program. Although it is possible to display different views of the program animation, the set of debugging operations in an animator is fixed, that is, the Animator Generator does not provide the means for specification of new debugging operations.

As mentioned above, our definition of evaluation step is simpler than an animation step, being easier to reason about. This property will be essential in proofs of debugger correctness, which are addressed in Chapter 7. Moreover, our approach allows the specification of a wide range of debugging operations, including the commands of the Animator Generator.

In [KHC91], Kishon et al. define Monitor Semantics as a system for the prototyping of program monitors based on a continuation-passing denotational semantics of the programming language. Monitor functions are specified as annotations to the original denotational semantics of the programming language. There exists a proof that the inclusion of a monitor specification does not change the semantics of the language. In [KHC91] there is no definition of a notion of an evaluation step; thus it is not clear how to specify a single stepping operation,

and nor it is clear how such an operation would behave.

The major difference between our approach and the four approaches discussed above is that while they aim at an implementation of a debugger generator we aim at the definition of a theory of debugger specification. This theory is the first, essential step towards a notion of *debugger correctness*, which is a major goal of this thesis. The definitions in this chapter set up a suitable framework in which we can study the problem of debugger correctness in Chapter 7.

6.2 Definitional Preliminaries

The following definitions are based on the definitions given in Section 4.2.

A *labelled transition system* is a triple $(\Gamma, \mathrm{L}, \rightarrow)$ where Γ is a set of states, L is a set of *labels*, and $\rightarrow \subseteq \Gamma \times \mathrm{L} \times \Gamma$ is the *labelled transition relation*. For $(st, l, st') \in \Gamma \times \mathrm{L} \times \Gamma$, the fact that $(st, l, st') \in \rightarrow$ is written $st \xrightarrow{l} st'$.

The generalisation of transitive-reflexive closure to a labelled transition relation is defined as follows. Let $sl \in \mathrm{L}^*$ be a sequence of labels $\langle l_1, \ldots, l_n \rangle$ for $n \geq 0$. If $n = 0$ then $st \xrightarrow{\langle\rangle}{}^* st$. If $n > 0$ then $st \xrightarrow{sl}{}^* st'$ if and only if there exists a state st'' such that $st \xrightarrow{l_1} st''$ and $st'' \xrightarrow{sl'}{}^* st'$, where $sl' = \langle l_2, \ldots, l_n \rangle$.

6.3 Interpreter-Debuggers

Let $\mathcal{S} = (\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Deterministic Relational Specification of some programming language and $p \in P_{\mathrm{X}}(\Omega)$ be an arbitrary structural program formula in $\mathcal{S}$. In this section we present a definition of an abstract debugger for p that evaluates the program formula using the Computational Semantics $\mathcal{CS}_<$ of $\mathcal{S}$ for a fixed but arbitrary total ordering $<$ consistent with $\preceq$, as discussed in Section 5.4 (page 146). This definition generalises trivially to a debugger for all program formulae in $P_{\mathrm{X}}(\Omega)$.

Let us recall some results and definitions from Chapter 5 related to the Com-

putational Semantics of Deterministic Specifications. Initially, $P_X(\Omega)$ denotes the set of *structural program formulae*; thus $p \in P_X(\Omega)$ is a structural formula whose input terms are ground. From Theorem 5.4, the Computational Semantics, with Standard Goal Choice and $\preceq$-expansion, of any Deterministic Specification is monogenic; thus, $\overset{CS}{\Rightarrow}_<$ of $\mathcal{CS}_<$ is a partial function, written $\overset{CS}{\Rightarrow}_< : \Gamma_{CS} \to \Gamma_{CS}$. However, at the risk of some confusion, we still use the (infix) relational notation $st \overset{CS}{\Rightarrow}_< st'$ instead of a functional notation, e.g., $\overset{CS}{\Rightarrow}_<(st) = st'$.

The choice of the total order $<$ is part of the design of a debugger in our framework. The following definitions are parametric on an arbitrary total order $<$ consistent with $\preceq$. To improve the readability, we will drop the subscript $<$ from $\mathcal{CS}$ and $\overset{CS}{\Rightarrow}$ hereafter. In the examples in this chapter and Chapter 7 we will use the standard $\mathcal{CS}$ system defined in Section 5.4 (page 147).

The reader will notice that some of the following definitions are more general than strictly necessary for the results we want in this thesis. In particular, we give definitions that could be easily generalised to the case where $\overset{CS}{\Rightarrow}$ is an actual relation. The main reason for this added generality is that it simplifies extensions to the theory, for instance to address debuggers for non-deterministic programming language, as discussed in Section 8.2.

We present the definition of an abstract debugger in two stages. We start with a very simple definition of a debugger with limited debugging functionality, and discuss its main problems and limitations. Motivated by this discussion, we present another characterisation that describes a larger class of debuggers, including most existing debuggers for deterministic programming languages. We start by defining a notion of evaluation history of programs; an evaluation history is sometimes called a *computation* in the literature.

Definition 1 (Evaluation History) The set of *evaluation histories* of p in $\mathcal{CS}$, written EH, is the subset of Γ^*_{CS} inductively defined as follows:

1. $\langle [\theta_0 : p] \rangle \in \mathrm{EH}$
2. If $\langle st_1, \ldots, st_n \rangle \in \mathrm{EH}$, and $st_n \overset{\mathrm{CS}}{\Rightarrow} st_{n+1}$,
   then $\langle st_1, \ldots, st_n, st_{n+1} \rangle \in \mathrm{EH}$ □

Notice that, because $\overset{\mathrm{CS}}{\Rightarrow}$ is a function, there exists at most one evaluation history $\langle st_1, \ldots, st_n \rangle$ for a given $n > 0$. For each history $\langle st_1, \ldots, st_n \rangle \in \mathrm{EH}$, $st_1 = \langle [\theta_0 : p] \rangle$ is the *initial* state, st_n is the *current* state. The terminology and definitions given for finite sequence in Section 2.2 also apply to evaluation histories. Moreover, if $h = \langle st_1, \ldots, st_n \rangle$ then $fst(h) = st_1$, $cur(h) = st_n$, and $rem(h) = \langle st_1, \ldots, st_{n-1} \rangle$.

Now, we define an extension of $\overset{\mathrm{CS}}{\Rightarrow}$ to a (unique) function $\overset{\mathrm{EH}}{\Longrightarrow}$: EH $\rightarrow$ EH consistent with $\overset{\mathrm{CS}}{\Rightarrow}$ as follows: for all $h \in \mathrm{EH}$, if $cur(h) \overset{\mathrm{CS}}{\Rightarrow} st$ then $h \overset{\mathrm{EH}}{\Longrightarrow} h \diamond st$. Clearly, if $\mathcal{CS}$ uses the function $\overset{\mathrm{EH}}{\Longrightarrow}$ instead of $\overset{\mathrm{CS}}{\Rightarrow}$ all the results of Chapter 5 remain valid. We use the term "$\mathcal{CS}$ with arrow $\overset{\mathrm{EH}}{\Longrightarrow}$" whenever we want to refer to the Computational Semantics of $\mathcal{S}$ that uses $\overset{\mathrm{EH}}{\Longrightarrow}$ as the transition relation.

The transition system $\mathcal{CS}$ with arrow $\overset{\mathrm{EH}}{\Longrightarrow}$ may be considered as a primitive debugger that only provides a single stepping operation between (consecutive) evaluation histories. This debugger is interactive only in the sense that the user can issue a step command and the debugger answers by advancing the evaluation history by one step, showing the new history to the user.

There are three major problems in considering $\mathcal{CS}$ as an abstract definition of a debugger. First, the *functionality* of this debugger is too limited. Second, the *granularity of the steps* is too small, i.e., little evaluation is really done at each evaluation step; this means that to reach a given state in the program evaluation we may need to issue many step commands. Third, an entire evaluation history contains a large amount of *information*; in general, we expect to be able to select only the parts of the evaluation history that we are interested in at each stage. These problems suggest we must develop a more powerful notion of debuggers so that we can address the three issues discussed above.

For this thesis, debuggers are interactive systems: the user inputs some debugging command and the debugger outputs the result of the command. Commands may be queries about the current program's evaluation state, or actions to either advance or go back in the evaluation. Results may be the required information about the evaluation, or the required action on the evaluation.

This view of a debugger may be formalised by characterising a debugger as a labelled transition system whose states contain an evaluation history as a component. The labels are pairs of a debugging command and its result, such that a transition of the form $ds \xrightarrow{(d,r)} ds'$ is interpreted as follows: in a debugging state ds and receiving a command d the debugger changes the state to ds' and outputs the result r.

We impose two restrictions on the transition systems that are allowed as debuggers in our approach. These restrictions have an intuitive motivation and are necessary so that we can prove properties about an abstract debugger. The first restriction ensures that debuggers are *robust* in the sense that all debugger commands evaluate to some result at every debugging state. The second restriction requires the existence of at least one debugging command that advances the evaluation. This is the *minimal functionality* we expect from a debugger, and it guarantees that whenever p succeeds in $\mathcal{CS}$ then there exists a sequence of debugging commands that advances p from its initial evaluation state to the successful state in $\mathcal{CS}$.

We believe that both restrictions arise naturally from a pragmatic point of view. Furthermore, in Chapter 7 we will prove results about debuggers that give a theoretical justification for the restrictions.

The above discussion leads to the definition of debuggers given below. A debugger, according to that definition, evaluates programs consistently with the $\mathcal{CS}$ system. Since the $\mathcal{CS}$ system is an interpreter of the programming language, Definition 2 characterises a class of debuggers that we call *Interpreter-debuggers.* For this definition, recall that $\mathcal{S} = (\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ is a Deterministic Relational Specification, $\Omega = (\mathrm{S}, \mathrm{F}, \Pi, \pi)$, Σ denotes (S, F), and $p \in P_{\mathrm{X}}(\Omega)$ is a program

formula in $\mathcal{S}$.

Definition 2 (Interpreter-Debugger) An *Interpreter-debugger* for p based on $\mathcal{S}$ and $\mathcal{CS}$ with arrow $\stackrel{\text{EH}}{\Longrightarrow}$ is a triple $(\Sigma_{\text{D}}, \Delta, \text{A}_{\text{D}})$ where:

- $\Sigma_{\text{D}} = (\text{S}_{\text{D}}, \text{F}_{\text{D}})$ and $\Sigma \sqsubset \Sigma_{\text{D}}$. The sorts *command* and *result* are in S_{D}, $T(\Sigma_{\text{D}})_{command}$ is the language of debugging *commands*, and $T(\Sigma_{\text{D}})_{result}$ is the language of *results* of debugging commands.
- $\text{A}_{\text{D}}/\Sigma = \text{A}$, i.e., the Σ_{D}-algebra of the debugger agrees with the Σ-algebra of the Relational Specification $\mathcal{S}$ on the programming language objects.
- Δ is a quintuple

$$(\Gamma_{\text{D}}, g, T(\Sigma_{\text{D}})_{command} \times T(\Sigma_{\text{D}})_{result}, \longrightarrow, \text{I})$$

  where $(\Gamma_{\text{D}}, T(\Sigma_{\text{D}})_{command} \times T(\Sigma_{\text{D}})_{result}, \longrightarrow)$ is a *monogenic labelled transition system* such that:

  - Γ_{D} is the set of *debugging states* and $g : \Gamma_{\text{D}} \rightarrow \text{EH}$ is a projection function from Γ_{D} into EH such that, for all $ds \in \Gamma_{\text{D}}$, $g(ds)$ is the evaluation history of p at ds.
  - $\longrightarrow \subseteq \Gamma_{\text{D}} \times (T(\Sigma_{\text{D}})_{command} \times T(\Sigma_{\text{D}})_{result}) \times \Gamma_{\text{D}}$ is a *labelled transition relation* such that the following *robustness requirement* holds: for all $dh \in \Gamma_{\text{D}}$ and $d \in T(\Sigma_{\text{D}})_{command}$ there exists $dh' \in \Gamma_{\text{D}}$ and $r \in T(\Sigma_{\text{D}})_{result}$ such that $dh \stackrel{(d,r)}{\longrightarrow} dh'$.
  - $\text{I} \in \Gamma_{\text{D}}$ the *initial debugging state*, such that $g(\text{I}) = \langle [\theta_0 : p] \rangle$.

such that the following *functionality requirement* holds: for all $ds \in \Gamma_{\text{D}}$ there exists $d \in T(\Sigma_{\text{D}})_{command}$, $r \in T(\Sigma_{\text{D}})_{result}$, and $ds' \in \Gamma_{\text{D}}$ such that $ds \stackrel{(d,r)}{\longrightarrow} ds'$ and $g(ds) \stackrel{\text{EH}}{\Longrightarrow}^{+} g(ds')$, where $\stackrel{\text{EH}}{\Longrightarrow}^{+}$ is the transitive closure of $\stackrel{\text{EH}}{\Longrightarrow}$ □

Let us now analyse some aspects of the above definition. The transition relation $\longrightarrow$ must be total, i.e., it must be defined for all debugging commands on all

debugging states. This requirement ensures that Interpreter-debuggers are *robust* as discussed above. It is always possible to add some failure states and results to a debugger to make it fulfill this requirement. Therefore, this requirement does not reduce the universe of debuggers that are addressed in this framework.

The functionality requirement ensures that every Interpreter-debugger is consistent with respect to the $\mathcal{CS}$ system regarding the evaluation of programs. In this sense, we can use any Interpreter-debugger as an alternative program evaluator. Furthermore, any correct evaluator with respect to the $\mathcal{CS}$ system can be easily transformed into an Interpreter-debugger with minimal functionality.

Another important aspect of Definition 2 is that for every debugging state $ds \in \Gamma_D$, $g(ds) \in$ EH. This implies that whenever

$$ds \xrightarrow{(d,r)} ds'$$

then either $g(ds) \overset{\text{EH}}{\Longrightarrow} g(ds')$ or, in the case that d is a backward evaluation command, $g(ds') \overset{\text{EH}}{\Longrightarrow} g(ds)$. Therefore, because the function $\overset{\text{EH}}{\Longrightarrow}$ is consistent with $\mathcal{CS}$, every Interpreter-debugger evaluates programs according to $\mathcal{CS}$ and uses the notion of evaluation step defined by the relation $\overset{\text{CS}}{\Rightarrow}$. This helps in the definition of debuggers that behave intuitively, since $\overset{\text{CS}}{\Rightarrow}$ is an intuitive notion of evaluation step, as discussed in Chapter 5.

The above fact has an important effect on the kinds of debuggers that are characterised by Definition 2. Since $g(ds) \in$ EH for all $ds \in \Gamma_D$, only evaluation states that are reachable from $[\theta_0 : p]$ using $\overset{\text{CS}}{\Rightarrow}$ are allowed in a debugging state. This implies that the debugging commands can only change the evaluation history component of a debugging state consistently with $\overset{\text{CS}}{\Rightarrow}$. This is the main reason Interpreter-debuggers are said to be based on a Relational Specification or "semantically driven".

There are various debugging commands in existing debuggers that allow the user to change the program data during debugging. In general, those commands produce states that are not reachable from I, in which case the debugger is not "semantically driven" and therefore excluded from our investigation.

6.4 Aspects of the Specification of Debuggers

In this section we study some issues related to the specification of Interpreter-debuggers. We use an example to introduce these issues. First, we define a semi-formal specification of a simple debugger. Then, we identify two problems in this specification and propose solutions for them for an arbitrary Interpreter-debugger. In the next section we shall see how this semi-formal specification can be made entirely formal using the debugger specification notation.

We will define an Interpreter-debugger for the language *Fun* called *Tiny*. The debugging language of *Tiny* is defined by the following BNF rule, which assume the definitions of *nat* and *var* of previous examples:

$$\begin{aligned} \textit{command} \ ::= \ & \textbf{reset} \mid \textbf{step}\ \textit{nat} \mid \textbf{run until}\ \textit{var} = \textit{nat} \mid \\ & \textbf{show}\ \textit{var} \mid \textbf{show pos} \mid \textbf{show res} \end{aligned}$$

We shall define *Tiny* based on the Relational Specification *Env* of *Fun* presented in Section 4.4. For this definition we replace rule 3.2(4) of the definition of *Env* by the following data-driven rule:

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad E \vdash e_2 \Rightarrow v_2 \qquad E \vdash \textbf{plus}(v_1, v_2) \Rightarrow v}{E \vdash e_1 + e_2 \Rightarrow v}$$

Tiny evaluates programs suing the Computational Semantics $\mathcal{CS}^{Env}$ with arrow $\overset{\text{EH}}{\Longrightarrow}_E$. *Tiny* will be defined for an arbitrary program formula $p \in P_{\mathbf{X}E}(\Omega^{Env})$. The set of states Γ^{Tiny} of *Tiny* is just the set EH of evaluation histories of p in $\mathcal{CS}^{Env}$ and g^{Tiny} is the identity on Γ^{Tiny}. The informal meaning of the debugging commands is as follows:

reset sets the debugging state to the initial evaluation history $\langle [\theta_0 : p] \rangle$.

step n advances the evaluation history by n steps of the relation $\overset{\text{EH}}{\Longrightarrow}_E$, where n is a natural number, or until the evaluation terminates. In the latter case the command outputs the constant **succeed** if the evaluation is successful or the constant **fail** otherwise. The result of **step** n when the evaluation does not terminate in n steps is **null**.

run until $id = n$ evaluates the expression from the current evaluation history until a state in which id is bound to n, or until the evaluation terminates. In the latter case the command outputs the constant **succeed** if the evaluation is successful or the constant **fail** otherwise. The result of **run until** $id = n$ when $id = n$ is **null**.

show id outputs the value of the variable id in the current environment if it is defined, and **fail** otherwise.

show pos outputs the expression that is about to be evaluated in the current debugging state.

show res outputs the result of the evaluation of p if applied to a successful state, **fail** is applied to a failure state, and **null** otherwise.

The debugging results are the union of $T(\Sigma^{Env})_{val}$, $T(\Sigma^{Env})_{exp}$, and three constant values. The value **fail** indicates a failure in the evaluation, **succeed** indicates a successful evaluation, and **null** denotes the empty result.

It remains to define the transition relation $\longrightarrow_{Tiny}$ of *Tiny*. The relation is defined for each form of debugging command defined above for an arbitrary $p \in P_{\mathbf{X}^E}(\Omega^{Env})$.

$d =$ **reset** For all $dh \in$ EH, $dh \xrightarrow{d,r} \langle [\theta_0 : p] \rangle$, for $r =$ **null**.

$d =$ **step** n For all $dh \in$ EH and natural number n, $dh \xrightarrow{d,r} dh'$ where:

- If $cur(dh) = [\theta : \varepsilon]$ then $dh = dh'$ and $r =$ **succeed**.
- Otherwise, if $cur(dh) = [\,]$ then $dh = dh'$ and $r =$ **fail**.
- Otherwise, if $n = 0$ then $dh = dh'$ and $r =$ **null**.
- Otherwise, $dh \overset{\text{EH}}{\Longrightarrow}_E dh''$ and $dh'' \xrightarrow{d',r} dh'$ where $d' =$ **step** n' for $n' = n - 1$.

$d =$ **run until** $id = n$ For all $dh \in$ EH, natural number n, and variable id, $dh \xrightarrow{d,r} dh'$, where:

- If $cur(dh) = [\theta : \varepsilon]$ then $dh = dh'$ and $r =$ **succeed**.
- Otherwise, if $cur(dh) = [\,]$ then $dh = dh'$ and $r =$ **fail**.
- Otherwise, if $cur(dh) = [\theta : E \vdash e \Rightarrow v :: \alpha]$ then:
  - if the value of id in $E\theta$ is n then $dh = dh'$ and $r =$ **null**;
  - otherwise, if the value of id in $E\theta$ is different from n or id is not defined in $E\theta$ then $dh \overset{\text{EH}}{\Longrightarrow}_E dh''$ and $dh'' \xrightarrow{d,r} dh'$

$d =$ **show** id For all $dh \in$ EH and variable id, $dh \xrightarrow{d,r} dh$, where: if $cur(dh) = [\theta : \varepsilon]$ then $r =$ **null**; otherwise, if $cur(dh) = [\,]$ then $r =$ **fail**; otherwise, if $cur(dh) = [\theta : E \vdash e \Rightarrow v :: \alpha]$ then r is the value of id in $E\theta$ or **fail** if id is not defined in $E\theta$.

$d =$ **show pos** For all $dh \in$ EH, $dh \xrightarrow{d,r} dh$, where: if $cur(dh) = [\theta : \varepsilon]$ then $r =$ **null**; otherwise, if $cur(dh) = [\,]$ then $r =$ **fail**; otherwise, if $cur(dh) = [\theta : E \vdash e \Rightarrow v :: \alpha]$ then $r = e$.

$d =$ **show res** For all $dh \in$ EH, $dh \xrightarrow{d,r} dh$, where: if $cur(dh) = [\theta : \varepsilon]$ and $fst(dh) = [\theta_0 : E_0 \vdash e_0 \Rightarrow v_0$ then $r = v_0\theta$; if $cur(dh) = [\,]$ then $r =$ **fail**; otherwise $r =$ **null**.

Comments about the Specification

1. It is easy to check that $\longrightarrow_{Tiny}$ obeys the robustness requirement of Definition 2. The functionality requirement of Definition 2 is satisfied by the **step** n command.

2. Single step command **step** 1 is equivalent to one step of the transition $\overset{\text{EH}}{\Longrightarrow}_E$ this being the most detailed step that can be defined in this framework. By using only **step** 1 and **show pos** commands we can see each step of the evaluation. However, in practice it is desirable to have a less detailed single step, so that we may skip details of the evaluation that we are not interested in. We discuss this problem for an abstract debugger in Section 6.4.1, where

we change the definition of *Tiny*'s **step** command to account for a coarser debugging step.

3. The definition of the **show pos** command is very limited because there may be various sub-expressions during the evaluation that are syntactically identical; therefore, we need context information to identify uniquely which sub-expression is about to be evaluated. This problem is studied in the context of an abstract debugger in Section 6.4.2.

4. Some parts of the definition are not formalised, e.g., "of the form" and "the value of *id* in $E\theta$". For a complete formal definition of the debugger those parts must be formalised. As the complexity of the debuggers increase, ad hoc definitions of each detail of the specification will become difficult to carry out, leading to concrete specifications that are difficult to reason about. In Section 6.5 we will see how these parts can be formalised using a specification language.

5. In general, debugging commands will require evaluation in the algebra A_D of the debugger. For instance, suppose we allowed an arbitrary arithmetic expression e in the **run until** *id*= e command. The definition of this command would have to evaluate e using ψ^{A_D} to obtain a natural number.

6.4.1 The Granularity of the Debugging Steps

In this section we study the problem of the granularity of the debugging step. For this, we chose to characterise this problem for an abstract debugger. A debugger according to Definition 2 evaluates programs using the CS system of a Relational Specification. Therefore, the *evaluation step* used by the debugger is a transition defined by the function $\stackrel{EH}{\Longrightarrow}$.

On the other hand, a debugger is a transition system whose states contain a evaluation history as a component. A step of this transition system is what we call a *debugging step*, whose granularity is informally characterised with respect

to the evaluation step: the debugging step is *fine grained* if some debugging command advances the evaluation at a rate close to the evaluation step; it is *coarse grained* if all debugging commands that advance the evaluation do so by the equivalent of many evaluation steps.

In this sense, *Tiny* has fine grained debugging step because the single step command advances the evaluation at exactly the rate of the evaluation step. In fact, *Tiny* has the finest possible debugging step granularity that can be defined in this framework.

A debugger does not necessarily need the finest granularity in the debugging step, and not all debugging commands have to advance the evaluation at the same rate. Sometimes it is useful and desirable to have coarser granularity in the single step command, or even alternative single step commands with different granularities. On the one hand, a useful definition of the granularity of the debugging step should show *every* evaluation step that is of interest for debugging. On the other hand, an efficient definition of the granularity should *only* show the steps of interest. This avoids the need to step through parts of the evaluation that are not relevant in a particular application.

It is obvious that a definition of the optimal granularity of the debugging step is impossible, since it depends on the particular debugger and the applications in which the debugger is used. Therefore, we need a generic mechanism to define the granularity of the debugging steps, which leads to the definition of a *step predicate.* Informally, a step predicate is a predicate on an evaluation history. If SP is a step predicate and $dh \stackrel{\text{EH}}{\Longrightarrow} dh \diamond st$ for some evaluation history dh and evaluation state st, then if $\text{SP}(dh \diamond st)$ is true, the evaluation step is said to be *visible*, and it is *hidden* otherwise.

Let us illustrate the use of step predicates before giving a formal definition. Suppose that we are not interested in debugging the steps of *Fun* expressions related to the evaluation of sum operations, i.e., when using *Tiny* we want to skip over the evaluation of every expression of the form $e_1 + e_2$ and all sub-expressions

in e_1 and e_2. Let *nosum* be a predicate on EH such that $nosum(dh \diamond st)$ holds if and only if $dh \overset{EH}{\Longrightarrow}_E dh \diamond st$ is not an evaluation step of a sum expression. Exactly how *nosum* is defined is left unspecified for the moment. In Section 6.6 we will give a definition of this predicate.

Now we change the definition of *Tiny* such that the commands only show visible steps according to *nosum*. The only command that requires modifications is **step** n, since the command **run until** $id = n$ uses **step** 1 to advance the evaluation.

$d =$ **step** n For all $dh \in$ EH, and natural number n, $dh \xrightarrow{d,r} dh'$, where:

- If $cur(dh) = [\theta : \varepsilon]$ then $dh = dh'$ and $r =$ **succeed**.
- Otherwise, if $cur(dh) = [\,]$ then $dh = dh'$ and $r =$ **fail**.
- Otherwise, if $n = 0$ then $dh = dh'$ and $r =$ **null**.
- Otherwise, $dh \overset{EH}{\Longrightarrow} dh \diamond st$ and:
  - if $nosum(dh, st)$ then $dh \diamond st \xrightarrow{d',r} dh'$ where $d' =$ **step** n' where $n' = n - 1$.
  - else $dh'' \xrightarrow{d,r} dh'$.

The definition below formalises a step predicate for an abstract Relational Specification, so that this concept can be used in any debugger based on such a specification.

Definition 3 (Step Predicate) Let $\mathcal{S} = (\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Relational Specification with Computational Semantics $\mathcal{CS} = \langle \Gamma_{\mathrm{CS}}, \overset{CS}{\Rightarrow}, \mathcal{T}_{\mathrm{CS}} \rangle$, and EH be the set of evaluation histories of p in $\mathcal{S}$. A *step predicate* is a total predicate on EH.

If SP is a step predicate, for all $dh \in$ EH and $st \in \Gamma_{\mathrm{CS}}$ the step $dh \overset{EH}{\Longrightarrow} dh \diamond st$ is *visible* if $\mathrm{SP}(dh \diamond st)$ holds; the step is *hidden* otherwise. If $dh \overset{EH}{\Longrightarrow} dh \diamond st$ is visible then st is a *visible state*. If $dh \overset{EH}{\Longrightarrow}^* dh'$ is a transition sequence in $\mathcal{CS}$, then $|dh \overset{EH}{\Longrightarrow}^* dh'|_{\mathrm{SP}}$ denotes the number of visible steps in the sequence according to SP. □

Step predicates can be used in different ways in a debugger specification and we discuss two of them below. First, a fixed set of step predicates can be incorporated at the specification phase. This fixes which predicates will be used, and (possibly) when they will be applied. Second, debuggers may have commands that give the user the ability to construct step predicates at debugging time, and apply these predicates whenever required. Both approaches are supported in the framework of this thesis and the first one is illustrated in subsequent examples.

An important aspect of the granularity of the debugging steps will be discussed in detail in Chapter 7, when we study Compiler-debuggers. In that case we have an (initial) specification of a debugger according to Definition 2, and another specification given by a Compiler-debugger in which programs are executed directly on a machine. For the Compiler-debugger to be correct with respect to the initial specification it must map the machine execution of the program into the visible evaluation steps of this specification. This mapping causes an undesirable overhead in the execution of the program. In general, this overhead is greatest where the granularity of the debugging steps is finest; thus specifications with a coarser granularity should allow for more efficient implementation of Compiler-debuggers.

6.4.2 Unique Reference to Sub-programs

As we pointed out before, the definition of the command **show pos** of *Tiny* is limited: its result, on a given evaluation history, does not reference a unique point in the program. However, the ability to reference a unique sub-program is important in debugging for setting break-points or trace-points for example.

One way of uniquely referencing a sub-program is by the path from the root of the abstract syntax tree representing the program to the sub-program we want to reference. For instance, let us consider a parse-tree representation of the term $+(+(5,1),+(\underline{+(5,1)},2))$ that is shown in Figure 1. The path to the (underlined) sub-term $\underline{+(5,1)}$ is indicated on the tree by the double lines.

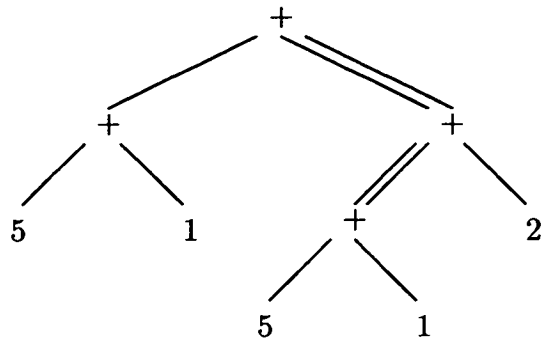

Figure 1: Parsing Tree

Notice that the term $+(5,1)$ does not give a unique reference to a sub-expression in this example. A possible representation for this path is a sequence $\langle \underline{0}, \underline{2}, \underline{1} \rangle$, where we underlined the numbers to avoid confusion with the constant expressions 1, 2, and so forth. In this representation, $\underline{0}$ denotes the root of the parsing tree, $\underline{2}$ denotes a right branch, and $\underline{1}$ denotes a left branch. The sequence represents the order in which the branches are taken to find the desired subtree. This representation generalises easily to parsing trees with nodes with more than 2 branches.

If paths of expressions are available for the definition of *Tiny*, the command **show pos** can be defined such that its result on any evaluation history is the path of the current expression. Using a convenient user interface, this result could be displayed to the user, e.g., as a highlight of the sub-expression.

However, paths are not an explicit component of an evaluation history; thus in the rest of this section we will address the problem of how to make them available for the debuggers. There are various approaches to this problem in our framework. We can built the path from the evaluation history whenever the path is required. Alternatively, we can build the path of the current sub-expression incrementally and keep this information as a component of every debugging state so that it is always readily available. Clearly there is a trade off between these two solutions for the former is time consuming whereas the latter consumes storage

resources to keep the path.

Another solution is to make the path an explicit component of the debugging history. This approach poses interesting problems, and we will develop it in more detail here. We start by formalising the concept of syntax path, and the first step in this direction is the definition of a path tree.

Definition 4 (Path Tree and Syntax Path) The *path tree* of a term $t \in T_X(\Sigma)$ is the term $\mathcal{T}_0(t)$, where $\mathcal{T}_n(t)$ is defined for each $n \geq 0$ as follows:

1. $t = x, x \in \mathrm{X}$, then $\mathcal{T}_n(t) = n(\,)$

2. $t = \sigma(t_1, \ldots, t_m), m \geq 0$, then $\mathcal{T}_n(t) = n(\mathcal{T}_1(t_1), \ldots, \mathcal{T}_m(t_m))$

An *abstract syntax path* (or simply a *path*) is a sequence of nodes occurring in the path from the root of $\mathcal{T}_0(t)$ to some sub-tree of $\mathcal{T}_0(t)$. □

For instance, the path tree of the term $+(+(5,1),+(\underline{+(5,1)},2))$ is the term $\underline{0}(\underline{1}(\underline{1}(),\underline{2}()),\underline{2}(\underline{1}(),\underline{2}(\underline{1}(),\underline{2}())))$. Therefore, the path to the underlined sub-term $\underline{+(5,1)}$ is the sequence $\langle \underline{0}, \underline{2}, \underline{1} \rangle$, as discussed above. It remains to show how paths can be included in the evaluation histories.

This problem can be considered from two points of view. First, we may consider that the CS system is deficient because it does not provide the information we need for debugging. If this view is taken, we must extend the CS system to incorporate this information, in this case the path of a sub-program.

There are various ways in which this extension can be done. A simple solution, that can be incorporated to the CS system, is illustrated by the following example. Suppose that the formula $\varepsilon_E \vdash +(3,6) \Rightarrow v$ is the query we want to evaluate in CS^{Env}. We first annotate each term of the expression $+(3,6)$ with its path, such that the *annotated term* becomes $+(3_{\langle \underline{0},\underline{1} \rangle}, 6_{\langle \underline{0},\underline{2} \rangle})_{\langle \underline{0} \rangle}$. Therefore, the original query is changed into $\varepsilon_E \vdash +(3_{\langle \underline{0},\underline{1} \rangle}, 6_{\langle \underline{0},\underline{2} \rangle})_{\langle \underline{0} \rangle} \Rightarrow v$.

Then we change the unification algorithm *Unify* used by the CS system such that it ignores the annotations when unifying expressions. With this new unification algorithm the (simplified) steps of the evaluation of the above query

become:

$$[\varepsilon_E \vdash +(3_{\langle 0,1\rangle}, 6_{\langle 0,2\rangle})_{\langle 0\rangle} \Rightarrow v] \stackrel{CS}{\Rightarrow}$$
$$[\varepsilon_E \vdash 3_{\langle 0,1\rangle} \Rightarrow v_1 :: \varepsilon_E \vdash 6_{\langle 0,2\rangle} \Rightarrow v_2 :: \ldots] \stackrel{CS}{\Rightarrow}$$
$$[\varepsilon_E \vdash 6_{\langle 0,2\rangle \Rightarrow v_2} :: \varepsilon_E \vdash \mathbf{plus}(3, v_2) \Rightarrow v] \stackrel{CS}{\Rightarrow}$$
$$[\varepsilon_E \vdash \mathbf{plus}(3,6) \Rightarrow v] \stackrel{CS}{\Rightarrow}$$
$$[\mathbf{success}]$$

In this example, we applied the substitution to the formulae in the goal stack and omitted the substitution from the states to simplify the presentation. Furthermore, the successful state is represented by [**success**]. These simplifications will be adopted in the examples hereafter. At each state of the above evaluation history we have access to the path of the sub-term of $+(3,6)$ that is about to be evaluated. Notice that this solution works even for function application involving a closure, because the expression in the body of the closure would also be annotated with its path.

A second approach the problem of including paths in the evaluation histories is to consider that the Relational Specification used in the definition of the debugger is deficient with respect to the information necessary for debugging. This view assumes that the Relational Specification determines the possible debugging functionality. In other words, two Relational Specifications that are equivalent with respect to final results of program evaluation may allow the definition of different debugging commands. The difference lies in that one specification may have more debugging information available than the other, although they are equivalent in the evaluation of programs.

In this approach, if we need to reference a unique point in the program, the path of the program must be an explicit component of the Relational Specification. For instance, we can add paths to *Env* as terms of sort *path*, and change the definition of closures as follows:

$$closure \quad ::= \quad (env, var, exp, path)$$

With the above changes, the following set of rules for the language *Fun* manip-

ulate the path of the expressions explicitly. In these rules, *pa* is a meta-variable of sort *path* and X^{E}_{path} is generated by *pa*.

Evaluation with Paths $\boxed{env \vdash exp, path \Rightarrow val}$

$$\frac{}{E \vdash n, pa \Rightarrow n}$$

$$\frac{(E, id) \longrightarrow_L v}{E \vdash id, pa \Rightarrow v}$$

$$\frac{E \vdash e_1, pa \diamond \underline{1} \Rightarrow v_1 \qquad E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 \qquad E \vdash \mathbf{plus}(v_1, v_2), \langle\,\rangle \Rightarrow v}{E \vdash e_1 + e_2, pa \Rightarrow v}$$

$$\frac{E \vdash e_1, pa \diamond \underline{2} \Rightarrow v_1 \qquad id \mapsto v_1 \cdot E \vdash e_2, pa \diamond \underline{3} \Rightarrow v_2}{E \vdash \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2, pa \Rightarrow v_2}$$

$$\frac{}{E \vdash \mathbf{fn}\ id\ .\ e, pa \Rightarrow (E, id, e, pa \diamond \underline{2})}$$

$$\frac{E \vdash e_1, pa \diamond \underline{1} \Rightarrow (E', id, e', pa') \qquad E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 \qquad id \mapsto v_2 \cdot E' \vdash e', pa' \Rightarrow v'}{E \vdash e_1(e_2), pa \Rightarrow v'}$$

It is possible to prove that the above set of rules is equivalent to the rules of ϕ^{Env} with respect to the final results of the evaluation of expressions. Furthermore, the evaluation steps produced using the new set of rules are in one-to-one correspondence with the evaluation steps using the original ϕ^{Env}. Moreover, using this set of rules in the definition of *Tiny* makes the path of the current expression available in the evaluation history. For instance, if we evaluate the query $\varepsilon_E \vdash +(3,6), \langle \underline{0} \rangle \Rightarrow v$ using the above rules, we obtain the following evaluation

history:

$$[\varepsilon_E \vdash +(3,6), \langle \underline{0} \rangle \Rightarrow v] \overset{CS}{\Rightarrow}$$
$$[\varepsilon_E \vdash 3, \langle \underline{0},\underline{1} \rangle \Rightarrow v_1 :: \varepsilon_E \vdash 6, \langle \underline{0},\underline{2} \rangle \Rightarrow v_2 :: \ldots] \overset{CS}{\Rightarrow}$$
$$[\varepsilon_E \vdash 6, \langle \underline{0},\underline{2} \rangle \Rightarrow v_2 :: \varepsilon_E \vdash \mathbf{plus}(3, v_2), \langle\,\rangle \Rightarrow v] \overset{CS}{\Rightarrow}$$
$$[\varepsilon_E \vdash \mathbf{plus}(3,6), \langle\,\rangle \Rightarrow v] \overset{CS}{\Rightarrow}$$
$$[\mathbf{success}]$$

At each state of the above evaluation history, the path of the current subexpression is available. In this sense, this solution is equivalent to that of annotating the expressions as discussed above. However, in this case we do not change the $\mathcal{CS}$ system, since the paths are just normal terms in the Relational Specification.

A possible argument against this approach is that it involves changing the Relational Specification of the programming language to include components that are not relevant to the definition of program evaluation. However, in this thesis Relational Specifications are used to defined program debugging as well as program evaluation; thus it seems reasonable to include debugging information in the Relational Specifications.

We adopt this approach hereafter, that is, we keep the $\mathcal{CS}$ system unchanged and extend the Relational Specifications with debugging information whenever it is necessary. We favour this approach for several reasons. First, it is more flexible than changing the $\mathcal{CS}$ system. Second, it keeps the basic framework simple. Finally, it does not preclude other solutions, including extensions to the $\mathcal{CS}$ system, as the one discussed above.

When presenting the specification language in the next section we shall discuss another reason to choose this approach instead of changing the $\mathcal{CS}$ system. Hereafter, we assume that the above set of rules for *Fun* replaces the definition of the relation $\_ \vdash \_ \Rightarrow \_$ in the Relational Specification *Env* (page 80). Therefore, paths are terms in *Env* of sort *path*. This new set of rules will be important in the formalisation of *Tiny* in the next section.

Alternative treatments of the problem addressed in this section are given

in [Ber91b,Ber92]. The former uses *occurrences* as defined in [Bou85] to designate unique sub-terms of a λ-term. The approach in [Ber92] uses the idea of origin functions. In both approaches the programming language is the simple untyped λ-calculus. It would be interesting to investigate the applicability of these approaches to other languages, e.g., imperative language.

6.5 A Notation for Specifying Debuggers

In the previous sections we presented an abstract definition of debuggers and studied some aspects of the specification of concrete debuggers according to that abstract definition. We showed how a semi-formal specification of a simple debugger may be directly defined in terms of the CS states, formulae, and substitutions. This example demonstrated that giving a complete formal specification of a debugger requires the formalisation of various details, for instance how to obtain the value of a variable from an environment at the current evaluation state. The ad hoc formalisation of such details is possible but may become difficult as the complexity of the debugging commands increase; moreover, it is tedious to formalise the same concept every time we are using a new Relational Specification.

In this section we define a notation to assist in the specification of debuggers, including the definition of visibility predicates. The main features of this specification notation, called DSL, is a powerful concept of *sequence patterns*, which are used to describe evaluation histories. In the rest of this section we introduce DSL using examples. In Section 6.6 we present a specification of *Tiny* written in DSL, including the visibility predicate *nosum* discussed in Section 6.4.1. The syntax and formal semantics of sequence patterns are defined in Appendix A.

The most important feature of DSL is the use of sequence patterns to specify evaluation histories. The concept of pattern matching on sequences is not new, appearing fist in the language SNOBOL [FGP64], and in other programming languages since then, e.g., SNOBOL4 [GPP68] and SL5 [GH77]. In these

languages, string patterns are used to describe strings of characters. In DSL, sequence patterns generalise this idea to sequences of arbitrary objects. In this thesis we will use DSL as a notation to describe evaluation histories; therefore we will be interested in *sequences of evaluation states.*

DSL is a "sugared" version of the untyped λ-calculus [Bar85] to which patterns and pattern matching are added in the standard way, e.g., as described in [FH88]. The use of the λ-calculus with pattern matching as the mathematical underpinings of concrete functional languages, e.g., Hope [BMS80], Standard ML [HMT89], and Haskell [HW90], is well known and so we assume some familiarity with this idea. However, the reader familiar with string pattern matching in SNOBOL must be warned that the pattern matching in functional languages is different from the SNOBOL paradigm. In SNOBOL, patterns are first class objects that can be constructed dynamically; it is even possible to creat recursive patterns whose meaning is determined when the pattern matching is taking place. In functional languages patterns are used only in function definition. It is the *functional language* notion of patterns that will be described in the rest of this section.

Values in DSL are divided into *basic values* and *sequences of basic values.* This classification on the values imposes a classification of the patterns into *basic patterns* and *sequence patterns.* Therefore, sequences in DSL are first order objects in the sense that there is no sequences of sequences. First order sequences are sufficient for the purposes of specifying debuggers since evaluation histories are first order sequences of evaluation states. An interesting problem is to generalise DSL to deal with higher order sequences.

DSL is parametric on a Relational Specification in the sense that the Specification provides the basic constructors of DSL values. To simplify this presentation we will describe the particular instantiation of DSL with the Relational Specification *Env.* This instantiation will be called DSL_{Env} hereafter. Recall that, in the set of rules of Section 6.4.2, the formulae in $F(\Omega^{Env})_{\_\vdash\_\Rightarrow\_}$ were extended with a

term of sort *path*, such that $E \vdash e \Rightarrow v$ becomes $E \vdash e, pa \Rightarrow v$.

We choose to present DSL as an untyped language mainly because it simplifies the presentation. Moreover, the typing of patterns does not pose new problems; it is the dynamic aspects of the pattern matching that are novel to DSL.

Basic Values and Basic Patterns

The basic values of DSL_{Env} are the evaluation states of $\mathcal{CS}^{Env}$. Therefore, basic values are not atomic: they are composed of substitutions, goal stacks, formulae, and terms. The function names in F^{Env}, the relation names in Π^{Env}, and the meta-variables in X^{E} are the basic constructors in DSL_{Env}. The following terms and formulae are values in DSL_{Env}:

$$2 + \mathbf{x}$$
$$\mathbf{x} \mapsto 2 \cdot \varepsilon_E$$
$$\langle \underline{0}, \underline{2}, \underline{1} \rangle$$
$$\mathbf{x} \mapsto 2 \cdot \varepsilon_E \vdash 2 + \mathbf{x}, \langle \underline{0}, \underline{2}, \underline{1} \rangle \Rightarrow v$$

To make the constructors of the Relational Semantics different from other DSL_{Env} constructors we use the following convention: reserved words of the programming language are written in **bold** type; mathematical symbols are used for the relation and function names of the Relational Specification of the programming language; meta-variables of the Relational Specification are written in *italic* font; DSL_{Env} reserved words and identifiers are written in `type writter` font. Therefore, **x**, + and 2 are programming language objects, whereas `x`, `+` and `2` are DSL_{Env} objects. As an example, the following state in $\mathcal{CS}^{Env}$ is a basic value in DSL_{Env}:

$$[\theta : \mathrm{E} \vdash e_1, pa_1 \Rightarrow (\mathrm{E}', id, e, pa) :: \mathrm{E} \vdash e_2, pa_2 \Rightarrow v :: id \mapsto v \cdot \mathrm{E}' \vdash e, pa \Rightarrow v' :: \ldots]$$

where ... is an informal notation to indicate that there are more formulae on the right of the goal stack. We decompose a basic value into its sub-components using pattern matching. *Basic patterns* are the same as constructor patterns. Such patterns are commonly used in programming languages like Standard ML,

and are formally presented for example in [FH88]. For instance, matching the pattern

$$\texttt{[sub : \_ :: f2 :: tl]}$$

on the above evaluation state binds the pattern variable sub to the substitution θ, f2 to the formula $\mathrm{E} \vdash e_2, pa_2 \Rightarrow v$, and tl to $id \mapsto v \cdot \mathrm{E}' \vdash e, pa \Rightarrow v' :: \ldots$; the pattern "_" is a wildcard that matches any value. To obtain the instantiation of the meta-variables, e.g., in the formula f2, we must use the substitution that is bound to sub. For instance, $\mathcal{I}(\texttt{f2}, \texttt{sub})$ is the instantiation of $\mathrm{E} \vdash e_2, pa_2 \Rightarrow v$ by θ, where $\mathcal{I}$ is the instantiation function defined in Section 2.3.

It is necessary to use $\mathcal{I}$ whenever we want to obtain the instantiation of a term or formula by a substitution. However, to improve the readability of the following examples we will "apply" the substitution to a term or formula as an abbreviation. For instance, we will use sub(f2) as an abbreviation for $\mathcal{I}(\texttt{f2}, \texttt{sub})$, whenever this does not cause ambiguities.

The constructor success matches an empty goal stack indicating a successful evaluation; thus the pattern [_ : success] matches a success state of the form $[\theta : \varepsilon]$. Furthermore, formulae can be decomposed into their terms also using constructor patterns. The matching of the pattern

$$\texttt{\_ ⊢ e, \_ ⇒ v}$$

on the formula $\mathrm{E} \vdash e_2, pa_2 \Rightarrow v$ binds e to e_2 and v to v. Then, applying sub to e gives that actual instantiation of the meta-variable e_2 in the above state, i.e., sub(e) $= e_2\theta$. Constructor patterns are also used to decompose a term into its sub-terms.

Every DSL_{Env} value is denoted by an expression. Terms, formulae, and $\mathcal{CS}^{Env}$ states are expressions that denote their corresponding values. Other forms of expression are λ-abstraction and function application. These are described in the next section.

Functions

A function value is denoted by a λ-abstraction that in the concrete DSL notation has the following general form:

$$\texttt{fn}\ pat_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ pat_n\ \texttt{=>}\ e_n$$

for some $n > 0$. Since patterns can overlap, i.e., more than one pat_i can match a given value, it is important to impose that the patterns $pat_1, \ldots, pat_n$ must be tried in this order.

Function application is written $e_1\ e_2$, where e_1 and e_2 are expressions. Parenthesis may be used to resolve ambiguities. The bindings of variables in λ-abstractions are determined at declaration time, and functions are strict on all their arguments. In other words, DSL is an "eager" functional language with "static binding". The formal semantics of such a language is standard, and can be found for instance in [FH88].

Let d denote a *declaration* of the form `val` pat `=` e. As a "syntactical sugar" we use `let` expressions of the form `let` d `in` e' `end`, where e' is an expression. The meaning of the evaluation of this `let` expression is defined by the application `(fn` pat `=>` e'`)(`e`)`.

Moreover, the expression `let` d_1 `in` $\ldots$ `let` d_n `in` e `end`, for some $n > 0$, is written `let` d_1`;`$\ldots$`;`d_n `in` e `end`. We will use declarations as a way to give names to values. These names will then be used in the specification of debuggers. Moreover, a declaration of the form:

$$\texttt{val f = fn}\ pat_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ pat_n\ \texttt{=>}\ e_n$$

for some identifier `f` and $n > 0$, is written as a *function declaration* of the form:

$$\texttt{fun f}\ pat_1\ \texttt{=}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{f}\ pat_n\ \texttt{=}\ e_n$$

If `f` is applied to an argument that does not match any pat_i then the result of the application is undefined. In later examples we will introduce other forms of expressions that are derived from the above expressions.

Furthermore, we add the integer numbers and boolean values `true` and `false` to DSL_{Env}. The usual arithmetic operators on integer numbers `+`, `-`, `*`, `div` are available as DSL_{Env} functions. The usual relational operators `<`, `<=`, `>`, `>=`, `=`, and `<>` are also available as DSL_{Env} functions that produce boolean results.

Evaluation History Values

The sequence values of DSL_{Env} that we are interested in for the specification of debuggers are evaluation histories in EH. Although we can use constructors to build evaluation histories as we do for the basic values, this use can make it difficult to establish the requirements of Definition 2. Therefore, we provide two pre-defined functions to (safely) build evaluation histories.

The pre-defined functions `initial` and `next` are defined with respect to an arbitrary Relation Specification $\mathcal{S} = (\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$, where $\Omega = (\mathrm{S}, \mathrm{F}, \Pi, \pi)$, and a S-sorted set of meta-variables X.

- `initial`$(p) = \langle [\theta_0 : p] \rangle$ if $p \in \mathrm{P}_{\mathrm{X}}(\Omega)$.
- `next`$(dh) = dh \diamond st$ such that $cur(dh) \overset{\mathrm{CS}}{\Rightarrow} st$.

The use of pre-defined functions to create histories guarantees that the debugging histories in DSL_{Env} are correct with respect to $\mathcal{CS}^{Env}$. In this sense, correctness means that we cannot construct a debugging history that is not reachable from some initial evaluation history $\langle [\theta_0 : p] \rangle$ using the $\mathcal{CS}^{Env}$ system. This is important for two main reasons. First, since $P_{\mathbf{X}E}(\Omega^{Env})$ is the set of data-driven program formulae of *Env*, according to Definition 5.2, `next` is complete in the sense of Theorem 5.3. This helps in proving the requirements of Definition 2 for specifications of concrete debuggers. Second, `next` gives the intuitive evaluation step defined in Section 5.4, which helps in the specification of debuggers that behave intuitively. The above discussion applies to an arbitrary Deterministic Relational Specification.

We give some examples of the use of the predefined functions. To simplify the presentation we write $\mathbf{x} \mapsto 2$ for the environment $\mathbf{x} \mapsto 2 \cdot \varepsilon_E$, omiting the ε_E

constant. For instance, the following expression:

$$\texttt{initial}(\mathbf{x} \mapsto 2 \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v)$$

evaluates to the initial debugging history $\langle [\theta_0 : \mathbf{x} \mapsto 2 \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v] \rangle$, where θ_0 is the identity substitution on X^E, and $\langle \underline{0} \rangle$ is the abstract-syntax path of the expression x. On the other hand, the value of $\texttt{initial}(E \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v)$ is undefined because E is a meta-variable and therefore $E \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v$ is not a program formula. As another example, the expression

$$\texttt{next}(\texttt{next}(\texttt{initial}(\mathbf{x} \mapsto 2 \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v)))$$

evaluates to the following history:

$$\langle [\theta_0 : \mathbf{x} \mapsto 2 \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v], [\theta_0\theta_1 : (\mathbf{x} \mapsto 2, \mathbf{x}) \longrightarrow_L v], [\theta_0\theta_1\theta_2 : \varepsilon] \rangle$$

such that $\theta_0\theta_1\theta_2(v) = 2$.

Sequence Patterns

Using the pre-defined functions discussed above we can construct an evaluation history from a program formula, and advance evaluation histories according to the function $\stackrel{\mathbf{EH}}{\Longrightarrow}$. However, we cannot extract states from a history or decompose a history into its sub-histories. These operations are done by pattern matching a sequence pattern on a debugging history value. Sequence patterns are the main feature of DSL for the specification of debuggers.

The simplest sequence pattern decomposes a debugging history into its current state and the rest of the history. For instance the following declaration:

$$\texttt{val <h, st> = next(next(initial(}\mathbf{x} \mapsto 2 \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v\texttt{)))}$$

binds the identifier h to the history

$$\langle [\theta_0 : \mathbf{x} \mapsto 2 \vdash \mathbf{x}, \langle \underline{0} \rangle \Rightarrow v], [\theta_0\theta_1 : (\mathbf{x} \mapsto 2, \mathbf{x}) \longrightarrow_L v] \rangle$$

and st to the state $[\theta_0\theta_1\theta_2 : \varepsilon]$.

The generic form of the above pattern is a sequence of constructor patterns separated by "," and enclosed in "<" and ">". The symbol "," should be interpreted as a left associative constructor. Constructor patterns were discussed above when we described the basic DSL_{Env} values. The pattern <> matches the empty sequence. In the following examples, we will omit "<"and ">" from sequence patterns of the form <pat> in which pat is a single constructor pattern.

Sometimes it is desirable to decompose a compound value using patterns and at the same time to bind the entire compound value to a pattern variable. For instance, in the declaration

```
val h' as <h, st> = next(next(initial(x ↦ 2 ⊢ x, ⟨0⟩ ⇒ v)))
```

the variable h and st are bound as above, and h' is bound to the entire history resulting from the function application.

The following functions show the use of sequence patterns to define the evaluation of an expression in an environment. In this and subsequent examples we use tuples of expressions and their corresponding values with the usual meaning. Furthermore, we add to DSL_{Env} the constructors defined by the following signature declaration:

result ::= **fail** | **succeed** | **null** | **val**(*val*) | **path**(*path*)

such that a term generated by this signature is a DSL_{Env} value.

```
fun complete(h as <h', [_ :  success]>) = h
  | complete(h as <h', [ ]>) = h
  | complete(h) = complete(next(h))

fun evaluation(E,e) =
     case complete(initial(E ⊢ e, ⟨0⟩ ⇒ v)) of
        <h', [sub :  success]> => val(sub(v))
        <h', [ ]> => fail
```

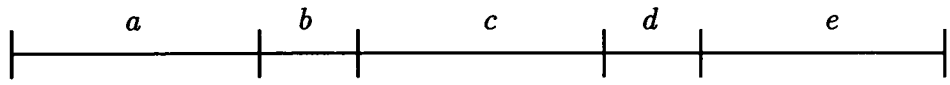

Figure 2: Schematic Representation of a Debugging History

The function `complete` advances the history `h` until either a success or a failure state is found. In the declaration of `evaluation` v is a constructor in X^{E}_{val}, and **fail** is a constant that indicates a failure in the evaluation. As an example, the call `evaluation(x` $\mapsto 2,\ 2 + $ `x)` results in 4.

The patterns described so far are the most basic patterns of DSL_{Env}. Given a debugging history, we can write a pattern to extract any state from this history, any formula from a goal stack, and so forth. However, these patterns cannot express certain events that are important for debugging. For instance, we often want to know the result of the most recently evaluated sub-expression of a certain form. Certainly, we cannot directly express this event using the above patterns. Without more sophisticated patterns we would have to resort to the use of recursive functions to specify these kind of events, making the specifications more difficult to understand and reason about.

The next patterns we shall describe offer more expressiveness in describing events that occur in a debugging history. For the presentation of these patterns let us consider the schematic representation of a debugging history that is shown in Figure 2, where the entire history will be referred to as h. The current state $cur(h)$ is the right most state, and the sub-histories a, b, c, d, and e indicated in the diagram are consecutive, i.e., $h = a^\frown b^\frown c^\frown d^\frown e$.

The sequence pattern $\iota(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e})$, where `a`, `b`, `c`, `d`, and `e` are sequence patterns, is an *interval pattern*. Intuitively, $\iota(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e})$ describes the right-most smallest interval delimited by sub-histories b and d such that each sub-matching is successful. This pattern matches h if e is the smallest sub-history of h such that `d` matches d, and once e and d are fixed then c is the smallest sub-history

such that b matches b, and once e, d and b are all fixed then a matches a, c matches c, and e matches e. Hereafter, we use the term ι-pattern for interval pattern.

It is important to emphasise that the matching of d on d must be fixed before we attempt to match b on b, and both these matchings must be fixed before we attempt the remaining matchings. This ensures that the pattern matching is deterministic and also avoids a (potentially) exponential number of attempts to match the ι-pattern.

The sequence pattern $\beta(\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e})$, where a, b, c, d, and e are sequence patterns, is a *balanced interval pattern*. Consider the diagram of Figure 2. This pattern matches h provided the same conditions for matching $\iota(\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e})$ are established and furthermore if in c there exist an equal number, say $n \geq 0$, of sub-histories d_i and b_i such that d matches d_i and b matches b_i, for $i \in [n_+]$. This last condition together with the fact that e and c are the smallest sub-histories of h for which the matchings are successful ensure that, if we consider b as describing an open bracket and d describing a close bracket, the brackets in the sequence $b^\frown c^\frown d$ are balanced in the usual sense.

Intuitively, $\beta(\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e})$ describes the right-most smallest interval delimited by b and d with the extra-condition that $b^\frown c^\frown d$ is balanced with respect to sub-sequences that match b and d. Hereafter, we use the term β-pattern for balanced interval pattern.

The bindings resulting from the matching of an ι-pattern $\iota(\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e})$ are the union of the bindings resulting from the matching of a, b, c, d, and e; same for a β-pattern. The same variable may occur more than once in a pattern, i.e., patterns do not have to be linear. For the matching to be successful, each occurrence of the same variable must match equal values. This assumes the existence of an equality relation on all values of DSL_{Env}. However, substitutions are DSL_{Env} values. Since substitutions are functions, and since we do not expect that the mathematical equality on functions to be checked automatically, we

will assume that two functions, and consequently two substitution, are always different for the purposes of pattern matching. Therefore, any two debugging histories and any two evaluation states are always different for the purposes of pattern matching.

The sequence pattern ... is a wild card that matches any debugging history. It is convenient to notice that since the matching of interval patterns start from the sub-pattern d, a pattern of the form $\iota(\mathtt{a}, \mathtt{b}, \mathtt{c}, ..., \mathtt{e})$ makes d be the entire sequence h. Therefore, this pattern only matches h if the sub-patterns a, b, c, and e were each either ... or <>.

Interval patterns can be used to describe the successful evaluation of a program formula starting from the initial state. For instance, consider the pattern below:

$$\iota(<>, [\_ : \ \mathtt{E} \vdash \mathtt{e}, \ \_ \Rightarrow \mathtt{v}], ..., [\mathtt{sub} : \ \mathtt{success}], <>)$$

where [_ : E ⊢ e, _ ⇒ v] describes the initial state, [sub : success] describes a final successful state, and ... allows an arbitrary number ot states between the initial and final ones. If there is a successful matching of this pattern on a debugging history then sub(v) is the result of the evaluation of the expression e on environment E.

A combination of ι-patterns and β-patterns can be used to describe the most recent evaluation of an expression of a certain form. For instance, suppose we want to describe the most recent evaluation of a sum expression $e_1 + e_2$. Let us start by defining a β-pattern to describe a successful evaluation of a sum expression:

$$\beta(..., [\_ : \ \_ \vdash \mathtt{e1} + \mathtt{e2}, \ \_ \Rightarrow \mathtt{v} :: \ \mathtt{gs}], ..., [\mathtt{sub} : \ \mathtt{gs}], ...)$$

To simplify the presentation, let us use *SUM* to refer to the above pattern. Notice that since sub and gs are pattern variables, the sub-pattern [sub : gs] matches the rightmost state in the history. Since gs must match equal values in both sub-patterns, *SUM* describes the successful evaluation of a sum expression

such that the last state in this evaluation is the second rightmost state of the history

To describe the most recent evaluation of a sum expression we have to make the last state of this evaluation to be in an arbitrary position in the history. In other words, we must "shift" the pattern *SUM* to the left of the history an arbitrary number of states. This is achieved using the following ι-pattern:

$$\iota(..., SUM, ..., [\_:\_], ...)$$

In this pattern, [_:_] matches the rightmost state, and the pattern matching then continues to the left until *SUM* matches successfully. Therefore, the above ι-pattern describes the most recent evaluation of a sum expression. When this pattern is matched successfully on a debugging history, sub(v) is the result of the evaluation of the sum expression sub(e1 + e2).

Since patterns of the form

$$\iota(..., \mathtt{b}, ..., \mathtt{d}, ...)$$
$$\beta(..., \mathtt{b}, ..., \mathtt{d}, ...)$$

for arbitrary b and d are likely to occur often in debugger specifications we introduce the following *derived form* of balanced interval patterns as an alternative to the above patterns:

$$\iota(\mathtt{b}, \mathtt{d})$$
$$\beta(\mathtt{b}, \mathtt{d})$$

Therefore, in the derived forms the above patterns become:

$$\beta([\_ : \_ \vdash \mathtt{e1} + \mathtt{e2}, \_ \Rightarrow \mathtt{v} :: \mathtt{gs}], [\mathtt{sub} : \mathtt{gs}])$$
$$\iota(SUM, [\_:\_])$$

Interval and balanced interval patterns express a large set of events in a debugging history that are important for debugging. To show more examples of the use of these patterns let SUM_i denote the pattern

$$\beta([\_ :\_ \vdash \mathtt{e1}_i + \mathtt{e2}_i, \_ \Rightarrow \mathtt{v}_i :: \mathtt{gs}_i], [\mathtt{sub}_i : \mathtt{gs}_i])$$

and $SINGLE_i$ denote the pattern $[\_:\_]$, for all $i \in [n_+]$ and some $n > 0$. The following balanced interval pattern describes the second most recently evaluated sum expression:

$$\iota(\iota(SUM_1, SINGLE_1), SUM_2, ..., SINGLE_2, ...)$$

Using the same idea we could describe the n^{th} most recently evaluated sum expression, for any $n > 0$. However, it can be easily predicted that, even for $n = 3$, the pattern will become tedious to write and difficult to read and understand, specially when we write the pattterns SUM_i and $SINGLE_i$ explicitly. Therefore, we introduce another derived form of balanced interval patterns which, for the patterns SUM_1 and $SINGLE_1$ defined above, is written:

$$\iota^n(SUM_1, SINGLE_1)$$

This pattern describes the n^{th} most recent evaluation of a sum expression. In general this derived form is written $\iota^n(\mathtt{b}, \mathtt{d})$ for arbitrary b and d. Using the same idea we introduce the derived interval pattern $\beta^n(\mathtt{b}, \mathtt{d})$.

The sequence patterns described above provide almost all the expressiveness that is required in the specification of debuggers. However, we cannot express the largest n such that $\iota^n(\mathtt{b}, \mathtt{d})$ matches the debugging history. For this we introduce the history pattern $\iota^{max}(\mathtt{b}, \mathtt{d})$ that denotes the pattern $\iota^n(\mathtt{b}, \mathtt{d})$ for the largest $n > 0$ such that $\iota^n(\mathtt{b}, \mathtt{d})$ matches the debugging history. Similarly, $\beta^{max}(\mathtt{b}, \mathtt{d})$ that denotes the pattern $\beta^n(\mathtt{b}, \mathtt{d})$ for the largest $n > 0$ such that $\beta^n(\mathtt{b}, \mathtt{d})$ matches the debugging history

Therefore, using the patterns SUM_1 and $SINGLE_1$ defined above, the pattern $\beta^{max}(SUM_1, SINGLE_1)$ describes the earliest evaluation of a sum expression with no unfinished evaluation of a sum expression on its right.

The combined use of interval, balanced interval, ι^{max}, and β^{max} patterns is powerful. For instance, we can write a single pattern that expresses the right-most unfinished evaluation of a sum expression as follows:

$$\iota([\_:\_ \vdash \mathtt{e1}_1 \ + \ \mathtt{e2}_1, \_ \Rightarrow \mathtt{v}_1 :: \mathtt{gs}_1], \iota^{max}(SUM_2, SINGLE_2))$$

Figure 3: Unfinished evaluation of a Sum Expression

We can understand the meaning of the above pattern by looking at the diagram of Figure 3. In that diagram, the sub-histories $dh_1, \ldots, dh_{max}$ are the successful evaluations of a sum expression, which are described by the pattern

$$\iota^{max}(SUM_2, SINGLE_2)$$

The state *st* is the first state on the left of dh_{max} such that the pattern

$$[\_ : \ \_ \vdash \mathtt{e1}_1 + \mathtt{e2}_1, \ \_ \Rightarrow \mathtt{v}_1 :: \ \mathtt{gs}_1]$$

matches *st*, and the evaluation of the sum expression described by the pattern $\mathtt{e1}_1 + \mathtt{e2}_1$ is unfinished otherwise *max* would not denote the largest number of matchings of the finished evaluations.

The patterns described in this section are all the patterns of the DSL_{Env}, and in general of DSL. We argue that those patterns can describe most of (if not all) the events in an debugging history that are relevant for debugging purposes. In the next section we shall see some examples of the use of these patterns in the specification of a concrete debugger. This finishes the introduction to language DSL, the complete syntax and semantics of which are defined in Appendix A.

It is possible to specify sequences, and in particular evaluation histories, using arbitrary recursive functions, as discussed in [GH80]. An important advantage of using patterns is the possibility of defining equational theories. Reasoning about the specified histories in such a theory can be simpler than reasoning about arbitrary recursive functions. Consequently, such a theory would make reasoning about the debuggers specified in DSL easier, e.g., in proofs of debugger

correctness. This problem is not addressed in this thesis, but constitute an interesting extension to our investigation.

6.6 The Specification of an Interpreter-Debugger

In this section we give a complete specification of *Tiny* using DSL_{Env}. In this example we illustrate that the specifications in DSL_{Env}, and in general in DSL, are indeed more abstract and concise than ad hoc formalisations of the debuggers.

We present the specification in two stages. First, we give a specification of *Tiny* with the finest step granularity. Then, we formalise two step predicates, including the predicate *nosum* of Section 6.4.1, and change *Tiny* to use the predicates to increase the granularity of the debugging steps.

The Definition of Some Auxiliary Functions

We start defining some auxiliary DSL_{Env} functions. For these definitions we use two derived forms of expressions, which are defined using abstraction and application as follows:

$$\texttt{case } e \texttt{ of } pat_1 \texttt{ => } e_1 \texttt{ |...| } pat_n \texttt{ => } e_n \equiv \texttt{(fn } pat_1 \texttt{ => } e_1 \texttt{ |...| } pat_n \texttt{ => } e_n\texttt{)(}e\texttt{)}$$

$$\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \equiv \texttt{case } e \texttt{ of true => } e_1 \texttt{ | false => } e_2$$

The following function specifies a stepping command.

```
fun step(h as <h', [_ :  success]>, n) = (h, succeed)
  | step(h as <h', [ ]>, n) = (h, fail)
  | step(h, n) = if n>0 then step(next(h), n-1)
                        else (h, null)
```

The following function specifies the command **run until** $id = n$.

```
fun run_until(h as <h', [_ :  success]>, x, n) = (h, succeed)
  | run_until(h as <h', [ ]>, x, n) = (h, fail)
  | run_until h as <h', [sub :  E ⊢ _, _ ⇒ _ ::_]>, x, n) =
      case evaluation(sub(E), x) of
            val(n1) =>
               if n = n1 then (h,null)
               else run_until(step(h,1), x, n)
          | _ => run_until(step(h,1), x, n)
  | run_until(h, x, n) = run_until(step(h,1), x, n)
```

where the call `evaluation(sub(E), x)` returns the value of `x` in `sub(E)`, or **fail** if `x` is not bound in `sub(E)`. It would be more natural to use the function `next` instead of `step` in the definition of `run_until`. However, when we define the step predicate *nosum* we will have to change the above functions to include this predicate. Using `step` in the definition of `run_until` we factorise the alterations to the function `step`. The following functions will be used in the specification of the *Tiny* commands **show** *id*, **show pos**, and **show res**.

```
fun showid(<h, [sub :  E ⊢ _, _ ⇒ _ ::_]>, y) = evaluation(sub(E), y)
  | showid(<h, [sub :  (E,_) ⟶_L _ ::_]>, y) = evaluation(sub(E), y)
  | showid(<h, [ ]>, _) = fail
```

```
fun showpos(<h, [sub :  _ ⊢ _, p ⇒ _ ::  gs]>) = path(sub(p))
  | showpos(<h, [ ]>) = fail
  | showpos(_) = null
```

```
fun showres( ι(<>,[_ :E⊢e,_ ⇒v],...,[sub:success],<>)) = val(sub(v))
  | showres(<h, [ ]>) = fail
  | showres(_) = null
```

The Specification of *Tiny* with Full Granularity

Now we define the debugger $Tiny = (\Sigma^{Tiny}, \Delta^{Tiny}, \mathrm{A}^{Tiny})$ for an arbitrary $p \in P_{\mathrm{X}^E}(\Omega^{Env})$ based on *Env* and $\mathcal{CS}^{Env}$. The algebraic signature Σ^{Tiny} is the union of Σ^{Env} and the signature defined by the following BNF rules, which define the debugging language and language of results:

$$\begin{array}{lll} \textit{command} & ::= & \textbf{reset} \mid \textbf{step}\ \textit{nat} \mid \textbf{run until}\ \textit{var} = \textit{nat} \mid \\ & & \textbf{show}\ \textit{var} \mid \textbf{show pos} \\ \textit{result} & ::= & \textbf{fail} \mid \textbf{succeed} \mid \textbf{null} \mid \textbf{val}(\textit{val}) \mid \textbf{path}(\textit{path}) \end{array}$$

where *nat*, *var*, and *val* are defined in Σ^{Env}. The algebra A^{Tiny} agrees with A^{Env} on the interpretation of the symbols of Σ^{Env}, i.e., $\mathrm{A}^{Tiny}/\Sigma^{Env} = \mathrm{A}^{Env}$, and gives the term algebra interpretation for the symbols defined by the above grammar.

The next step is to define the debugging commands. Let Δ^{Tiny} be as follows:

$$(\mathrm{EH}, g^{Tiny}, T(\Sigma^{Tiny})_{command} \times T(\Sigma^{Tiny})_{result}, \longrightarrow_{Tiny}, \mathrm{I}^{Tiny})$$

where EH is the set of debugging histories of p in $\mathcal{CS}^{Env}$, g^{Tiny} is the identity on EH, and the initial state I^{Tiny} is defined as follows:

$$\mathrm{I}^{Tiny} = \begin{cases} \texttt{initial}(p) & \text{if } p = E \vdash e, pa \Rightarrow v \text{ and } \mathrm{FV}(\mathit{close}(e, E)) = \{\,\} \\ [\,] & \text{otherwise} \end{cases}$$

Since the evaluation of expressions with free variables always fails, the definition of I^{Tiny} guarantees that we only attempt to debug closed expressions. This is reasonable since in a real debugger we expect such errors to be detected in a static checking phase prior to the debugging phase.

Finally, we define the transition relation $\longrightarrow_{Tiny}$. For all $dh \in \mathrm{EH}$ and $d \in T(\Sigma^{Tiny})_{command}$ the transition:

$$dh \xrightarrow{d,r}_{Tiny} dh'$$

is defined as follows:

$d =$ **reset**	$r =$ **null** and $dh' = \mathrm{I}^{Tiny}$
$d =$ **step n**	$(dh', r) =$ `step`$(dh,\ n)$
$d =$ **run until** $id = n$	$(dh', r) =$ `run_until`$(dh,\ id,\ n)$
$d =$ **show** id	$r =$ `showid`$(dh,\ id)$ and $dh' = dh$
$d =$ **show pos**	$r =$ `showpos`(dh) and $dh' = dh$
$d =$ **show res**	$r =$ `showres`(dh) and $dh' = dh$

Checking the Requirements of Definition 2

It is easy to check that *Tiny* complies with Definition 2. First, notice that functions `step`, `run_until`, `showid`, `showpos`, and `showres` are total. It is then easy to check that $\longrightarrow_{Tiny}$ obeys the robustness requirement of Definition 2. The functionality requirement of Definition 2 is fulfilled by either the **step** n command.

Specifications of *Tiny* using Step Predicates

Now we define the step predicate *nosum* that was informally discussed in Section 6.4.1. Then we change the `step` function to use *nosum* to enlarge the granularity of the debugging steps of *Tiny*. The predicate *nosum* is formalised as a DSL_{Env} function that returns a boolean value:

```
fun nosum
   (ι([_:_⊢e1₁ + e2₁,_⇒v₁::gs₁],ι^max(SUM₂,SINGLE₂)))
              = false
  | nosum(_) = true
```

The pattern used in the first clause of the definition of the function nosum was discussed in Section 6.5 (page 187). Therefore, *nosum*(*dh*) holds if and only if the result of nosum(*dh*) is true. With this definition of *nosum* it is easy to change step as follows:

```
fun step(h as <h', [_ :  success]> , n) = (h, succeed)
  | step(h as <h', [ ]>, n) = (h, fail)
  | step(h, n) = if n>0 then case next(h) of
                               h1 => if nosum(h1,st)
                                     then step(h1,n-1)
                                     else step(h1,n)
                        else (h,null)
```

If this new definition of step overrides the definition of page 188, no more changes are necessary in the specification of *Tiny* to include a coarser debugging step granularity. Moreover, since run_until is defined using step, we only have to include the nosum function in step.

The next example defines another step predicate. This time we want to hide the states related to the lookup of a variable in an environment, and the state related to the application of the **plus** function. The latter state has a formula $E \vdash e, \langle\,\rangle \Rightarrow v$ on the top of the goal stack, since this is the form of the third premiss of the rule that defines "+" expressions in page 172. Therefore, the pattern $\_ \vdash \_,\ \langle\,\rangle \Rightarrow \_$ will be used to describe the above formula. The function nolookup, defined below, is used to define a step predicate *nolookup* with this meaning.

```
fun nolookup(<_, [_ :  (E,e) ⟶E v ::  _]>) = false
  | nolookup(<_, [_ :  _ ⊢ _, ⟨⟩ ⇒ _ ::_]>) = false
  | nolookup(_, _) = true
```

Here the first clause describes the lookup of variables and the second describes the application of **plus** as described above. Therefore, *nolookup*(*dh*) holds if and only if the result of `nolookup`(*dh*, *st*) is `true`. To include the `nolookup` function in the `step` command we just need to change the `step` function defined above, replacing the calls to `nosum` by the corresponding calls to `nolookup`. In the example of Section 7.4 we assume that the function `step` uses the `nolookup` function.

This example illustrates the flexibility we achieve by using step predicates in the definition of the granularity of the debugging steps. We can imagine a debugger with a set of step predicates and a command that selects which one is to be used during the evaluation. Therefore, the same debugging session can use more than one step granularity, according to the needs of the user.

6.7 Summary and Conclusions

In this chapter we have studied the problem of how to give formal specifications of debuggers. This problem was addressed in three stages. First, we developed a formal definition of the class of Interpreter-debuggers. Second, we studied two problems that are common to the design of most debuggers: the granularity of debugging steps and the reference to sub-programs. Finally, we defined a notation to assist in the specification of debuggers.

As far as we are aware, an abstract account of debuggers as presented in this section is novel in the literature. DSL differs from the approach of the Animator Generator in that it allows the definition of various debugging commands, whereas the Animator Generator only has a fixed set of commands. However, we have not addressed problems related to user interface, which is a strong point of the Animator Generator.

In the examples, we demonstrated that definitions of concrete debuggers using DSL are abstract. We shall demonstrate in the Chapter 7 that these definitions

are suitable for formal reasoning. It remains to show how other debuggers for other language paradigms can be specified. We believe that debuggers for object oriented languages, logic programming languages, and imperative languages can all be expressed in DSL.

Chapter 7

Debugger Correctness

In Chapter 6 we defined a class of debuggers that we called *Interpreter-debuggers*. An instance of this class is a debugger that uses the CS system defined in Chapter 5 as an interpreter. The specification of a concrete Interpreter-debugger is given in terms of the objects of a Relational Specification of the programming language, resulting in a definition that is easy to understand and reason about. Another positive point of Interpreter-debuggers is that they use the notion of evaluation step of Definition 5.5, which agrees with our intuitive notion of evaluation step. Therefore, we argue that Interpreter-debuggers are more likely to behave according to our intuition than debuggers designed in an ad hoc fashion.

The main drawback of Interpreter-debuggers is that interpretation of programs using the CS interpreter is, in general, too inefficient to be used with large and complex programs. However, those programs are the most likely to have errors, and therefore to need debugging. Therefore, we are faced with the following dichotomy: on the one hand, debuggers based on the CS system are easy to specify, understand, and reason about; on the other hand, there exist a practical need for more efficient evaluation techniques if we hope to use the debuggers with large and complex programs.

A natural solution to this problem is to provide two definitions of a debugger: an Interpreter-debugger to be used as the *specification* of the behaviours of

a debugger, and another definition in which the evaluation of programs is performed more efficiently, to be used as the actual *implementation* of the debugger. For this solution to be sound it is necessary to prove that the specification and the implementation are equivalent, i.e., that the two debuggers have equivalent behaviour.

The means to specify Interpreter-debuggers, according to Definition 6.2, were already studied in Chapter 6. The main objective of this chapter is to define a formal criterion for the equivalence between a two specifications of a debuggers, that is, we shall study the problem of *debugger correctness.*

Furthermore, we also characterise the class of Compiler-debuggers. An instance of this class is a debugger in which a program is compiled and its code runs in a machine instead of being interpreted. We study aspects of the specifications of such debuggers and present a proof of equivalence between an Interpreter-debugger and a Compiler-debugger.

In Section 6.1 we discussed other approaches to formal semantics of debuggers. None of those approaches treats the problem of debugger correctness, nor do they consider Compiler-debuggers and their relationship with Interpreter-debuggers. Therefore, the results of this chapter are novel to this thesis.

7.1 Introduction

In this chapter we shall define what it means for two debuggers to have *equivalent behaviours*; thus, it is essential that we understand what characterises the behaviour of a debugger. According to Definition 6.2, an Interpreter-debugger is characterised by a *monogenic labelled transition system* of the form:

$$(\Gamma, T(\Sigma)_{command} \times T(\Sigma)_{result}, \longrightarrow)$$

The entire behaviour of such a debugger is defined by the transition relation $\longrightarrow: \Gamma \times (T(\Sigma)_{command} \times T(\Sigma)_{result}) \times \Gamma$. This transition has two components that characterise different parts of the behaviour of a debugger. The state com-

ponent of the transition relation characterises how the programs are evaluated, by describing how this evaluation changes the state at each transition. The label component characterises the interaction between the evaluation and the external world. The latter is the *observable behaviour*, in contrast with the internal behaviour characterised by the states.

It is the observable part of the behaviour that matters when we compare two debuggers. We want to consider two debuggers as being equivalent if they have equivalent observable behaviour. This leads to the notion of *observational equivalence* between debuggers. In order to understand this notion of equivalence let us take two debuggers $\mathcal{D}$ and $\mathcal{E}$, and compare their observable behaviour.

Suppose that both debuggers are in the initial state, say $\mathrm{I}^{\mathcal{D}}$ for $\mathcal{D}$ and $\mathrm{I}^{\mathcal{E}}$ for $\mathcal{E}$. Let us experiment with the debuggers by issuing a debugging command d_1 to both of them. On input d_1 the debugger $\mathcal{D}$ moves to a state ds_1 and produces a result r_1. Similarly, $\mathcal{E}$ moves to a state ds_1' and produces a result r_1', which is written:

$$\mathrm{I}^{\mathcal{D}} \xrightarrow{(d_1, r_1)}_{\mathcal{D}} ds_1$$

$$\mathrm{I}^{\mathcal{E}} \xrightarrow{(d_1, r_1')}_{\mathcal{E}} ds_1'$$

and the observable behaviours of these transitions are equivalent if $r_1 = r_1'$. As an initial proposal we will consider the relation $=$ as the syntactical equality between results. Later on in this section we will see how this condition can be relaxed. Now, in the new current states we would like to repeat the experiment with some other debugging command, say d_2. Again, for the debuggers to be equivalent the new transitions must have the following form:

$$ds_1 \xrightarrow{(d_2, r_2)}_{\mathcal{D}} ds_2$$

$$ds_1' \xrightarrow{(d_2, r_2')}_{\mathcal{E}} ds_2'$$

and $r_2 = r_2'$. The two debuggers will be considered observationally equivalent if they behave as described above for all possible experiments. This notion of equivalence certainly agrees with the discussion of the beginning of this section: two debuggers are equivalent if their observable behaviour are equivalent. Ob-

servational equivalence as discussed above has similarities with the notion of bisimularity developed in [Par81] and strong equivalence in [Mil89].

This notion of equivalence guarantees that we cannot distinguish between equivalent debuggers by looking only to the results of the debugging commands; thus, intuitively it is a sufficient criterion for equivalence. However, the requirement that the results of the debugging commands must be equal at each corresponding transition is too strong in general, for it does not allow different representations of the results in each debugger.

A similar problem was encountered in Chapters 3 and 4 when we studied the equivalence of Relational Specifications and compiler correctness. For instance, in the compiler correctness case whenever a program evaluates to a value in the (standard) Relational Specification and to another in a Compilation, we required the two values to be equivalent with respect to an observation signature.

Now let us turn back to the problem of debugger equivalence. Results of debugging commands will be, in most cases, the intermediate results of the evaluation of sub-programs. Therefore, it is natural to require the results of debugging commands to be equivalent with respect to an observation signature. This establishes a strong connection between Observational Equivalence of Relational Specifications, including compiler correctness, and equivalence of debuggers, a connection we believe to be natural since debuggers in our framework are based on a Relational Specification. This connection is formalised in various results in Section 7.2.

The above discussion leads to a notion of observational equivalence between debuggers with respect to an observation signature. This equivalence allows non-observable results of debugging commands to have different representations in each debugger, while observable results must be (syntactically) equal.

Another major objective of this chapter is to extend the framework developed in Chapter 6 to deal with Compiler-debuggers, i.e., debuggers in which programs are compiled into code for some machine and the evaluation of the program is

performed by running this code on the machine. Intuitively, those debuggers are likely to be the most relevant for practical applications and also to yield the most difficult proofs of equivalence.

Our objective is to demonstrate that it is possible to formally define Compiler-debuggers and to carry out proofs of equivalence involving such debuggers. A Compiler-debugger that is proved equivalent to an Interpreter-debugger will have the same observable behaviour as the Interpreter-debugger. We argued that Interpreter-debuggers are more likely to have an intuitive behaviour than debuggers defined in ad hoc fashion. Therefore, the equivalence between an Interpreter-debugger and a Compiler-debugger implies that the latter also has an intuitive behaviour.

7.2 Observational Equivalence between Debuggers

Our initial objective is to formalise the notion of equivalence between debuggers that was informally discussed in the previous section. For this definition we have a similar flexibility as in the case of the equivalence between Relational Specifications (see Section 3.4). We can define a relation parametric on an observation signature. Alternatively, we can include an observation signature as a component of the debugger and then define the equivalence relation on the class of debuggers with the same observation signature. In Section 3.4 we saw that adding the observation signature to the Relational Specifications led to simpler definitions. Therefore, we choose this approach also in the definition of equivalence between debuggers.

Before we state the definition of equivalence between debuggers we must solve the following problem: the only abstract characterisation of debuggers that we have is that of Interpreter-debuggers of Definition 6.2. However, we require alternative designs of debuggers because the evaluation of the programs by an Interpreter-debugger is often too inefficient. Therefore, we need a more general characterisation of debuggers in which the evaluation of programs is not required

to be that of an interpreter based on the $\mathcal{CS}$ system.

A solution to this problem is to define a *meta-class* of debuggers by abstracting from how programs are evaluated. An instance of such a meta-class is a class of debuggers, e.g., Interpreter-debuggers as in Definition 6.2. Such an instance is obtained by supplying the meta-class with an *evaluator* for the programming language, e.g., the $\mathcal{CS}$ system as in the case of Interpreter-debuggers. This leads to the definition of *Debugger-schema* which is a generalisation of Interpreter-debuggers obtained by replacing the $\mathcal{CS}$ system by a monogenic transition system, which describes the evaluation of programs and its transition relation defines the evaluation step used by the Debugger-schema.

In the rest of this section, let $\mathcal{S} = (\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$ be a Relational Specification with Declarative Semantics M, where $\Omega = (\mathrm{S}, \mathrm{F}, \Pi, \pi)$, and $\Omega_{\mathrm{OBS}} = (\mathrm{S}_{\mathrm{OBS}}, \mathrm{F}_{\mathrm{OBS}}, \Pi_{\mathrm{OBS}}, \pi)$. We will use Σ as a name for (S, F) and similarly Σ_{OBS} as a name for $(\mathrm{S}_{\mathrm{OBS}}, \mathrm{F}_{\mathrm{OBS}})$. The following definitions will be given for an arbitrary program formula $p \in P_{\mathrm{X}}(\Omega)$ for any set of meta-variables X. The generalisation to the entire set of program formulae $P_{\mathrm{X}}(\Omega)$ is straightforward, and is presented later in this section.

The definition of a Debugger-schema is given with respect to a monogenic transition system $\mathcal{EV} = (\Gamma_{\mathrm{EV}}, \overset{\mathrm{EV}}{\rightarrow}, \mathrm{I}_{\mathrm{EV}})$, the *evaluator* of p, where $\mathrm{I} \in \Gamma_{\mathrm{EV}}$ is the initial state for p in $\mathcal{EV}$. A particular definition of $\mathcal{EV}$ produces a particular instance of a Debugger-schema. We will see below how to define $\mathcal{EV}$ so that the instance of a Debugger-schema obtained is the class of Interpreter-debuggers.

Definition 1 (Debugger-schema) A Debugger-schema for p based on $\mathcal{S}$ and $\mathcal{EV}$ is a quadruple $(\Sigma_{\mathrm{D}}, \Delta, \mathrm{A}_{\mathrm{D}}, \Sigma\mathrm{D}_{\mathrm{OBS}})$ where Σ_{D} and A_{D} are as in Definition 6.2 and:

- $\Sigma\mathrm{D}_{\mathrm{OBS}} = (\mathrm{SD}_{\mathrm{OBS}}, \mathrm{FD}_{\mathrm{OBS}})$, where $\Sigma\mathrm{D}_{\mathrm{OBS}} \sqsubseteq \Sigma_{\mathrm{D}}$, is an *observation signature* with the following constraints: $\Sigma_{\mathrm{OBS}} \sqsubset \Sigma\mathrm{D}_{\mathrm{OBS}}$, i.e., the algebraic observation signature of $\mathcal{S}$ is included in $\Sigma\mathrm{D}_{\mathrm{OBS}}$; and $\mathit{result} \in \mathrm{SD}_{\mathrm{OBS}}$.

- Δ is a quintuple

$$(\Gamma_{\mathrm{D}}, g, T(\Sigma_{\mathrm{D}})_{command} \times T(\Sigma_{\mathrm{D}})_{result}, \longrightarrow, \mathrm{I})$$

where: Γ_{D}, g, $T(\Sigma_{\mathrm{D}})_{command}$, $T(\Sigma_{\mathrm{D}})_{result}$, and $\longrightarrow$ are as in Definition 6.2 and $\mathrm{I}_{\mathrm{D}} \in \Gamma_{\mathrm{D}}$ is the *initial state*, such that $g(\mathrm{I}_{\mathrm{D}}) = \mathrm{I}_{\mathrm{EV}}$.

such that the following *functionality requirement* holds: for all $ds \in \Gamma_{\mathrm{D}}$ there exists $d \in T(\Sigma_{\mathrm{D}})_{command}$, $r \in T(\Sigma_{\mathrm{D}})_{result}$, and $ds' \in \Gamma_{\mathrm{D}}$ such that $ds \xrightarrow{(d,r)} ds'$ and $g(ds) \overset{\mathrm{EV}^+}{\rightarrow} g(ds')$, where $\overset{\mathrm{EV}^+}{\rightarrow}$ is the transitive closure of $\overset{\mathrm{EV}}{\rightarrow}$. □

Let $\Sigma\mathrm{D}_L \sqsubset \Sigma_{\mathrm{D}}$ be the sub-signature that defines the debugging language, i.e., $T(\Sigma\mathrm{D}_L)_{command} = T(\Sigma_{\mathrm{D}})_{command}$. The class of Debugger-schemas for p with observation signature $\Sigma\mathrm{D}_{\mathrm{OBS}}$ and debugging language defined by $\Sigma\mathrm{D}_L$ is denoted by $Deb_p(\Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$.

As discussed before, the definition of a Debugger-schema characterises a meta-class of debuggers, which can be instantiated into a class of debuggers such as the Interpreter-debuggers. For this instantiation we need to give a definition of the evaluator $\mathcal{EV}$. For instance, let us define $\mathcal{EV}$ to be the quadruple

$$(\mathrm{EH}, \overset{\mathrm{EH}}{\Longrightarrow}, \langle [\theta_0 : p] \rangle)$$

where EH is the set of evaluation histories of p in $\mathcal{S}$ and $\overset{\mathrm{EH}}{\Longrightarrow}$ is the function on EH defined in Section 6.3 (page 160). A Debugger-schema for p with the above $\mathcal{EV}$ characterises the class of Interpreter-debuggers for p based on $\mathcal{S}$. A concrete Interpreter-debugger is an instance of this class obtained by suplying the remaining components of the Debugger-schema.

So far we have generalised our definition of debuggers in order to allow more efficient program evaluation than the interpretation used by Interpreter-debuggers. We can now turn our attention to the problem of defining a notion of equivalence between debuggers. As we discussed in Section 7.1, we will define an equivalence

relation on $Deb_p(\Sigma D_L, \Sigma D_{OBS})$ similar to the notion of bisimulation of [Par81]. Our notion of *Debugger Bisimulation* differs from the (standard) notion of bisimulation in two aspects that will be discussed below.

Informally, a bisimulation is a relation between the states of two transition systems that is *preserved* under the transition relation in the following sense: whenever we start with two related states and make a transition with equivalent labels we reach related states. In the (standard) notion of bisimulation the equivalence of labels is syntactical equality. In Debugger Bisimulation the commands of a transition must be syntactically equal and the results are compared with respect to an observation signature. That is, the results of corresponding transitions are required to be syntactically equal only if they are observable according to the observation signature.

Since the transition systems we are dealing with are Debugger-schemas the following definition could be simplified. For instance, trace equivalence [Mil89, page 204] would also be suitable as an equivalence between debuggers. We chose bisimulation because it allows our approach to be generalised to debuggers for non-deterministic programming language without changing the notion of equivalence. We discuss such a generalisation in Section 8.2.

Finally, we are ready to formalise the notion of observational equivalence between Debugger-schemas with respect to an observation signature as an equivalence relation on $Deb_p(\Sigma D_L, \Sigma D_{OBS})$.

Definition 2 (Debugger Bisimulation) Let $\mathcal{D} = (\Sigma^{\mathcal{D}}, \Delta^{\mathcal{D}}, A^{\mathcal{D}}, \Sigma D_{OBS})$ and $\mathcal{E} = (\Sigma^{\mathcal{E}}, \Delta^{\mathcal{E}}, A^{\mathcal{E}}, \Sigma D_{OBS})$ be Debugger-schemas in $Deb_p(\Sigma D_L, \Sigma D_{OBS})$. The Debugger-schemas $\mathcal{D}$ and $\mathcal{E}$ are observationally equivalent, written $\mathcal{D} \approx \mathcal{E}$, if and only if there exists a relation $\mathcal{B} \in \Gamma^{\mathcal{D}} \times \Gamma^{\mathcal{E}}$ such that $(I^{\mathcal{D}}, I^{\mathcal{E}}) \in \mathcal{B}$ and for all $(ds_{\mathcal{D}}, ds_{\mathcal{E}}) \in \mathcal{B}$ and $d \in T(\Sigma^{\mathcal{D}})_{command}$ the following holds:

1. For all $r_{\mathcal{D}} \in T(\Sigma^{\mathcal{D}})_{result}$ and $ds'_{\mathcal{D}} \in \Gamma^{\mathcal{D}}$, $ds_{\mathcal{D}} \xrightarrow{(d,r_{\mathcal{D}})}_{\mathcal{D}} ds'_{\mathcal{D}}$ implies that there exist $r_{\mathcal{E}} \in T(\Sigma^{\mathcal{E}})_{result}$ and $ds' \in \Gamma^{\mathcal{E}}$ such that $ds_{\mathcal{E}} \xrightarrow{(d,r_{\mathcal{E}})}_{\mathcal{E}} ds'_{\mathcal{E}}$ and both requirements below hold:

(*Bisimulation*) $(ds'_{\mathcal{D}}, ds'_{\mathcal{E}}) \in \mathcal{B}$

(*Observational Equivalence of Results*) If $r_{\mathcal{D}} \in T(\Sigma\mathrm{D}_{\mathrm{OBS}})$ then $r_{\mathcal{D}} = r_{\mathcal{E}}$.

2. And conversely, interchanging $\mathcal{D}$ and $\mathcal{E}$.

A relation $\mathcal{B}$ as above is called a Debugger Bisimulation between $\mathcal{D}$ and $\mathcal{E}$. □

Proposition 1 The relation $\approx$ is an equivalence relation on $Deb_p(\Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$.

Proof (Sketch) For reflexivity, just take $\mathcal{B}$ to be the identity relation on $\Gamma^{\mathcal{D}}$, then it is trivial to check that $\mathcal{D} \approx \mathcal{D}$ for $\mathcal{B}$. For symmetry, suppose that $\mathcal{D} \approx \mathcal{E}$ for a relation $\mathcal{B}$. It is easy to check that $\mathcal{E} \approx \mathcal{D}$ for $\mathcal{B}^{-1}$. For transitivity, suppose that $\mathcal{D}_1 \approx \mathcal{D}_2$ for $\mathcal{B}_1$ and $\mathcal{D}_2 \approx \mathcal{D}_3$ for $\mathcal{B}_2$. It is also simple to prove that $\mathcal{D}_1 \approx \mathcal{D}_3$ for the relation $\mathcal{B}_1 \circ \mathcal{B}_2$. □

The results of the rest of this section depend on the generalisation of a Debugger-schema for p to a Debugger-schema for the entire set $P_{\mathrm{X}}(\Omega)$ of program formulae. First, let $\mathcal{EV} = (\Gamma_{\mathrm{EV}}, \overset{\mathrm{EV}}{\rightarrow}, \{\mathrm{I}_{\mathrm{EV}p}\}_{p \in \mathrm{P_X}(\Omega)})$ be an evaluator for $P_{\mathrm{X}}(\Omega)$, which is the same as in page 200 except that the initial state is replace by a $P_{\mathrm{X}}(\Omega)$-sorted set of initial states. A Debugger-schema for $P_{\mathrm{X}}(\Omega)$ based on $\mathcal{S}$ and $\mathcal{EV}$ is the same as in Definition 1 except that for

$$\Delta = (\Gamma_{\mathrm{D}}, g, T(\Sigma_{\mathrm{D}})_{command} \times T(\Sigma_{\mathrm{D}})_{result}, \longrightarrow, \{\mathrm{I}_p\}_{p \in \mathrm{P_X}(\Omega)})$$

the initial state is replaced by a $P_{\mathrm{X}}(\Omega)$-sorted set of initial states such that for all $p \in \mathrm{P_X}(\Omega)$, $g(\mathrm{I}_p) = \mathrm{I}_{\mathrm{EV}p}$.

The class of debugging-schemas for $P_{\mathrm{X}}(\Omega)$ with observation signature $\Sigma\mathrm{D}_{\mathrm{OBS}}$ and debugging language defined by $\Sigma\mathrm{D}_L$ is denoted by $Deb(\Omega, \Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$. For $\mathcal{D} \in Deb(\Omega, \Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$, we denote by $\mathcal{D}_p$ the Debugger-schema for $p \in \mathrm{P_X}(\Omega)$ that is the same as $\mathcal{D}$ except that $\{\mathrm{I}_p\}_{p \in \mathrm{P_X}(\Omega)}$ is replaced by I_p.

The relation $\approx$ can be naturally extended to an equivalence relation $\approx_\Omega$ on $Deb(\Omega, \Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$: for each $\mathcal{D}, \mathcal{E} \in Deb(\Omega, \Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$, $\mathcal{D} \approx_\Omega \mathcal{E}$ if and only

if for all $p \in P_{\mathrm{X}}(\Omega)$, $\mathcal{D}_p \approx \mathcal{E}_p$. Hereafter, we use $\approx$ for $\approx_\Omega$ whenever $P_{\mathrm{X}}(\Omega)$ is understood in the context. It is clear that if $\mathcal{D} \in \mathit{Deb}(\Omega, \Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$ then $\mathcal{D} \in \mathit{Deb}(\Omega', \Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$ for all $\Omega' \sqsubseteq \Omega$ for which $P_{\mathrm{X}}(\Omega') \neq \{\ \}$.

An important motivation to define an equivalence between debuggers at an abstract level rather than to take an ad hoc approach is that we can prove properties about the abstract definition. In the rest of this section we will illustrate a practical example of such a property: we will prove that the equivalence between two Interpreter-debuggers implies the Observational Equivalence of their underlying Relational Specifications under certain conditions. We call this theorem the *Coherence Theorem.*

Let us discuss the first of necessary condition to establish the Coherence Theorem. Recall that the functionality requirement of Definition 6.2 (page 160) only requires the existence of a command that advances the evaluation. However, this requirement is not enough to guarantee the validity of the Coherence Theorem, as we will emphasise during its proof.

Therefore, we need an extra functionality requirement that guarantees we can use a debugging command to check whether a program terminated successfully. This requirement is formalised by the definition of a *Termination-explicit* Interpreter-debugger (or simply a Termination-explicit debugger).

Definition 3 (Termination-explicit Debugger)
An Interpreter-debugger $(\Sigma_{\mathrm{D}}, \Delta, \mathrm{A}_{\mathrm{D}}, \Sigma\mathrm{D}_{\mathrm{OBS}})$ is *Termination-explicit* if there exits a distinguished observable result **succeed** $\in T(\Sigma\mathrm{D}_{\mathrm{OBS}})_{\mathit{result}}$, and for all $ds \in \Gamma_{\mathrm{D}}$, there exits $d \in T(\Sigma_{\mathrm{D}})_{\mathit{command}}$ and $ds' \in \Gamma_{\mathrm{D}}$ such that $ds \xrightarrow{(d,r)} ds'$ and $r =$ **succeed** if and only if $\mathit{cur}(g(ds)) = [\theta : \varepsilon]$, for some substitution θ. □

The minimal functionality of a Termination-explicit debugger guarantees we can evaluate a program step-by-step, and check whether the evaluation terminated successfully. From Theorems 5.1 and 5.3, this functionality is equivalent to be able to ask whether there exists a satisfying substitution for a program formula.

This functionality is still weaker than the functionality provided by the $\mathcal{CS}$ system since in the case of non-ground program formula, the $\mathcal{CS}$ system not only answers whether there exists a satisfying substitution but also constructs one. However, the minimal functionality of a Termination-explicit debugger is sufficient for the proof of the Coherence Theorem since we will only have to reason about ground program formulae.

Theorem 1 (Coherence) Let $\mathcal{S}, \mathcal{R} \in \mathit{Spec}(\Omega_{\mathrm{OBS}})$ with algebras $\mathrm{A}^{\mathcal{S}}$ and $\mathrm{A}^{\mathcal{R}}$, and Declarative Semantics $\mathrm{M}^{\mathcal{S}}$ and $\mathrm{M}^{\mathcal{R}}$ respectively, where

$$\Omega_{\mathrm{OBS}} = (\mathrm{S}_{\mathrm{OBS}}, \mathrm{F}_{\mathrm{OBS}}, \Pi_{\mathrm{OBS}}, \pi)$$

and π is the only relation symbol in Π_{OBS}. Let $\mathcal{D}^{\mathcal{S}}, \mathcal{D}^{\mathcal{R}} \in \mathit{Deb}(\Omega_{\mathrm{OBS}}, \Sigma\mathrm{D}_L, \Sigma\mathrm{D}_{\mathrm{OBS}})$ be two Termination-explicit debuggers based on $\mathcal{S}$ and $\mathcal{R}$ respectively.
If $\mathrm{A}^{\mathcal{S}}/\Sigma_{\mathrm{OBS}} = \mathrm{A}^{\mathcal{R}}/\Sigma_{\mathrm{OBS}}$, then:

$$\mathcal{D}^{\mathcal{S}} \approx \mathcal{D}^{\mathcal{R}} \qquad \text{implies} \qquad \mathcal{S} \cong \mathcal{R}$$

Proof (Sketch) From Definition 3.4 we must show that:

$$\mathcal{D}^{\mathcal{S}} \approx \mathcal{D}^{\mathcal{R}} \qquad \textit{implies} \qquad \mathrm{M}^{\mathcal{S}}/\Omega_{\mathrm{OBS}} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{M}^{\mathcal{R}}/\Omega_{\mathrm{OBS}}$$

Therefore, we have to prove that Definitions 3.1(1) and 3.1(2) are satisfied. Since from the hypothesis $\mathrm{A}^{\mathcal{S}}/\Sigma_{\mathrm{OBS}} = \mathrm{A}^{\mathcal{R}}/\Sigma_{\mathrm{OBS}}$, Definition 3.1(1) is trivially satisfied. Let us now prove that Definition 3.1(2) is satisfied, which in the context of this theorem is written:

For all $p \in P(\Omega_{\mathrm{OBS}})$:

$$p \in \mathrm{dom}\ \Psi_{\pi}^{\mathrm{A}^{\mathcal{S}}} \text{ if and only if } p \in \mathrm{dom}\ \Psi_{\pi}^{\mathrm{A}^{\mathcal{R}}} \tag{1}$$

and if both sides of the above equivalence are true then

$$\Psi_{\pi}^{\mathrm{A}^{\mathcal{S}}}(p) \in \pi^{\mathrm{M}^{\mathcal{S}}} \text{ if and only if } \Psi_{\pi}^{\mathrm{A}^{\mathcal{R}}}(p) \in \pi^{\mathrm{M}^{\mathcal{R}}} \tag{2}$$

Equivalence (1) holds trivially because $\mathrm{A}^{\mathcal{S}}/\Sigma_{\mathrm{OBS}} = \mathrm{A}^{\mathcal{R}}/\Sigma_{\mathrm{OBS}}$. If both sides of equivalence (1) are false then we do not have anything further to prove. When

both sides are true we must prove that equivalence (2) is satisfied. Let us prove the left to right implication of (2) first.

If $\Psi_{\pi}^{\mathrm{A}^{\mathcal{S}}}(p) \in \pi^{\mathrm{M}^{\mathcal{S}}}$ then, from the Completeness of the Computational Semantics (Theorem 5.3), there exists a $\mathcal{CS}^{\mathcal{S}}$-sequence for p of the form:

$$[\theta_0 : p] \stackrel{\mathrm{CS}}{\Rightarrow}^{n} [\theta : \varepsilon] \qquad n > 0$$

From the functionality requirement of Definition 6.2 there exists a command that advances the evaluation by one transition of $\mathcal{CS}^{\mathcal{S}}$. Therefore, there exists a sequence of debugging interactions in $\mathcal{D}^{\mathcal{S}}$ of the form:

$$\mathrm{I}^{\mathcal{D}^{\mathcal{S}}} \stackrel{(d,r)^n}{\longrightarrow}^{n} ds_{\mathcal{S}}$$

such that $cur(g(ds_{\mathcal{S}})) = [\theta : \varepsilon]$. Since $\mathcal{D}^{\mathcal{S}} \approx \mathcal{D}^{\mathcal{R}}$ then there exits a relation $\mathcal{B} \in \Gamma^{\mathcal{D}^{\mathcal{S}}} \times \Gamma^{\mathcal{D}^{\mathcal{R}}}$ that is a Debugger Bisimulation between $\mathcal{D}^{\mathcal{S}}$ and $\mathcal{D}^{\mathcal{R}}$. Therefore, it is easy to prove that there exists a sequence of debugging interactions in $\mathcal{D}^{\mathcal{R}}$ of the form:

$$\mathrm{I}^{\mathcal{D}^{\mathcal{R}}} \stackrel{(d,r')^n}{\longrightarrow}^{n} ds_{\mathcal{R}}$$

such that $(ds_{\mathcal{S}}, ds_{\mathcal{R}}) \in \mathcal{B}$. From the definition of Termination-explicit debugger (Definition 3) there exists a debugging command $d' \in T(\Sigma \mathrm{D}_L)_{command}$ and $ds'_{\mathcal{S}} \in \Gamma^{\mathcal{D}^{\mathcal{S}}}$ such that:

$$ds_{\mathcal{S}} \stackrel{(d',r)}{\longrightarrow} ds'_{\mathcal{S}}$$

where $r = \mathbf{succeed}$. Since $\mathcal{D}^{\mathcal{S}} \approx \mathcal{D}^{\mathcal{R}}$, there exists a transition in $\mathcal{D}^{\mathcal{R}}$ of the form:

$$ds_{\mathcal{R}} \stackrel{(d,r')}{\longrightarrow} ds'_{\mathcal{R}}$$

and since $\mathbf{succeed} \in T(\Sigma \mathrm{D}_{\mathrm{OBS}})_{result}$, then from the Observational Equivalence requirement of Definition 2(1) it follows that $r' = \mathbf{succeed}$. Since $\mathcal{D}^{\mathcal{R}}$ is Termination-explicit then $cur(g(ds'_{\mathcal{R}})) = [\theta' : \varepsilon]$, for some θ'. Therefore, there exists a $\mathcal{CS}^{\mathcal{R}}$-sequence for p, and from Soundness of the Computational Semantics (Theorem 5.1), it follows that $\Psi_{\pi}^{\mathrm{A}^{\mathcal{R}}}(p) \in \pi^{\mathrm{M}^{\mathcal{R}}}$.

The proof of the right to left implication in (2) follows similarly and is omitted. This finishes the proof of the Coherence Theorem. □

7.3 Compiler-debuggers

In this section we characterise the class of Compiler-debuggers and study aspects that are specific to the design of debuggers in this class. Our objective is to define a framework for the design of Compiler-debuggers along the lines of the framework defined in Chapter 6 for Interpreter-debuggers. As far as we are aware of, Compiler-debuggers have not been addressed in a formal and abstract way in the literature. Therefore, the results in this section are important contributions of this thesis.

It becomes simpler to understand what characterises a Compiler-debugger by making a parallel with the notion of Compilation defined in Chapter 4. In a Compilation programs are evaluated in a three stage process. First, a program is *compiled* into machine code. Second, the code is loaded and *executed* on the machine. Finally, the result is *unloaded* from the machine whenever a successful evaluation is achieved.

In a Compiler-debugger programs are debugged in a process with similar stages. Suppose that p is the program to be debugged. First, p is compiled into code for some (abstract) machine and this code is loaded into the machine for execution. The execution proceeds from an initial state until a halting point is reached; at such a point the control returns to the debugger, and we use the debugging commands to obtain information about the current state of the evaluation. The execution can then be resumed either from the current state, or from a previous state in the execution, until another halting point is reached; again, the control returns to the debugger. This process repeats until either the program reaches a final state in the execution (success or failure) or we find an error and finish the debugging session.

A Compiler-debugger is an instance of a Debugger-schema obtained by using an evaluator based on an abstract machine for $\mathcal{EV}$. Furthermore, a compiler is used to construct the initial state of $\mathcal{EV}$. Therefore, the results of Section 7.2 are applicable to Compiler-debuggers. Our objective in the rest of this section is to

show how to design Compiler-debuggers and prove them correct. In Section 7.3.1 we study various aspects that are specific to the design of Compiler-debuggers. We illustrate these aspects by gradually building a compiler for the language *Fun*, based on the compiler developed in Chapter 4. The code generated by this compiler contains extra code that is used for debugging purposes.

In Section 7.3.2 we use the compiler developed in Section 7.3.1 in the specification of a Compiler-debugger for *Fun* programs. Finally, in Section 7.4 we prove that this Compiler-debugger is equivalent to the Interpreter-debugger *Tiny* defined in Section 6.6.

Recall that in Section 4.3 we argued that the starting point for compiler design in our framework is a Relational Specification of the programming language. Similarly, we now advocate that the starting point for the design of a Compiler-debugger is the specification of an Interpreter-debugger. An Interpreter-debugger will serve as a guide for the design as well as the reference point to establish the correctness of the Compiler-debugger.

Therefore, the first step in the design of a Compiler-debugger is the specification of the Interpreter-debugger that will serve as our reference. In Section 6.6, we defined the Interpreter-debugger *Tiny* for the language *Fun*. We now have to add an observation signature to *Tiny* to specify the observable results of debugging commands. The signature $\Sigma_{\mathrm{OBS}}^{Tiny}$ is defined by the following BNF rules together with the definition of *path* terms presented in Section 6.4.2:

$$
\begin{array}{lll}
\mathit{result} & ::= & \mathbf{fail} \mid \mathbf{succeed} \mid \mathbf{null} \mid \mathbf{val}(\mathit{val}) \mid \mathbf{path}(\mathit{path}) \\
\mathit{var} & ::= & \mathbf{x} \mid \mathbf{y} \mid \cdots \\
\mathit{nat} & ::= & 0 \mid 1 \mid \cdots \\
\mathit{closure} & ::= & \\
\mathit{val} & ::= & \mathit{nat}
\end{array}
$$

Therefore, as far as the results of *Fun* expressions are concerned, *nat* and *path* results are visible, whereas *closure* results are not, since there are no observable terms of sort *closure*.

Let $Tiny = (\Sigma^{Tiny}, \Delta^{Tiny}, \mathrm{A}^{Tiny}, \Sigma^{Tiny}_{\mathrm{OBS}})$ be the new definition of *Tiny* where $(\Sigma^{Tiny}, \Delta^{Tiny}, \mathrm{A}^{Tiny})$ was defined in Section 6.6. For this definition of *Tiny* we will use the definition of the **step** command that uses the step predicate *nolookup* defined in Section 6.6.

7.3.1 Aspects of the Specification of Compiler-debuggers

In this section we analyse two important differences between Compiler-debuggers and Interpreter-debuggers which can cause problems in establishing the equivalence between a Compiler-debugger and an Interpreter-debugger. Our objective in this section is to identify those problems and propose generic solutions for them at the level of an abstract Compiler-debugger.

The first of these differences is the *notion of evaluation step.* While the steps of an Interpreter-debugger are the evaluation steps of Definition 5.5, in the Compiler-debugger the steps are those of the abstract machine on which the code of the program is executed. The second difference is the *information about the evaluation* that is available to the debuggers. An Interpreter-debugger has access to potentially the entire evaluation history of a program, whereas a Compiler-debugger has access to a single machine state, and to the information about the evaluation that is encoded in this state.

We start by studying the problems that arise from the difference between the two notions of evaluation step. The steps of the interpreter and the machine steps may differ in two aspects that are relevant for debugging purposes: the *granularity of the steps*, and the *order of the evaluation of sub-programs.*

The Granularity of the Debugging Step

We treat first the problems that arise from the difference between the granularity of the steps. To illustrate these problems, let us consider the *Fun* expression

$$\textbf{let x} = 1 + 2 \textbf{ in x}$$

and compare its evaluation in an empty environment using the $\mathcal{CS}^{Env}$ system and the abstract machine CAM. The CAM code for the expression, denote by c, according to the compiler of Example 4.3 is as follows:

$$\textbf{push} \cdot \textbf{push} \cdot \textbf{quote}(1) \cdot \textbf{swap} \cdot \textbf{quote}(2) \cdot \textbf{add} \cdot \textbf{cons} \cdot \textbf{cdr} \cdot \varepsilon_C$$

Below we present the evaluation of the formula $\varepsilon_E \vdash \textbf{let } x = 1+2 \textbf{ in } x, \langle \underline{0} \rangle \Rightarrow v$, using $\mathcal{CS}^{Env}$, in which we omit the substitutions and elided some goals to simplify the presentation. In this evaluation, the hidden states according to *nolookup* are presented in boxes and the hidden transitions are not numbered. The numbered steps are the ones produced by the debugger *Tiny*.

$[\varepsilon_E \vdash \textbf{let x} = 1+2 \textbf{ in x}, \langle \underline{0} \rangle \Rightarrow v] \overset{CS}{\Rightarrow}$ (1)

$[\varepsilon_E \vdash 1+2, \langle \underline{0}, \underline{2} \rangle \Rightarrow v_1 :: x \mapsto v_1 \cdot \varepsilon_E \vdash x, \langle \underline{0}, \underline{3} \rangle \Rightarrow v] \overset{CS}{\Rightarrow}$ (2)

$[\varepsilon_E \vdash 1, \langle \underline{0}, \underline{2}, \underline{1} \rangle \Rightarrow v_1' :: \varepsilon_E \vdash 2, \langle \underline{0}, \underline{2}, \underline{2} \rangle \Rightarrow v_2' :: \ldots] \overset{CS}{\Rightarrow}$ (3)

$[\varepsilon_E \vdash 2, \langle \underline{0}, \underline{2}, \underline{2} \rangle \Rightarrow v_2' :: \varepsilon_E \vdash \textbf{plus}(v_1', v_2'), \langle\,\rangle \Rightarrow v_1, :: \ldots] \overset{CS}{\Rightarrow}$ (4)

$\boxed{[\varepsilon_E \vdash \textbf{plus}(v_1', v_2'), \langle\,\rangle \Rightarrow v_1 :: x \mapsto v_1 \cdot \varepsilon_E \vdash x, \langle \underline{0}, \underline{3} \rangle \Rightarrow v]} \overset{CS}{\Rightarrow}$

$[x \mapsto 3 \cdot \varepsilon_E \vdash x, \langle \underline{0}, \underline{3} \rangle \Rightarrow v] \overset{CS}{\Rightarrow}$ (5)

$\boxed{[(x \mapsto 3 \cdot \varepsilon_E, x) \longrightarrow_L v]} \overset{CS}{\Rightarrow}$

[success]

The first hidden state refers to the evaluation of **plus** function, which is hidden because it matches the pattern $[\_ : \_ \vdash \_, \langle\,\rangle \Rightarrow \_ :: \_]$ in the definition of the visibility predicate *nolookup*. Now, we show the steps of the evaluation of the code for the above expression, on the CAM.

$$((\,)\cdot\varepsilon_S, \mathbf{push}\cdot\mathbf{push}\cdot\mathbf{quote}(1)\cdot\mathbf{swap}\cdot\mathbf{quote}(2)\cdot\mathbf{add}\cdot\ldots) \xrightarrow{\text{CAM}} \quad (1)$$

$$((\,)\cdot(\,)\cdot\varepsilon_S, \mathbf{push}\cdot\mathbf{quote}(1)\cdot\mathbf{swap}\cdot\mathbf{quote}(2)\cdot\mathbf{add}\cdot\mathbf{cons}\ldots) \xrightarrow{\text{CAM}} \quad (2)$$

$$((\,)\cdot(\,)\cdot(\,)\cdot\varepsilon_S, \mathbf{quote}(1)\cdot\mathbf{swap}\cdot\mathbf{quote}(2)\cdot\mathbf{add}\cdot\mathbf{cons}\cdot\mathbf{cdr}\cdot\varepsilon_C) \xrightarrow{\text{CAM}} \quad (3)$$

$$\underline{(1\cdot(\,)\cdot(\,)\cdot\varepsilon_S, \mathbf{swap}\cdot\mathbf{quote}(2)\cdot\mathbf{add}\cdot\mathbf{cons}\cdot\mathbf{cdr}\cdot\varepsilon_C)} \xrightarrow{\text{CAM}} \quad (4)$$

$$((\,)\cdot 1\cdot(\,)\cdot\varepsilon_S, \mathbf{quote}(2)\cdot\mathbf{add}\cdot\mathbf{cons}\cdot\mathbf{cdr}\cdot\varepsilon_C) \xrightarrow{\text{CAM}} \quad (5)$$

$$\underline{(2\cdot 1\cdot(\,)\cdot\varepsilon_S, \mathbf{add}\cdot\mathbf{cons}\cdot\mathbf{cdr}\cdot\varepsilon_C)} \xrightarrow{\text{CAM}} \quad (6)$$

$$\underline{(\mathbf{plus}(1,2)\cdot(\,)\cdot\varepsilon_S, \mathbf{cons}\cdot\mathbf{cdr}\cdot\varepsilon_C)} \xrightarrow{\text{CAM}} \quad (7)$$

$$(((\,),3), \mathbf{cdr}\cdot\varepsilon_C) \xrightarrow{\text{CAM}} \quad (8)$$

$$(3\cdot\varepsilon_S, \varepsilon_C)$$

As we can easily check, there is no one-to-one correspondence between the $\mathcal{CS}^{Env}$ steps and the machine steps. In particular, there are more machine steps than there are steps of the interpreter, and we expect that this will often be the case. Furthermore, the use of the step predicate to hide parts of the evaluation increased the gap between the two sequence of steps. However, for some sub-programs we may also expect that optimisations can reduce the number of machine steps; thus, for these sub-programs the interpreter may have more steps than the evaluation of the code on the machine.

Our next objective is to show how to modify the machine steps to establish a one-to-one correspondence between the steps of the two evaluation of the programs. This correspondence is a necessary stage in the design of a Compiler-debugger, whenever we intend this debugger to have the same observable behaviour of an Interpreter-debugger. We first discuss a solution for the example given above, and then show how it can be generalise to an arbitrary abstract machine.

A solution to the particular problem treated above is to skip the steps (4), (6), and (7) of the CAM evaluation by hidding the underlined states. The remaining

five machine steps are on a one-to-one correspondence with the visible $\mathcal{CS}^{Env}$ steps. Furthermore, there is an important invariant preserved by each pair of corresponding states: the CAM state has at the top of the stack the machine environment corresponding to the source language environment that is in the top of the goal stack of the $\mathcal{CS}^{Env}$ state. This can be checked, for instance, by comparing the two states below:

$$[x \mapsto 3 \cdot \varepsilon_E \vdash x, \langle \underline{0}, \underline{3} \rangle \Rightarrow v] \qquad (((\,),3), \mathbf{cdr} \cdot \varepsilon_C)$$

where the machine environment $((\,),3)$ corresponds to the source language environment $x \mapsto 3 \cdot \varepsilon_E$. This invariant will be important in the definition of debugging commands over the machine states.

The generic solution we are looking for is a mechanism to hide machine steps. We propose a mechanism similar to the concept of step predicates defined in Section 6.4.1, in this case applied to the steps of an abstract machine. This solution combines the notions of *break-point predicate* and *debugging machine*; we first give the definition of a break-point predicate.

Definition 4 (Break-point Predicate) Let $\mathcal{M} = (\Gamma_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{T}_{\mathcal{M}})$ be an abstract machine. A *break-point predicate* in $\mathcal{M}$ is a total predicate on $\Gamma_{\mathcal{M}}$.
Let $st \rightarrow_{\mathcal{M}} st'$ and BP be a break-point predicate in $\mathcal{M}$; st is *visible* if $\mathrm{BP}(st)$ holds, and it is *hidden* otherwise. We denote by $\mathcal{VS}_{\mathcal{M}}(\mathrm{BP})$ the set:

$$\mathcal{T}_{\mathcal{M}} \cup \{st : \mathrm{BP}(st)\} \qquad \square$$

With the above definitions we are ready to define the concept of a Debugging Machine.

Definition 5 (Debugging Machine) Let $\mathcal{M} = (\Gamma_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{T}_{\mathcal{M}})$ be an abstract machine and BP be a break-point predicate in $\mathcal{M}$. A debugging machine based on $\mathcal{M}$ with respect to BP is a terminal transition system $(\mathcal{VS}_{\mathcal{M}}(\mathrm{BP}), \longrightarrow_{\mathcal{DM}}, \mathcal{T}_{\mathcal{M}})$ where the transition relation $\longrightarrow_{\mathcal{DM}} \subseteq \rightarrow^*_{\mathcal{M}}$ is defined as follows:

$st \longrightarrow_{\mathcal{DM}} st'$ if and only if there exists $st_1, \ldots, st_n \in \Gamma_{\mathcal{M}}$, for $n \geq 0$, such that

$$st = st_0 \rightarrow_{\mathcal{M}} st_1 \rightarrow_{\mathcal{M}} \ldots \rightarrow_{\mathcal{M}} st_n \rightarrow_{\mathcal{M}} st_{n+1} = st'$$

and st, $st' \in \mathcal{VS}_{\mathcal{M}}(\mathrm{BP})$ and $st_1, \ldots, st_n \notin \mathcal{VS}_{\mathcal{M}}(\mathrm{BP})$. □

Given a debugging machine $\mathcal{DM}$, a Compilation can be defined such that the execution of the code of a program in $\mathcal{DM}$ yields evaluation steps that match the $\mathcal{CS}$ steps of the evaluation of the program. The next example constructs a debugging machine based on the CAM of Example 4.1, with the objective of illustrating the above ideas. Moreover, this machine will be used in the definition of a complete Compiler-debugger in the next section.

Example 1 In this example we define a debugging machine based on an extension of the CAM of Example 4.1. We first present this extension.

We extend the CAM with the instructions **brk**, **quoteid**, **quotep**, and **pop**: **brk** is a break-point instruction whose execution causes the machine to halt; **quoteid** has an identifier as a parameter, and its execution loads the parameter on the top of the stack; **quotep** has an abstract syntax path (as defined in Section 6.4.2) as a parameter and its execution also loads the parameter on the top of the stack; and the **pop** instruction discharges the element on the top of the stack.

The instructions **quoteid** and **quotep** require the CAM values to be extended with identifiers and abstract-syntax path. An abstract-syntax path is just a sequence of natural numbers, and is a term of sort *path* in the Relational Rules. The instructions **quoteid**, **quotep**, and **pop** can be simulated by sequences of CAM instructions; thus the only instruction that is necessary is **brk**. However, the use of **quoteid**, **quotep**, and **pop** makes the specification of the Compiler-debugger simpler to understand, and also simplifies the proof of correctness.

Although the signature of the new CAM is based on the signature Ω^{Cam} of Example 4.1 we define $\Omega^{DCam} = (\mathrm{S}^{DCam}, \mathrm{F}^{DCam}, \Pi^{DCam}, \overset{\mathrm{DCAM}}{\Longrightarrow})$ in full to make this

example self contained. The sets S^{DCam} and F^{DCam} are defined by the following BNF:

$$
\begin{array}{lcl}
\mathit{state} & ::= & (\mathit{stack}, \mathit{code}) \\
\mathit{stack} & ::= & \varepsilon_S \mid \mathit{val} \cdot \mathit{stack} \\
\mathit{code} & ::= & \varepsilon_C \mid \mathit{inst} \cdot \mathit{code} \\
\mathit{val} & ::= & \mathit{nat} \mid \mathit{funval} \mid (\mathit{val}, \mathit{val}) \mid () \\
\mathit{nat} & ::= & \mathbf{plus}(\mathit{nat}, \mathit{nat}) \mid 0 \mid 1 \mid \cdots \\
\mathit{funval} & ::= & [\mathit{val}, \mathit{code}] \\
\mathit{inst} & ::= & \mathbf{quote}(\mathit{nat}) \mid \mathbf{push} \mid \mathbf{car} \mid \mathbf{cdr} \mid \mathbf{cons} \mid \mathbf{swap} \\
 & & \mid \mathbf{cur}(\mathit{code}) \mid \mathbf{app} \mid \mathbf{add} \mid \mathbf{quoteid}(\mathit{var}) \mid \mathbf{brk} \\
 & & \mid \mathbf{quotep}(\mathit{path}) \mid \mathbf{pop}
\end{array}
$$

The meta-variables used in the following rules are those in X^E, to which we add the set X^E_{path} generated by *pa*. The Relational Rules below extend the relation $\xrightarrow{\text{CAM}}$ with the definition of the new instructions discussed above. These rules are additions to the rules 4.1(1) to 4.1(9).

A Extension to the CAM $\boxed{\mathit{state} \xrightarrow{\text{CAM}} \mathit{state}}$

$$\frac{}{(v \cdot S, \mathbf{quoteid}(id) \cdot c) \xrightarrow{\text{CAM}} (id \cdot S, c)} \tag{10}$$

$$\frac{}{(S, \mathbf{quotep}(pa) \cdot c) \xrightarrow{\text{CAM}} (pa \cdot S, c)} \tag{11}$$

$$\frac{}{(v \cdot S, \mathbf{pop} \cdot c) \xrightarrow{\text{CAM}} (S, c)} \tag{12}$$

$$\frac{}{(S, \mathbf{brk} \cdot c) \xrightarrow{\text{CAM}} (S, c)} \tag{13}$$

Comment

In rule (11), the definition of the instruction $\mathbf{quotep}(pa)$ adds *pa* to the top of the stack instead of replacing the top of the stack by *pa*. The reason is that the

path stored on the top of the stack will be only used for debugging purposes, and will be discharged by a **pop** instruction such that the normal evaluation of the program may continue on the original stack. If **quotep**(pa) replaced the top element of the stack by pa than we would need an extra **push** instruction to save the original top before each **quotep** when compiling *Fun* expressions; by defining **quotep** as above we avoid this extra **push** instruction in the compilation.

The transition $\stackrel{\text{CAM}}{\longrightarrow}$ defines the abstract machine CAM = $(\Gamma_{\text{CAM}}, \stackrel{\text{CAM}}{\longrightarrow}, \mathcal{T}_{\text{CAM}})$ where $\Gamma_{\text{CAM}} = T(\Sigma^{DCam})_{state}$. Now, let BP be a total predicate on Γ_{CAM} defined as follows: for all $S \in T(\Sigma^{DCam})_{stack}$, $op \in T(\Sigma^{DCam})_{inst}$, and $c \in T(\Sigma^{DCam})_{code}$:

$$\text{BP}((S, op \cdot c)) \text{ is true if and only if } op = \textbf{brk}$$

This defines a debugging machine DCAM = $(\Gamma_{\text{DCAM}}, \stackrel{\text{DCAM}}{\Longrightarrow}, \mathcal{T}_{\text{CAM}})$ where the set of states $\Gamma_{\text{DCAM}} = \mathcal{VS}(\text{BP})$ and $\stackrel{\text{DCAM}}{\Longrightarrow}$ is defined as in Definition 5. □

The next example changes the code generate by the compiler of Example 4.2. The objective is to establish a correspondence between the $\mathcal{CS}^{Env}$ steps and DCAM steps of the evaluation of expressions. This correspondence is achieved by the correct placement of **brk** instructions on the code of the expressions.

Example 2 This example defines a compiler for the language *Fun* into code for the DCAM by a Relational Specification based on the compiler *Trans* defined in Example 4.2. This compiler is defined by a relation $(comp\_env, exp) \stackrel{DComp}{\longrightarrow} state$, which uses $(comp\_env, var) \longrightarrow_{LC} code$ and $env \longrightarrow_{E} (comp\_env, val)$ of *Trans* as auxiliary relations.

The signature for this compiler is the union of the signature of the compiler of Example 4.2 and the signature Ω^{DCam} defined in Example 1. Moreover, the meta-variables used in the Relational Rules are those defined in X^{E}.

The set of Relational Rules defines the code generation for expressions, in which break-point instructions are inserted in order to establish a correspondence between the machine steps and the $\mathcal{CS}^{Env}$ steps. As in Example 4.2, we write a

sequence $op_1 \cdot \ldots \cdot op_n \cdot \varepsilon_C$ using the usual sequence notation $\langle op_1, \ldots, op_n \rangle$, and denote the concatenation of two code sequences c and c' by $c@c'$.

Code Generator $\boxed{comp\_env \vdash exp \stackrel{DComp}{\longrightarrow} code}$

$$\frac{}{CE \vdash n \stackrel{DComp}{\longrightarrow} \langle \mathbf{brk}, \mathbf{quote}(n) \rangle} \quad (1)$$

$$\frac{(CE, id) \longrightarrow_{LC} c}{CE \vdash id \stackrel{DComp}{\longrightarrow} \langle \mathbf{brk} \rangle @ c} \quad (2)$$

$$\frac{(CE, id) \vdash e \stackrel{DComp}{\longrightarrow} c}{CE \vdash \mathbf{fn}\ id\ .\ e \stackrel{DComp}{\longrightarrow} \langle \mathbf{brk}, \mathbf{cur}(c) \rangle} \quad (3)$$

$$\frac{CE \vdash e_1 \stackrel{DComp}{\longrightarrow} c_1 \qquad CE \vdash e_2 \stackrel{DComp}{\longrightarrow} c_2}{CE \vdash e_1 + e_2 \stackrel{DComp}{\longrightarrow} \langle \mathbf{brk}, \mathbf{push} \rangle @ c_1 @ \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{add} \rangle} \quad (4)$$

$$\frac{CE \vdash e_1 \stackrel{DComp}{\longrightarrow} c_1 \qquad CE \vdash e_2 \stackrel{DComp}{\longrightarrow} c_2}{CE \vdash \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \stackrel{DComp}{\longrightarrow} \langle \mathbf{brk}, \mathbf{push} \rangle @ c_1 @ \langle \mathbf{cons} \rangle @ c_2} \quad (5)$$

$$\frac{CE \vdash e_1 \stackrel{DComp}{\longrightarrow} c_1 \qquad CE \vdash e_2 \stackrel{DComp}{\longrightarrow} c_2}{CE \vdash e_1(e_2) \stackrel{DComp}{\longrightarrow} \langle \mathbf{brk}, \mathbf{push} \rangle @ c_1 @ \langle \mathbf{swap} \rangle @ c_2 @ \langle \mathbf{cons}, \mathbf{app} \rangle} \quad (6)$$

Compiler $\boxed{(env, exp) \stackrel{DComp}{\longrightarrow} state}$

$$\frac{E \longrightarrow_E (CE, v) \qquad CE \vdash e \stackrel{DComp}{\longrightarrow} c}{(E, e) \stackrel{DComp}{\longrightarrow} (v \cdot \varepsilon_S, c)} \quad (7)$$

This finishes Example 2. □

The above compiler generates the following DCAM code for **let** $x = 1 + 2$ **in** x in the empty environment:

$$\langle \mathbf{brk}, \mathbf{push}, \mathbf{brk}, \mathbf{push}, \mathbf{brk}, \mathbf{quote}(1), \mathbf{swap}, \mathbf{brk}, \mathbf{quote}(2), \mathbf{add}, \mathbf{cons}, \mathbf{brk}, \mathbf{cdr} \rangle$$

We denote the above code by c'. If we execute this code on the DCAM machine, starting at the state $((\,) \cdot \varepsilon_S, c')$, we obtain the following DCAM steps:

$$((\,)\cdot\varepsilon_S, \mathbf{brk}\cdot\mathbf{push}\cdot\mathbf{brk}\cdot\mathbf{push}\cdot\mathbf{brk}\cdot\mathbf{quote}(1)\cdot\ldots) \overset{\text{DCAM}}{\Longrightarrow} \quad (1)$$

$$((\,)\cdot(\,)\cdot\varepsilon_S, \mathbf{brk}\cdot\mathbf{push}\cdot\mathbf{brk}\cdot\mathbf{quote}(1)\cdot\mathbf{swap}\cdot\ldots) \overset{\text{DCAM}}{\Longrightarrow} \quad (2)$$

$$((\,)\cdot(\,)\cdot(\,)\cdot\varepsilon_S, \mathbf{brk}\cdot\mathbf{quote}(1)\cdot\mathbf{swap}\cdot\mathbf{brk}\cdot\mathbf{quote}(2)\cdot\ldots) \overset{\text{DCAM}}{\Longrightarrow} \quad (3)$$

$$((\,)\cdot 1\cdot(\,)\cdot\varepsilon_S, \mathbf{brk}\cdot\mathbf{quote}(2)\cdot\mathbf{add}\cdot\mathbf{cons}\cdot\mathbf{brk}\cdot\mathbf{cdr}\cdot\varepsilon_C) \overset{\text{DCAM}}{\Longrightarrow} \quad (4)$$

$$(((\,),3), \mathbf{brk}\cdot\mathbf{cdr}\cdot\varepsilon_C) \overset{\text{DCAM}}{\Longrightarrow} \quad (5)$$

$$(3\cdot\varepsilon_S, \varepsilon_C)$$

The above steps nnd states are in a one to one correspondence with the CS^{Env} steps and states of page 210, after hiding from that history the states in the boxes. Therefore, using the concept of a debugging machine we achieve the one-to-one correspondence between the machine and interpreter evaluation steps.

The Evaluation Order of Sub-programs

Let us consider other problems related to the difference between the two notions of evaluation step. The point we want to emphasise is that compilation techniques, and in particular optimisation, may break any correspondence between the code and the original program; thus the evaluation of the code will produce steps that not only have different granularity, but in which the order of the evaluation of the sub-programs may be different from the steps of the interpreter.

To illustrate this problem, let us consider the code for the sub-expression $1+2$ in the expression **let** $x = 1 + 2$ **in** x:

$$\langle \underbrace{\mathbf{brk},\mathbf{quote}(1)}_{c_1}, \mathbf{swap}, \underbrace{\mathbf{brk},\mathbf{quote}(2)}_{c_2}, \mathbf{add}\rangle$$

In this code sequence, the sub-sequences c_1 and c_2 are the code of the sub-expressions 1 and 2 respectively. This correspondence between the source program and its machine code greatly helps in establishing a correspondence between the steps of the interpretation of the program and the steps of the evaluation of its code on the machine.

However, there are at least two ways of replacing this code sequence by another sequence that performs the same computation, i.e, produces the constant value 3 on the top of the machine stack, and whose evaluation steps do not correspond to the $\mathcal{CS}^{Env}$ steps. The first equivalent sequence is the single instruction **quote**(3); a reasonably simple optimiser should be able to perform this optimisation. Because the sum operation is commutative, another equivalent sequence is the code in which the relative order between c_1 and c_2 is changed, resulting in the following sequence:

$$\langle \textbf{brk},\textbf{quote}(2),\textbf{swap},\textbf{brk},\textbf{quote}(1),\textbf{add}\rangle \qquad (*)$$

The final evaluation of each of the three sequences of code produces 3 on the top of the stack. However, the evaluation steps resulting from the evaluations of the two alternative sequences do not match the $\mathcal{CS}^{Env}$ steps.

A possible solution to reestablish the correspondence between the steps is to define step predicates at the interpreter level that regard the evaluation of the expression $1 + 2$ as a single step, hiding its sub-evaluations. Then, the correct placement of **brk** instructions in any of the above code sequences may also produce the machine steps in which the evaluation of $1 + 2$ is considered as a single step. Therefore, the correspondence between the steps is established again. For instance, for the code sequence $(*)$ the following placement of **brk** instructions will yield a single DCAM step for the evaluation of the entire sequence:

$$\langle \textbf{brk},\textbf{quote}(2),\textbf{swap},\textbf{quote}(1),\textbf{add}\rangle$$

The problem of debugging in the presence of optimised code is a difficult one mainly because the optimisations may break the correspondence between the code and the source program, as discussed above. This problem has been addressed by several authors in the literature [Hen82,ZJ91]. The basic objective of those approaches is to obtain a debugger with "expected behaviour" even in the presence of optimised code. Their main problem is that, due to the absence of a formal definition of evaluation step, the notion of "expected behaviour" cannot

be precisely defined. Therefore, it is difficult to conclude whether their objectives were achieved, since those objectives were not clearly defined.

In the framework of this thesis we do have a formal notion of evaluation step that agrees with our intuition; thus, we have the means to give a precise definition of "expected behaviour". This behaviour can be specified in a concrete Interpreter-debugger, and step predicates can be used to refine this behaviour with coarser or finer debugging steps as required by the user. Then, break-point predicates can be used to make the behaviour of the Compiler-debugger to match the behaviour of the Interpreter-debugger, so that we can prove that the actual behaviour of the Compiler-debugger matches the expected behaviour defined by the Interpreter-debugger.

We shall not discuss the problem of debugging optimised code in this thesis any further. However, this is certainly an important problem to be formally addressed, and the framework developed in this thesis can be used in this task.

The Access to Debugging Information

We now discuss a second difference between Interpreter-debuggers and Compiler-debuggers that is essential for debugging purposes: the access to information about the evaluation. An Interpreter-debugger has access to the entire evaluation history of the program; a Compiler-debugger has access to only the contents of a single machine state. Therefore, the information that is necessary for a Compiler-debugger must be encoded in the states of the machine or else must be explicitly built and manipulated by the debugger. The main conceptual difference between the two kinds of information is that evaluation histories are automatically built by the CS based interpreter, while in the Compiler-debugger the information must be constructed, and manipulated explicitly.

To illustrate some problems that may occur when a debugger has to be designed using only the information that is in the machine states, let us compare a CS^{Env} state of the evaluation of the expression **let** $x = 1 + 2$ **in** x with the corresponding DCAM state of the evaluation of the code of this expression. At

step (5), the two corresponding states are as follows, where the state on the left is the $\mathcal{CS}^{Env}$ state, and the one on the right is the DCAM state:

$$[x \mapsto 3 \cdot \varepsilon_E \vdash x, \langle \underline{0}, \underline{3} \rangle \Rightarrow v] \qquad\qquad (((\,),3) \cdot \varepsilon_S, \mathbf{brk} \cdot \mathbf{cdr} \cdot \varepsilon_C)$$

As far as the evaluation is concerned, both states have equivalent information. However, for debugging purposes the $\mathcal{CS}^{Env}$ state has more information than the DCAM state. For example, the identifier x and the path of the current expression are missing in the machine state. Moreover, we cannot reconstruct these data from the machine state as it stands

A possible solution, to this problem is to add to the normal code of the above expression, some *debugging-code* that builds the required information on the machine states. For instance, using **quoteid** instructions we can build an environment with the variables in it, and using the **quotep** instruction we can place the path of the current expression on the top of the stack. These informations can then be used by the debugger to evaluate the debugging commands. With a suitable debugging code we build the machine state $(\langle \underline{0}, \underline{3} \rangle \cdot ((), (x,3)) \cdot \varepsilon_S, \mathbf{brk} \cdot \mathbf{cdr} \cdot \varepsilon_C)$ to replace the state $(((\,),3) \cdot \varepsilon_S, \mathbf{brk} \cdot \mathbf{cdr} \cdot \varepsilon_C)$.

Let us now consider another instance of the same problem. Let the two states below be hypothetical states of the evaluation of some *Fun* expression: the first is a $\mathcal{CS}^{Env}$ state; the second is its corresponding DCAM state:

$$[x \mapsto (\varepsilon_E, y, 1+2, pa) \vdash x, pa' \Rightarrow v]$$

$$(((\,), [(\,), \mathbf{brk} \cdot \mathbf{quote}(1) \cdot \mathbf{swap} \cdot \mathbf{brk} \cdot \mathbf{quote}(2) \cdot \mathbf{add} \cdot \varepsilon_C]) \cdot \varepsilon_S, \mathbf{brk} \cdot \mathbf{cdr} \cdot \varepsilon_C)$$

This situation is rather different from the one we just discussed above: in this case the closures in each environment do not have the same representation. If a debugger is required to output the value of x from the above states, an Interpreter-debugger using the $\mathcal{CS}^{Env}$ state would output $(\varepsilon_E, y, 1+2, pa)$ and a Compiler-debugger using the DCAM state would output

$$[(\,), \mathbf{brk} \cdot \mathbf{quote}(1) \cdot \mathbf{swap} \cdot \mathbf{brk} \cdot \mathbf{quote}(2) \cdot \mathbf{add} \cdot \varepsilon_C]$$

A possible solution to this problem is also to include enough information in the machine state so that we can reconstruct the $\mathcal{CS}^{Env}$ closure from the states. Another solution is to consider closures as being non observable, by choosing an appropriate observation signature for the debugger. This second solution agrees with the view that, in most cases, we are not interested in looking to the internal details of the closures when debugging. The latter is the solution used in *Tiny* since closures are non-observable according to $\Sigma_{\mathrm{OBS}}^{Tiny}$.

Therefore, we can solve the problem of access to information on the Compiler-debugger in two ways. We can add the information needed by the debugger on the machine state by generating debugging code that construct such information. Or else, we can make the information non observable in the Interpreter-debugger, such that we do not need to include debugging code. The choice for which solution to adopt depends on the functionality we require from the debugger. However, both solutions are supported by our framework.

The next example shows another compiler for *Fun* into DCAM code; the code it generates builds and manipulates machine environments with identifier names, and creates a pointer to the current source sub-expression on the top of the stack of some states. The objective is to establish a correspondence between the $\mathcal{CS}^{Env}$ and the DCAM states with respect to the (observable) source level information of the state, as well as with respect to the visible evaluation steps.

Example 3 This example modifies the definition of the relation $comp\_env \vdash exp \xrightarrow{DComp} code$. The code the new relation generates for expressions builds and manipulates environments with identifier names. For this, an environment becomes a pair whose first component is an environment and the second component is a pair of an identifier and its value, e.g., $((), (x, 3))$, where $()$ denotes the empty environment.

There are three parts of the code generator where changes are required to manipulate the new form of machine environment: the look up code must have an extra **cdr** instruction to project the value out from the pair of identifier and

value; the code for **let** expression must add the identifier, as well as the value of the first sub-expression, to the current environment; and the code for closures must update the environment with the identifier of its formal parameter. For the last change, we will need to prepend the following code sequence to the code of each closure:

$$\langle \textbf{push,car,swap,cdr,push,quoteid}(id),$$
$$\textbf{swap,cons,cons,quotep}(pa),\textbf{brk}\rangle$$

where id is the formal parameter of the function, and pa is the path of the function's body. In rule (3) we refer to this code as $adj(id, pa)$, and use it as a parameterised macro.

To include the pointer to the current source expression we use the technique developed in Chapter 6: during the compilation we build the abstract-syntax path pa of each sub-expression that is being compiled; this path is compiled into a **quotep**(pa) instruction whose execution loads the path pa on the stack. The debugger can use this path to identify the source expression that is about to be evaluated by, for instance, looking up which source expression the path refers to on a table built during the compilation.

The signature for the new compiler remains the same as in Example 2. The new rules for the code generator and the compiler are given below.

Debugging-Code Generator $\boxed{comp\_env \vdash exp, path \overset{DComp}{\longrightarrow} code}$

$$\frac{}{CE \vdash n, pa \overset{DComp}{\longrightarrow} \langle \textbf{pop,quote}(n)\rangle} \quad (1)$$

$$\frac{(CE, id) \longrightarrow_{LC} c}{CE \vdash id, pa \overset{DComp}{\longrightarrow} \langle \textbf{pop}\rangle @ c @ \langle \textbf{cdr}\rangle} \quad (2)$$

$$\frac{(CE, id) \vdash e, pa \diamond \underline{2} \overset{DComp}{\longrightarrow} c}{CE \vdash \textbf{fn}\ id\ .\ e, pa \overset{DComp}{\longrightarrow} \langle \textbf{pop,cur}(adj(id, pa \diamond \underline{2}) @ c)\rangle} \quad (3)$$

$$\frac{CE \vdash e_1, pa \diamond \underline{1} \overset{DComp}{\longrightarrow} c_1 \qquad CE \vdash e_2, pa \diamond \underline{2} \overset{DComp}{\longrightarrow} c_2}{CE \vdash e_1 + e_2, pa \overset{DComp}{\longrightarrow} \langle \textbf{pop,push,quotep}(pa \diamond \underline{1}),\textbf{brk}\rangle @ c_1 @ \langle \textbf{swap,quotep}(pa \diamond \underline{2}),\textbf{brk}\rangle @ c_2 @ \langle \textbf{add}\rangle} \tag{4}$$

$$\frac{CE \vdash e_1, pa \diamond \underline{2} \overset{DComp}{\longrightarrow} c_1 \qquad CE \vdash e_2, pa \diamond \underline{3} \overset{DComp}{\longrightarrow} c_2}{CE \vdash \textbf{let } id = e_1 \textbf{ in } e_2, pa \overset{DComp}{\longrightarrow} \langle \textbf{pop,push,quotep}(pa \diamond \underline{2}),\textbf{brk}\rangle @ c_1 @ \langle \textbf{push,quoteid}(id),\textbf{swap,cons,cons,quotep}(pa \diamond \underline{3}),\textbf{brk}\rangle @ c_2} \tag{5}$$

$$\frac{CE \vdash e_1, pa \diamond \underline{1} \overset{DComp}{\longrightarrow} c_1 \qquad CE \vdash e_2, pa \diamond \underline{2} \overset{DComp}{\longrightarrow} c_2}{CE \vdash e_1(e_2), pa \overset{DComp}{\longrightarrow} \langle \textbf{pop,push,quotep}(pa \diamond \underline{1}),\textbf{brk}\rangle @ c_1 @ \langle \textbf{swap,quotep}(pa \diamond \underline{2}),\textbf{brk}\rangle @ c_2 @ \langle \textbf{cons,app}\rangle} \tag{6}$$

Compilation of Environments $\boxed{env \longrightarrow_E (comp\_env, val)}$

$$\frac{}{\varepsilon_E \longrightarrow_E (\varepsilon_{CE}, ())} \tag{7}$$

$$\frac{E \longrightarrow_E (CE, v)}{id \mapsto n \cdot E \longrightarrow_E ((CE, id), (v, (id, n)))} \tag{8}$$

$$\frac{E' \longrightarrow_E (CE', v') \qquad (CE', id') \vdash e', pa \overset{DComp}{\longrightarrow} c \qquad E \longrightarrow_E (CE, v)}{id \mapsto (E', id', e', pa) \cdot E \longrightarrow_E ((CE, id), (v, (id, [v', adj(id', pa)@c])))} \tag{9}$$

Compiler $\boxed{(env, exp) \overset{\text{Comp}}{\longrightarrow} state}$

$$\frac{E \longrightarrow_E CE, v \qquad CE \vdash e, pa \overset{DComp}{\longrightarrow} c}{(E, e, pa) \overset{DComp}{\longrightarrow} (pa \cdot v \cdot \varepsilon_S, \langle \textbf{brk}\rangle @ c)} \tag{10}$$

Comments

(3) - (6) Each rule constructs the abstract-syntax path of the sub-expressions.

(7) The compiler now has a third input component *pa* that is the initial path of the expression *e*.

The above rules finishes the new compiler of *Fun* into DCAM. Informally, we will use *DComp* as a name for a Relational Semantics that has the above rules and the relation $\stackrel{DComp}{\longrightarrow}$ as the initial relation. This finishes Example 3. □

The code generated by the above compiler for the expression **let x** $= 1 + 2$ **in x**, with initial path $\langle \underline{0} \rangle$ and on the empty environment, is given below:

$\langle$**pop,push,quotep(**$\langle \underline{0}, \underline{2} \rangle$**),brk,pop,**
push,quotep($\langle \underline{0}, \underline{2}, \underline{1} \rangle$**),brk,pop,quote(**1**),swap,**
quotep($\langle \underline{0}, \underline{2}, \underline{2} \rangle$**),brk,pop,quote(**2**),plus,push,**
quoteid(x**),swap,cons,cons,quotep(**$\langle \underline{0}, \underline{3} \rangle$**),brk,pop,cdr,cdr**$\rangle$

We refer to this sequence by c''. If we start the DCAM with state

$$(\langle \underline{0} \rangle \cdot (\,) \cdot \varepsilon_S, \langle \mathbf{brk} \rangle @ c'' @ \varepsilon_C)$$

the current state after 5 steps is:

$$(\langle \underline{0}, \underline{3} \rangle \cdot ((\,), (x, 3)) \cdot \varepsilon_S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{cdr} \cdot \mathbf{cdr} \cdot \varepsilon_C)$$

At this state $\langle \underline{0}, \underline{3} \rangle$ indicates that boxed sub-expression of **let x** $= 1 + 2$ **in** $\boxed{\mathbf{x}}$ is the current expression since $\langle \underline{0}, \underline{3} \rangle$ is the abstract syntax path of this expression. The environment $((\,), (x, 3))$ contains the identifier x as well as its value 3. If closures are not observable, then the machine states resulting from the evaluation of the code generated by the above compiler contains enough information for a definition of a Compiler-debugger with the same observable behaviour as *Tiny*. In the next section we show a design of a Compiler-debugger called *CTiny* that uses the above compiler, and in Section 7.4 we prove the equivalence between *Tiny* and *CTiny*.

The inclusion of **brk** and **quotep** instructions in the code of expressions for debugging purposes is inspired by the techniques used in a concrete implementation of a compiler-debugger for the CHILL language presented in [CCP91].

Therefore, our techniques to generate debugging code are realistic. This also suggests that our framework can express practical Compiler-debuggers.

Debugging Code and Performance

Let us now discuss the interaction between the two problems discussed above: the different notions of evaluation step and the different access to information about the evaluation between the Interpreter-debuggers and the Compiler-debuggers. This interaction is studied informally, but we argue that the conclusions drawn from this discussion apply to most debuggers in practice.

In order to be able to design a Compiler-debugger whose behaviour is equivalent to the behaviour of an Interpreter-debugger, we must undertake two tasks. First, to establish a correspondence between the evaluation steps and states in both debuggers. Second, to encode in each machine state the observable source level information about the evaluation that is required by the debugger.

On the one hand, how much information needs to be encoded depends directly on the observation signature of the debuggers: in general, the larger the signature the more information is necessary. On the other hand, the states in which this information must be present depends on the granularity of the evaluation steps: the finer is the granularity the more often the information will be required; thus, it will have to be present in more states.

To encode the source information in the machine states often requires machine code to build and manipulate this information, which we call *debugging-code*. For instance, the *adj* code sequence in the compiler of Example 3 is a sequence of debugging-code. Since this code does not perform any computation related to the actual evaluation of the program, it will cause an overhead on the evaluation of the program. This overhead is directly proportional to how much information is necessary to be encoded and inversely proportional to how often the debugging-code has to be evaluated.

If we consider that a debugger is more expressive the finer is the granularity of its steps and the more information about the evaluation it provides, we have the

following problem: the more expressive the debugger is the larger the overhead on the evaluation of the program. This problem is part of the specification of any Compiler-debugger in practice, although often it is not identified nor addressed explicitly. Moreover, the literature does not present any formal approach to this problem.

The framework we defined in this chapter for the specification of Compiler-debuggers forms the basis for such an approach. Although we do not treat this problem in depth, it is an interesting problem for future work.

Concluding Remarks

In this section we studied various aspects of Compiler-debuggers: the evaluation step, the access to information about the evaluation, and the influence of debugging code on the performance of the programs. Each of these aspects is part of the specification of every Compiler-debugger in practice. However, the approaches to debugger design in the literature in general do not treat these problems explicitly. When those problems are addressed, they are addressed in an ad hoc fashion with solutions that in general cannot be applied to different debuggers.

In our approach, these problems are treated for an arbitrary Compiler-debugger. In the next section, the aspects discussed in this section will be used in the complete specification of a Compiler-debugger.

7.3.2 The Specification of a Compiler-debugger

In this section we design a debugger, called *CTiny*, based on the compiler of *Fun* into DCAM, defined in Example 3.

A Specification Notation for Compiler-debuggers

We will use the notation DSL defined in Section 6.5 in the specification of Compiler-debuggers as well. In the specification of a Compiler-debugger we need the ability to describe single evaluation states; thus the sequence patterns of DSL

are not necessary in such specifications. Therefore, we will only use the subset of DSL that has the constructor patterns. In this sense, the subset of DSL used in this section is just the untyped λ-calculus with pattern matching.

Since DSL is parametric on a Relational Specification, we will use the instantiation of DSLby *DComp*. For this instantiation we define the functions corresponding to `initial` and `next` that were defined in Section 6.5. The function corresponding to `initial` is called `M_initial`, and in its definition recall that $\stackrel{DComp}{\longrightarrow}$ is a partial function.

$$\begin{array}{ll} \texttt{M\_initial}(E \vdash e \Rightarrow v) = st & \text{if there exists } st \in T(\Sigma^{DComp})_{state} \\ & \text{such that } (E, e) \stackrel{DComp}{\longrightarrow} st \\ \texttt{M\_initial}(E \vdash e \Rightarrow v) = (\varepsilon_S, \varepsilon_C) & \text{otherwise} \end{array}$$

The function `M_next` is the corresponding to the function `next`. In the definition of `M_next`, recall that $\stackrel{\text{DCAM}}{\Longrightarrow}$ is a partial function.

$$\begin{array}{ll} \texttt{M\_next}(st) = st' & \text{if } st \stackrel{\text{DCAM}}{\Longrightarrow} st' \\ \texttt{M\_next}(st) = (\varepsilon_S, \varepsilon_C) & \text{if there is no } st' \text{ such that } st \stackrel{\text{DCAM}}{\Longrightarrow} st' \end{array}$$

The function `M_initial` builds an initial DCAM state from an environment and expression, by compiling them using the compiler of *DComp*. The function `M_next` advances the current state by one step of the DCAM machine. These are the only pre-defined functions that produce states as results; in particular `M_next` is the only function that advances the evaluation; thus, only states that are derivatives of an initial DCAM state can be reached using the pre-defined functions. This simplifies the verification of the correctness condition of Definition 1 for a Compiler-debugger using the pre-defined functions.

Some Auxiliary Functions

The next functions are defined to be used in the specification of the debugging commands of *CTiny*. In these definitions, the results of the functions are the terms of sort *result* defined in the specification of *Tiny* in Section 6.6 (page 190).

The following function specifies a stepping command that advances the evaluation by n DCAM steps, where n is an integer number.

```
fun M_step((v·S,ε_C), n) = ((v·S,ε_C),succeed)
  | M_step((ε_S,ε_C), n) = ((ε_S,ε_C),fail)
  | M_step(st, n) = if n>0 then M_step(M_next(st),n-1)
                           else (st,null)
```

The function `lookup` searches for the value of an identifier on a machine environment.

```
fun M_lookup((ME,(x,v)), y) = if x = y then val(v)
                                       else lookup(ME,y)
  | M_lookup(( ), y) = null
```

The function `M_run_until` advances the evaluation until `x` has value `i` in the current machine environment. In its definition, the pattern `(pa · E · _, _)` matches a state that has a path `pa` on the top of the stack, and a machine environment `E` on the second position of the stack.

```
fun M_run_until((v·S,ε_C), x, n) = ((v·S,ε_C),succeed)
  | M_run_until((ε_S,ε_C), x, n) = ((ε_S,ε_C),fail)
  | M_run_until (st as (pa · ME · _,_), x, n) =
      case lookup(ME, x) of val(n1) => if n1 = n then (s, null)
                                       else M_run_until(M_next(st), x, n)
                          | _ => M_run_until(M_next(st), x, n)
```

The following functions will be used in the specification of the *CTiny* commands **show** *id*, **show pos**, and **show res**

```
fun M_showid((_ · ME · _, _), y) = M_lookup(ME, y)
  | M_showid((ε_S, ε_C), y) = fail
  | M_showid(_,_) = null
```

```
fun M_showpos((pa · _, _)) = path(pa)
  | M_showpos((ε_S, ε_C)) = fail
  | M_showpos(_) = null
```

```
fun M_showres((v · _, ε_C)) = val(v)
  | M_showres((ε_S, ε_C)) = fail
  | M_showres(_) = null
```

The Specification of *CTiny*

We define a Compiler-debugger for an arbitrary program formula $p \in P_{X^E}(\Omega^{DComp})$. We start by defining an evaluator for p. Let $EV = (\Gamma_{DCAM}, \stackrel{DCAM}{\Longrightarrow}, I_{EV})$ be the evaluator derived from the debugging machine DCAM of Example 1 where $I_{EV} = \texttt{M\_initial}(p)$.

We now define the Compiler-debugger $CTiny = (\Sigma^{CTiny}, \Delta^{CTiny}, A^{CTiny}, \Sigma^{Tiny}_{OBS})$, for p based on *DComp* and EV. The algebraic signature Σ^{CTiny} is the union of Σ^{DComp} and the signature that defines the debugging language and language of results. This signature is the same given for *Tiny* in Section 6.6 (page 190), except that the function operations of sort *val* are defined in Σ^{DComp} instead of in Σ^{Env}.

The algebra A^{CTiny} agrees with A^{DComp} on the interpretation of the symbols of Σ^{DComp} and give the term algebra interpretation for debugging commands and results. The observation signature Σ^{Tiny}_{OBS} was defined on page 208.

It remains to define the debugging commands. Let Δ^{CTiny} be the following labelled terminal transition system:

$$(\Gamma_{\mathrm{DCAM}}, g^{CTiny}, T(\Sigma^{CTiny})_{command} \times T(\Sigma^{CTiny})_{result}, \longrightarrow_{CTiny}, \mathrm{I}^{CTiny})$$

where the initial state $\mathrm{I}^{CTiny} = \mathrm{I}_{\mathrm{EV}}$, and g^{CTiny} is the identity function on Γ_{DCAM}. It remains to define the relation $\longrightarrow_{CTiny}$. For all $st \in \Gamma_{\mathrm{DCAM}}$ and $d \in T(\Sigma^{CTiny})_{command}$ the transition:

$$st \xrightarrow{d,r}_{CTiny} st'$$

is defined as follows:

$d =$ **reset**	$r =$ **null** and $st' = \mathrm{I}^{CTiny}$
$d =$ **step n**	$(st', r) =$ `M_step`$(st,\ n)$
$d =$ **run until** $id = n$	$(st', r) =$ `M_run_until`$(st,\ id,\ n)$
$d =$ **show** id	$r =$ `M_showid`$(st,\ id)$ and $st' = st$
$d =$ **show pos**	$r =$ `M_showpos`(st) and $st' = st$
$d =$ **show res**	$r =$ `M_showres`(st) and $st' = st$

Checking the Requirements of Definition 1

It is easy to check that *CTiny* is a Debugger-schema for p based on *DComp* and EV. First, notice that functions `M_step`, `M_run_until`, `M_showid`, `M_showpos`, and `M_showres` are total. Then it is easy to check that $\longrightarrow_{CTiny}$ obeys the robustness requirement of Definition 1. The functionality requirement of Definition 2 is fulfilled by the **step** n command.

7.4 A Proof of Debugger Correctness

In this section we prove that the Compiler-debugger *CTiny* is equivalent to the Interpreter-debugger *Tiny* of Section 6.6, which we redefined in Section 7.3 to include an observation signature. The main objectives of this example are to illustrate the proof method of Debugger Bisimulation and demonstrate that proofs

of debugger correctness are feasible. Not all parts of the proof will be given in full detail.

Proposition 2 The debuggers *Tiny* and *CTiny* are observationally equivalent, i.e., $Tiny \approx CTiny$. □

The proof of this proposition is given for an arbitrary $p \in P_{X^E}(\Omega^{Env})$. To carry out this proof we define a relation $\mathcal{B} \subseteq \mathrm{DH} \times \Gamma_{\mathrm{EV}}$ and prove that $\mathcal{B}$ is a Debugger Bisimulation according to Definition 2.

We first prove some lemmas that will simplify the proof of Proposition 2. For these lemmas we define a relation G on the carriers of observable sorts of the algebras A^{Tiny} and A^{CTiny}.

Definition 6 (The Relation G) Let us define a $\mathrm{S}_{\mathrm{OBS}}^{Tiny}$-sorted relation G, such that for all $s \in \mathrm{S}_{\mathrm{OBS}}^{Tiny}$, $\mathrm{G}_s \subseteq \mathrm{A}_s^{Tiny} \times \mathrm{A}_s^{CTiny}$.

For $s \in \{var, nat, path\}$, G_s is the identity relation on $\mathrm{A}_s^{Tiny} \times \mathrm{A}_s^{CTiny}$.

$\mathrm{G}_{funval} =$

$\{([\mathrm{E}, id, e, pa], [S, adj(id, pa)@c]) : \mathrm{E} \longrightarrow_E (\mathrm{CE}, \mathrm{S}) \text{ and } (\mathrm{CE}, id) \vdash e, pa \stackrel{Comp}{\longrightarrow} c\}$

$\mathrm{G}_{val} = \mathrm{G}_{funval} + \mathrm{G}_{nat}$

$\mathrm{G}_{result} = \mathrm{G}_{val} + \mathrm{G}_{exp} + \mathrm{G}_{env} + \mathrm{G}_{var} + \mathrm{G}_{path}$ □

Since the algebras A^{Tiny} and A^{CTiny} give the same interpretations for terms in $T(\Sigma_{\mathrm{OBS}}^{Tiny})$ it is trivial to prove that $\mathrm{G} : \mathrm{A}^{Tiny} \Longrightarrow\!\!\!\!\times_{\Sigma_{\mathrm{OBS}}^{Tiny}} \mathrm{A}^{CTiny}$. We will assume this fact in the following proofs.

In the rest of this section we use the following notational convention: whenever $E \vdash e, pa \Rightarrow v$ is a program formula in $\mathrm{P}_{\mathrm{X}}(\Omega_{Env})$, v_E and c_e denote the DCAM environment and DCAM code such that $E \longrightarrow_E (v_E, CE)$ and $CE \vdash e, pa \Rightarrow c_e$. Similarly, whenever $(E, id) \longrightarrow_L v$ belongs to $\mathrm{P}_{\mathrm{X}}(\Omega_{Env})$, v_E and c_{id} denote the DCAM environment and DCAM code such that $E \longrightarrow_E (v_E, CE)$ and $CE \vdash id, \langle \underline{0} \rangle \Rightarrow c_{id}$.

The following lemma states the equivalence between environment manipulation in *Env* and *DComp*. This lemma will be used in the proof of Lemma 2.

Lemma 1 (Environment Lemma)
For all $E \in T(\Sigma^{Env})_{env}$, $S \in T(\Sigma^{DComp})_{stack}$, $c \in T(\Sigma^{DComp})_{code}$, $(id, id) \in \mathrm{G}_{var}$, $v \in \mathrm{X}_{val}$, substitution θ, and goal stack α:

1. If there is a $\mathcal{CS}$ sequence such that $[\theta : (E, id) \longrightarrow_L v :: \alpha] \overset{\mathrm{CS}^*}{\Rightarrow} [\theta' : \alpha]$ then there is $v' \in \mathrm{A}_{val}^{DComp}$ such that $(v_E \cdot S, c_{id} \cdot c) \overset{\mathrm{CAM}^*}{\longrightarrow} (v' \cdot S, c)$ and $(v\theta', v') \in \mathrm{G}_{val}$.

2. And conversely.

Proof (Sketch) The proof of Lemma 1(1) is by structural induction on E. Since we are assuming that there exists a $\mathcal{CS}$ sequence, then $E \neq \varepsilon_E$.

$\boxed{E = id' \mapsto \overline{v} \cdot \varepsilon_E}$ In this case, since there is a $\mathcal{CS}$ sequence, $id' = id$. Furthermore, $v_E = (v_{E'}, (id, \overline{v'}))$ and $c_{id} = \mathbf{cdr} \cdot \mathbf{cdr}$. Then $v\theta' = \overline{v}$ in the first evaluation, and $v' = \overline{v'}$ in the second evaluation. From the definition of the relation $env \longrightarrow_E (val, comp\_env)$ in Example 3, it is simple to check that $(\overline{v}, \overline{v'}) \in \mathrm{G}_{val}$.

$\boxed{E = id' \mapsto \overline{v} \cdot E'}$ If $id = id'$, this case follows as above. If $id \neq id'$, then $v_E = (v_{E'}, (id', \overline{v'}))$ and $c_{id} = \mathbf{car} \cdot c' \cdot \mathbf{cdr}$. Then the result follows by applying the inductive hypothesis on the following states:

$$[\theta'' : (E', id) \longrightarrow_L v :: \alpha] \qquad (v_{E'} \cdot S, c' \cdot c)$$

The proof of Lemma 1(2) follows similarly by structural induction on v_E. □

The following lemma establishes a correspondence between the steps of the evaluation on the $\mathcal{CS}_{Env}$ system and on the DCAM. Therefore, it is essential in the proof of the equivalence between the debuggers.

Intuitively, the lemma says that if $\mathcal{CS}_{Env}$ takes N visible steps to evaluate a sub-program to its result, then, on the machine, the code of the sub-program evaluates to its results in $\mathrm{N} - 1$ DCAM step followed by $\mathrm{M} \geq 0$ steps of the (underlying) CAM. In Lemma 3, we shall see that the CAM steps that complete the Nth DCAM step of the sub-program also adjust the stack in order to establish a correspondence between the $\mathcal{CS}^{Env}$ state and the machine state with respect to the information about the evaluation.

Recall from section 6.4.1 that, for any debugging history dh and step predicate SP, $\mid dh \mid_{\mathrm{SP}}$ denotes the number of visible steps in dh according to SP. In the next lemma we use $\mid dh \mid_{nolookup}$ denoting the number of visible steps of $dh \in$ DH according to the step predicate *nolookup* defined in section 6.6. Moreover, whenever $st \overset{\mathrm{CS}}{\Rightarrow}{}^{k} st'$ is a debugging history, we abbreviate $\mid st \overset{\mathrm{CS}}{\Rightarrow}{}^{k} st' \mid_{nolookup}$ by $\mid \overset{\mathrm{CS}}{\Rightarrow}{}^{k} \mid_{nolookup}$.

To improve the readability of the following proofs we will write a sequence of machine code separated by "·" even where we should use the concatenation operator "@". The context and the names of the meta-variables will be sufficient to resolve ambiguities.

Lemma 2 For all program formula $E \vdash e, pa \Rightarrow v \in \mathrm{P_X}(\Omega_{Env})$, substitution θ, and goal stack α the following holds:

1. If there is a $\mathcal{CS}$ sequence of length $\mathrm{K} > 0$ of the form

$$[\theta : E \vdash e, pa \Rightarrow v :: \alpha] \overset{\mathrm{CS}}{\Rightarrow}{}^{\mathrm{K}} [\theta' : \alpha]$$

   then there exist $\mathrm{L}, \mathrm{M} \geq 0$ such that

$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot c_e \cdot c) \overset{\mathrm{DCAM}^{\mathrm{L}}}{\Longrightarrow} \overset{\mathrm{CAM}^{\mathrm{M}}}{\longrightarrow} (v' \cdot S, c)$$

   $(v\theta', v') \in \mathrm{G}_{result}$, and $\mid \overset{\mathrm{CS}}{\Rightarrow}{}^{\mathrm{K}} \mid_{nolookup} = \mathrm{L} + 1$

2. And conversely.

Proof The proof of item 1 is by induction on the number N of visible steps of the transition $\stackrel{CS^K}{\Rightarrow}$. The proof is done by case analysis on the expression e.

Base Case For $N = 1$, we have three cases to analyse:

$\boxed{e = n}$ In this case the transition sequences are as follows:

$$[\theta : E \vdash n, pa \Rightarrow v :: \alpha] \stackrel{CS^1}{\Rightarrow} [\theta' : \alpha]$$

$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{quote}(n) \cdot c) \stackrel{CAM^2}{\longrightarrow} (n \cdot S, c)$$

In the above sequences $v\theta = n$ and $L = 0$; thus the result holds.

$\boxed{e = id}$ In this case the CS transition sequence is as follows:

$$[\theta : E \vdash id, pa \Rightarrow v :: \alpha] \stackrel{CS^1}{\Rightarrow} \quad (1)$$

$$[\theta'' : (E, id) \longrightarrow_L v :: \alpha] \stackrel{CS^*}{\Rightarrow} \quad (2)$$

$$[\theta''' : \alpha] \quad (3)$$

Applying Lemma 1 in (2) we obtain the following sequence of machine transitions:

$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot c_{id} \cdot c) \stackrel{CAM^2}{\longrightarrow} \quad (1)$$

$$(v_E \cdot S, c_{id} \cdot c) \stackrel{CAM^*}{\longrightarrow} \quad (2)$$

$$(v' \cdot S, c) \quad (3)$$

and from Lemma 1 $(v\theta''', v') \in G_{val}$. Moreover, $| \stackrel{CS^*}{\Rightarrow} |_{nolookup} = 0$ because the lookup steps are hidden according to *nolookup*. Therefore, $L = 0$ and the result holds.

$\boxed{e = \mathbf{fn}\ id\ .\ e'}$ In this case the transition sequences are as follows:

$$[\theta : E \vdash \mathbf{fn}\ id\ .\ e', pa \Rightarrow v :: \alpha] \stackrel{CS^1}{\Rightarrow} [\theta' : \alpha]$$

$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{cur}(adj \cdot c_{e'}) \cdot c) \stackrel{CAM^2}{\longrightarrow} ([v_E, adj \cdot c_{e'}] \cdot S, c)$$

and $v\theta''' = (E, id, e', pa)$. From the definition of G_{funval}, $(v\theta''', v') \in G_{funval}$. Moreover, $L = 0$ and the result holds.

Inductive Step For $\mathrm{N} > 1$ we have three cases to analyse:

$\boxed{e = e_1 + e_2}$ In this case the transition sequences are as follows:

$$[\theta : E \vdash e_1 + e_2, pa \Rightarrow v :: \alpha]^{\overset{\mathrm{CS}}{\Rightarrow}{}^{1}} \quad (1)$$

$$[\theta' : E \vdash e_1, pa \diamond \underline{1} \Rightarrow v_1 :: E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 :: \ldots]^{\overset{\mathrm{CS}}{\Rightarrow}{}^{k}} \quad (2)$$

$$[\theta'' : E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 :: E \vdash \mathbf{num}(\mathbf{plus}(v_1, v_2)), \langle\,\rangle \Rightarrow v :: \alpha]^{\overset{\mathrm{CS}}{\Rightarrow}{}^{l}} \quad (3)$$

$$[\theta''' : \alpha] \quad (4)$$

Where, $| \overset{\mathrm{CS}}{\Rightarrow}{}^{k} |_{nolookup} = m$, and $| \overset{\mathrm{CS}}{\Rightarrow}{}^{l} |_{nolookup} = n$. The total number of visible steps of the above history is $\mathrm{N} = m + n + 1$, because the step (1) is visible and the step of the evaluation of the formula $E \vdash \mathbf{plus}(v_1, v_2), \langle\,\rangle \Rightarrow v$ is hidden. Now, we must show that the machine evaluation of the code c_e is done in $\mathrm{N} - 1$ steps of the DCAM.

$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{push} \cdot \mathbf{quotep}(pa \diamond \underline{1}) \cdot \mathbf{brk} \cdot c_{e_1} \cdot \mathbf{swap} \cdot \ldots)^{\overset{\mathrm{DCAM}}{\Longrightarrow}{}^{1}} \quad (1)$$

$$(pa \diamond \underline{1} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \mathbf{swap} \cdot \mathbf{quotep}(pa \diamond \underline{2}) \ldots)^{\overset{\mathrm{DCAM}}{\Longrightarrow}{}^{m-1} \overset{\mathrm{CAM}}{\longrightarrow}{}^{+}} \quad (2^{\dagger})$$

$$(v_1' \cdot v_E \cdot S, \mathbf{swap} \cdot \mathbf{quotep}(pa \diamond \underline{2}) \cdot \mathbf{brk} \cdot c_{e_2} \ldots)^{\overset{\mathrm{CAM}}{\longrightarrow}{}^{2}} \quad (3)$$

$$(pa \diamond \underline{2} \cdot v_E \cdot v_1' \cdot S, \mathbf{brk} \cdot c_{e_2} \cdot \mathbf{add} \cdot \ldots)^{\overset{\mathrm{DCAM}}{\Longrightarrow}{}^{n-1} \overset{\mathrm{CAM}}{\longrightarrow}{}^{*}} \quad (4^{\dagger})$$

$$(v_2' \cdot v_1' \cdot S, \mathbf{add} \cdot c)^{\overset{\mathrm{CAM}}{\longrightarrow}{}^{1}} \quad (5)$$

$$(v' \cdot S, c) \quad (6)$$

Where, the transitions marked with † are the points where the inductive hypothesis was applied. Therefore, $(v_1\theta''', v_1') \in \mathrm{G}_{val}$, $(v_2\theta''', v_2') \in \mathrm{G}_{val}$, $v\theta''' = \mathbf{plus}(v_1\theta''', v_2\theta''')$, and $v' = \mathbf{plus}(v_1', v_2')$. Since G is a strong correspondence and **plus** is compatible with G, we have that $(v\theta''', v') \in \mathrm{G}_{val}$. The sub-sequence from the state (2) to the state (4) has $(m-1)+1$ $\overset{\mathrm{DCAM}}{\Longrightarrow}$ steps. Therefore, the total $\overset{\mathrm{DCAM}}{\Longrightarrow}$ steps of the above sequence is $1 + (m-1) + 1 + (n-1) = m + n = \mathrm{N} - 1$. Therefore, the result holds for this case.

$\boxed{e = \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2}$ In this case the transition sequences are as follows:

$[\theta : E \vdash \textbf{let } id = e_1 \textbf{ in } e_2, pa \Rightarrow v :: \alpha] \overset{\text{CS}}{\Rightarrow}^1$ (1)

$[\theta' : E \vdash e_1, pa \diamond \underline{2} \Rightarrow v_1 :: id \mapsto v_1 \cdot E \vdash e_2, pa \diamond \underline{3} \Rightarrow v_2 :: \alpha] \overset{\text{CS}}{\Rightarrow}^k$ (2)

$[\theta'' : id \mapsto v_1 \cdot E \vdash e_2, pa \diamond \underline{3} \Rightarrow v_2 :: \alpha] \overset{\text{CS}}{\Rightarrow}^l$ (3)

$[\theta''' : \alpha]$ (4)

Where, $| \overset{\text{CS}}{\Rightarrow}^k |_{nolookup} = m$, and $| \overset{\text{CS}}{\Rightarrow}^l |_{nolookup} = n$. Therefore, $\text{N} = m + n + 1$. Now, we must show that the machine evaluation of the code for e is done in $\text{N} - 1$ steps of the DCAM machine.

$(pa \cdot v_E \cdot S, \textbf{brk} \cdot \textbf{pop} \cdot \textbf{push} \cdot \textbf{quotep}(pa \diamond \underline{2}) \cdot \textbf{brk} \cdot c_{e_1} \cdot \textbf{push} \cdot \ldots) \overset{\text{DCAM}}{\Longrightarrow}^1$ (1)

$(pa \diamond \underline{2} \cdot v_E \cdot v_E \cdot S, \textbf{brk} \cdot c_{e_1} \cdot \textbf{push} \cdot \textbf{quoteid}(id) \cdot \ldots) \overset{\text{DCAM}}{\Longrightarrow}^{m-1} \overset{\text{CAM}^*}{\longrightarrow}$ ($2^\dagger$)

$(v_1' \cdot v_E \cdot S, \textbf{push} \cdot \textbf{quoteid}(id) \cdot \textbf{swap} \cdot \textbf{cons} \cdot \ldots) \overset{\text{CAM}}{\longrightarrow}^4$ (3)

$((id, v_1') \cdot v_E \cdot S, \textbf{cons} \cdot \textbf{quotep}(pa \diamond \underline{3}) \cdot \textbf{brk} \cdot c_{e_2} \ldots) \overset{\text{CAM}}{\longrightarrow}^2$ (4)

$(pa \diamond \underline{3} \cdot (v_E, (id, v_1')) \cdot S, \textbf{brk} \cdot c_{e_2} \ldots) \overset{\text{DCAM}}{\Longrightarrow}^{n-1} \overset{\text{CAM}^*}{\longrightarrow}$ ($5^\dagger$)

$(v_2 \cdot S, c)$ (6)

where the transitions marked with † are the points where the inductive hypothesis was applied. Therefore, $(v_1\theta''', v_1') \in \text{G}_{val}$ and $(v_2\theta''', v_2') \in \text{G}_{val}$.

The sub-sequence from the state (2) to the state (5) in the above transition sequence, has $(m-1)+1$ $\overset{\text{DCAM}}{\Longrightarrow}$ steps. Therefore, the total $\overset{\text{DCAM}}{\Longrightarrow}$ steps of the above sequence is $1 + (m-1) + 1 + (n-1) = m + n = \text{N} - 1$. Therefore, the result holds for this case.

$\boxed{e = e_1(e_2)}$ In this case the transition sequences are as follows:

$[\theta : E \vdash e_1(e_2), pa \Rightarrow v :: \alpha] \overset{\text{CS}}{\Rightarrow}^1$ (1)

$[\theta' : E \vdash e_1, pa \diamond \underline{1} \Rightarrow (E, id, e', pa') :: E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 :: \ldots] \overset{\text{CS}}{\Rightarrow}^j$ (2)

$[\theta'' : E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 :: id \mapsto v_2 \cdot E \vdash e', pa' \Rightarrow v :: \alpha] \overset{\text{CS}}{\Rightarrow}^k$ (3)

$[\theta''' : id \mapsto v_2 \cdot E \vdash e', pa' \Rightarrow v :: \alpha] \overset{\text{CS}}{\Rightarrow}^l$ (4)

$[\theta'''' : \alpha]$ (5)

where $| \overset{CS^j}{\Rightarrow} |_{nolookup} = m$, $| \overset{CS^k}{\Rightarrow} |_{nolookup} = n$, and $| \overset{CS^l}{\Rightarrow} |_{nolookup} = o$. Therefore, $\mathrm{N} = m + n + o + 1$. Now, we must show that the machine evaluation of the code for e is done in $\mathrm{N} - 1$ steps of the DCAM machine.

$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{push} \cdot \mathbf{quotep}(pa \diamond \underline{1}) \cdot \mathbf{brk} \cdot c_{e_1} \cdot \mathbf{swap} \cdot \ldots) \overset{\mathrm{DCAM}^1}{\Longrightarrow} \quad (1)$$

$$(pa \diamond \underline{1} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \mathbf{swap} \cdot \mathbf{quotep}(pa \diamond \underline{2}) \cdot \ldots) \overset{\mathrm{DCAM}^{m-1}}{\Longrightarrow} \overset{\mathrm{CAM}^*}{\longrightarrow} \quad (2^\dagger)$$

$$([S_{E'}, adj \cdot c_{e'}] \cdot v_E \cdot S, \mathbf{swap} \cdot \mathbf{quotep}(pa \diamond \underline{2}) \cdot \mathbf{brk} \cdot c_{e_2} \cdot \ldots) \overset{\mathrm{CAM}^2}{\longrightarrow} \quad (3)$$

$$(pa \diamond \underline{2} \cdot v_E \cdot [S_{E'}, adj \cdot c_{e'}] \cdot S, \mathbf{brk} \cdot c_{e_2} \cdot \mathbf{cons} \cdot \mathbf{app} \cdot c) \overset{\mathrm{DCAM}^{n-1}}{\Longrightarrow} \overset{\mathrm{CAM}^*}{\longrightarrow} \quad (4^\dagger)$$

$$(v_2 \cdot [S_{E'}, adj \cdot c_{e'}] \cdot S, \mathbf{cons} \cdot \mathbf{app} \cdot c) \overset{\mathrm{CAM}^2}{\longrightarrow} \quad (5)$$

$$((v_{E'}, v_2) \cdot S, adj \cdot c_{e'} \cdot c) \overset{\mathrm{CAM}^{10}}{\longrightarrow} \quad (6)$$

$$(pa' \cdot (v_{E'}, (id, v_2)) \cdot S, \mathbf{brk} \cdot c_{e'} \cdot c) \overset{\mathrm{DCAM}^{o-1}}{\Longrightarrow} \overset{\mathrm{CAM}^*}{\longrightarrow} \quad (7^\dagger)$$

$$(v' \cdot S, c) \quad (8)$$

where the transitions marked with † are the points where the inductive hypothesis was applied. Therefore, $((E', id, e', pa', [v_{E'}, adj \cdot c_{e'}]) \in \mathrm{G}_{funval}$, and $(v_2\theta''', v_2') \in \mathrm{G}_{val}$, where the code adj is as follows:

$$\langle \mathbf{push,car,swap,cdr,push,quoteid}(id),$$
$$\mathbf{swap,cons,cons,quotep}(pa'),\mathbf{brk}\rangle$$

The sub-sequence from the state (2) to the state (4) in the above transition sequence, has $(m-1)+1$ $\overset{\mathrm{DCAM}}{\Longrightarrow}$ steps, the sub-sequence from (4) to (7) has $(n-1)+1$ $\overset{\mathrm{DCAM}}{\Longrightarrow}$ steps. Therefore, the total $\overset{\mathrm{DCAM}}{\Longrightarrow}$ steps of the above sequence is $1+(m-1)+1+(n-1)+1+(o-1) = m+n+o = \mathrm{N}-1$. Therefore, the result holds for this case.

The proof of item 2 of this lemma follows similarly by induction on the length of the DCAM evaluation of the code of the expressions. This finishes the proof of Lemma 1. □

We now define a relation $\mathcal{B}$ between the states of *Tiny* and *CTiny*.

Definition 7 (The Relation $\mathcal{B}$) Let $\mathcal{B} \subseteq \mathrm{DH} \times \Gamma^{\mathrm{EV}}$ be the relation inductively defined as follows:

1. $(\mathrm{I}^{Tiny}, \mathrm{I}^{CTiny}) \in \mathcal{B}$.
2. If $(dh, st) \in \mathcal{B}$ then $(\texttt{step}(dh,\texttt{1}), \texttt{M\_next}(st)) \in \mathcal{B}$. □

In the next proofs we will use the following proposition. This proposition states that for all program formulae in *Env* either its initial debugging state is the failure state in *Tiny* and *CTiny* or the $\mathcal{CS}^{Env}$ evaluation of the formula and the execution of its DCAM code never fail.

Proposition 3 For all program formula $p = E \vdash e, pa \Rightarrow v \in P_{\mathrm{X}E}(\Omega^{Env})$,

$$\mathrm{I}_p^{Tiny} = \langle [\,] \rangle \text{ if and only if } \mathrm{I}_p^{CTiny} = ((\,), \varepsilon_C)$$

and whenever $\texttt{initial}(p) = [\theta_0 : p]$ and $\texttt{M\_initial}(p) = (v_E, c_e)$ then there is no transitions of the form:

$$[\theta_0 : p] \overset{\mathrm{CS}^*}{\Rightarrow} [\,]$$

$$(v_E, c_e) \overset{\mathrm{DCAM}^*}{\Longrightarrow} (\varepsilon_S, \varepsilon_C)$$

Proof (Sketch) For the above equivalence, if $\mathrm{I}_p^{Tiny} = \langle [] \rangle$ then from the definition of I_p^{Tiny} on page 230 we have that $\mathrm{FV}(close(e, E)) \neq \{\,\}$. In this case it is easy to check that the compiler $\overset{DComp}{\longrightarrow}$ fails when generating code for any sub-expression $x \in \mathrm{FV}(close(e, E))$. Therefore, $\mathrm{I}_p^{CTiny} = (\varepsilon_S, \varepsilon_C)$.

Otherwise, if $\mathrm{I}_p^{CTiny} = (\varepsilon_S, \varepsilon_C)$ then the compiler failed. It is easy to check that the compiler only fails in when generating code for some free-variable. Therefore, $\mathrm{FV}(close(e, E)) \neq \{\,\}$ and $\mathrm{I}_p^{Tiny} = \langle [\,] \rangle$.

For the second part of the proposition, it is not difficult to prove that the evaluation of a closed expression and the execution of its code never fail. □

The following lemma states if a pair (dh, st) belongs to $\mathcal{B}$ then dh and st contain the same observable information about the evaluation that is used by the debugging commands. A completely formal proof of this lemma would have to consider the formalisation of the pattern matching defined in Appendix A. That is, to show that a pattern matches a debugging history or a state, we would have to use the rules of the semantic of the pattern matching of section A.3. However, the patterns that are treated below are very simple and we shall informally verify the pattern matching. However, it should be clear that the formalisation of the debugging specification language is essential for a completely formal proof of debugger correctness.

Lemma 3 For all pairs $(dh, st) \in \mathcal{B}$, one of the following statements hold:

1. <h, []> matches dh if and only if $(\varepsilon_S, \varepsilon_C)$ matches st.
2. ι(<>, [_: E ⊢ e, pa ⇒ v], ..., [sub : success], <>) matches dh if and only if (v' · S, ε_C) matches st and if both sides of this equivalence are true then (sub(v), v') $\in G_{val}$.
3. <h, [E ⊢ e, pa ⇒ v :: _]> matches dh if and only if (pa' · SE · _, ce · _) matches st and if both sides of this equivalence are true then ap = pa', E is bound to E, SE is bound to v_E, e is bound to e, and ce is bound to c_e.

Proof

The proof of Lemma 3(1) is an immediate consequence of Proposition 3. The proof of Lemma 3(2) follows immediately from Lemma 2 for the case that $\alpha = \varepsilon$, $S = \varepsilon_S$, and $c = \varepsilon_C$. For the proof of Lemma 3(3), it is easy to check from the definition of $\mathcal{B}$ that for all $(dh, st) \in \mathcal{B}$ there is a natural numbers N such that:

$$dh = \texttt{step}(\mathrm{I}^{Tiny}, \mathrm{N}) \qquad \text{and} \qquad st = \texttt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny})$$

where $\texttt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny})$ denotes the composition of N applications of the function M_next. The right to left implication of Lemma 3(3) is proved by induction on N.

Base Case $\mathrm{N} = 0$.

$$\mathtt{step}(\mathrm{I}^{Tiny}, 0) = \mathrm{I}^{Tiny} = \langle [\theta_0 : E \vdash e, pa \Rightarrow v] \rangle$$
$$\mathtt{M\_next}^0(\mathrm{I}^{CTiny}) = \mathrm{I}^{CTiny} = (pa \cdot v_E \cdot \varepsilon_S, c_e)$$

and the result holds trivially.

Inductive Step $\mathrm{N} > 0$. For an arbitrary debugging history:

$$dh' \diamond [\theta : E \vdash e, pa' \Rightarrow v :: \alpha] = \mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N})$$

we have to analyse the Relational Rule in ϕ^{Env} that was used in the expansion of the current goal of this history. Such a rule exists otherwise N would be 0. For each case we show that the corresponding DCAM state has the required form, i.e., the machine pattern of Lemma 3(3) matches the DCAM state with the required bidings.

Rule 3.2(4) For this rule we have to consider two cases. First, when e is the operand on the left of $e_1 + e_2$, and then when e is the operand on the right of such expression.

$\boxed{e + e_2}$ In this case we have:

$$\mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N} - 1) = dh'' \diamond [\theta : E \vdash e + e_2, pa \Rightarrow v :: \alpha] \quad (1)$$

$$\mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N}) = dh' \diamond [\theta : E \vdash e, pa \diamond \underline{1} \Rightarrow v :: \ldots) \quad (2)$$

Applying the inductive hypothesis on (1) we have the following:

$$\mathtt{M\_next}^{\mathrm{N}-1}(\mathrm{I}^{CTiny}) = (pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{push} \cdot \mathbf{quotep}(pa \diamond \underline{1}) \cdot \mathbf{brk} \cdot \ldots)$$

(1)

$$\mathtt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny}) = (pa \diamond \underline{1} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \ldots) \quad (2)$$

Therefore, the state $\mathtt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny})$ has the required form, and this case holds.

$\boxed{e_1 + e}$ In this case we have:

$$\mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N} - \mathrm{K} - 1) = dh''' \diamond [\theta : E \vdash e_1 + e, pa \Rightarrow v :: \alpha] \quad (1)$$

$$\texttt{step}(\mathrm{I}^{Tiny}, \mathrm{N}-\mathrm{K}) = dh'' \diamond [\theta' : E \vdash e_1, pa \diamond \underline{1} \Rightarrow v :: E \vdash e, pa \diamond \underline{2} \Rightarrow v :: \ldots]$$

(2)

$$\texttt{step}(\mathrm{I}^{Tiny}, \mathrm{N}) = dh' \diamond [\theta'' : E \vdash e, pa \diamond \underline{2} \Rightarrow v :: \ldots] \quad (3)$$

Applying the inductive hypothesis on (1) we have the following:

$$\texttt{M\_next}^{(\mathrm{N}-\mathrm{K}-1)}(\mathrm{I}^{CTiny}) = (pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{push} \cdot \mathbf{quotep}(pa \diamond \underline{1}) \cdot \ldots)(1)$$

$$\texttt{M\_next}^{\mathrm{N}-\mathrm{K}}(\mathrm{I}^{CTiny}) = (pa \diamond \underline{1} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \ldots) \quad (2)$$

Applying Lemma 2 on (2) we have:

$$(pa \diamond \underline{1} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \ldots) \overset{\mathrm{DCAM}^{\mathrm{K}-1}}{\Longrightarrow} \overset{\mathrm{CAM}^*}{\longrightarrow}$$

$$(v_1' \cdot v_E \cdot S, \mathbf{swap} \cdot \mathbf{quotep}(pa \diamond \underline{2}) \cdot \mathbf{brk} \cdot c_{e_2} \cdot \ldots) \overset{\mathrm{CAM}^2}{\longrightarrow} \quad (3)$$

$$(pa \diamond \underline{2} \cdot v_E \cdot v_1' \cdot S, \mathbf{brk} \cdot c_{e_2} \cdot \mathbf{add} \cdot \ldots) = \texttt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny}) \quad (4)$$

Therefore, $\texttt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny})$ has the required form, and this case holds.

Rule 3.2(5) For this rule we also have to analyse two cases:

`let id = e in e2` This case is similar to the first case for rule 3.2(4).

`let id = e1 in e` In this case we have:

$$\texttt{step}(\mathrm{I}^{Tiny}, \mathrm{N}-\mathrm{K}-1) = dh'' \diamond [\theta : E \vdash \texttt{let } id \texttt{ = } e_1 \texttt{ in } e, pa \Rightarrow v :: \alpha] \quad (1)$$

$$\texttt{step}(\mathrm{I}^{Tiny}, \mathrm{N}-\mathrm{K}) =$$

$$dh'' \diamond [\theta' : E \vdash e_1, pa \diamond \underline{2} \Rightarrow v_1 :: id \mapsto v_1 \cdot E \vdash e_2, pa \diamond \underline{3} \Rightarrow v_2 :: \alpha] \quad (2)$$

$$\texttt{step}(\mathrm{I}^{Tiny}, \mathrm{N}) = dh''' \diamond [\theta'' : id \mapsto v_1 \cdot E \vdash e_2, pa \diamond \underline{3} \Rightarrow v_2 :: \alpha] \quad (3)$$

Applying the inductive hypothesis on (1) we have:

$$\texttt{M\_next}^{(\mathrm{N}-\mathrm{K}-1)}(\mathrm{I}^{CTiny}) =$$

$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{push} \cdot \mathbf{quotep}(pa \diamond \underline{2}) \cdot \mathbf{brk} \cdot c_{e_1} \cdot \ldots) \quad (1)$$

$$\texttt{M\_next}^{(\mathrm{N}-\mathrm{K})}(\mathrm{I}^{CTiny}) = (pa \diamond \underline{2} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \mathbf{push} \cdot \ldots) \quad (2)$$

Applying Lemma 2 on (2) we obtain:

$$(pa \diamond \underline{2} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \mathbf{push} \cdot \ldots) \overset{\text{DCAM}^{M-1}}{\Longrightarrow} \overset{\text{CAM}^*}{\longrightarrow} \quad (3)$$

$$(v_1' \cdot v_E \cdot S, \mathbf{push} \cdot \mathbf{quoteid}(id) \cdot \mathbf{swap} \cdot \mathbf{cons} \cdot \ldots) \overset{\text{CAM}^4}{\longrightarrow} \quad (4)$$

$$((id, v_1')v_E \cdot S, \mathbf{cons} \cdot \mathbf{quotep}(pa \diamond \underline{3}) \cdot \mathbf{brk} \cdot c_{e_2} \cdot \ldots) \overset{\text{CAM}^2}{\longrightarrow} \quad (5)$$

$$(pa \diamond \underline{2} \cdot (v_E, (id, v_1')) \cdot S, \mathbf{brk} \cdot c_{e_2} \cdot \ldots) = \mathtt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny}) \quad (6)$$

Therefore, $\mathtt{M\_next}^{\mathrm{N}}(\mathrm{I}^{CTiny})$ has the required form, and this case holds.

Rule 3.2(7) For this rule we have to analyse three cases:

$\boxed{e(e_2)}$ This case is similar to the first case for rule 3.2(4).

$\boxed{e_1(e)}$ This case is similar to the second case for rule 3.2(4).

The third case is when the current goal comes from the third premiss of rule 3.2(7). For this case we have:

$$\mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N} - \mathrm{K} - \mathrm{L} - 1) = dh'''' \diamond [\theta : E \vdash e_1(e_2), pa \Rightarrow v :: \alpha] \quad (1)$$

$$\mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N} - \mathrm{K} - \mathrm{L}) =$$
$$dh''' \diamond [\theta' : E \vdash e_1, pa \diamond \underline{1} \Rightarrow (E', id, e, pa'') :: E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 :: \ldots] \quad (2)$$

$$\mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N} - \mathrm{K}) =$$
$$dh'' \diamond [\theta'' : E \vdash e_2, pa \diamond \underline{2} \Rightarrow v_2 :: id \mapsto v_2 \cdot E \vdash e', pa'' \Rightarrow v :: \alpha] \quad (3)$$

$$\mathtt{step}(\mathrm{I}^{Tiny}, \mathrm{N}) = dh' \diamond (id \mapsto v_2 \cdot E \vdash e, pa'' \Rightarrow v :: \alpha] \quad (3)$$

Applying the inductive hypothesis on (2), we obtain:

$$\mathtt{M\_next}^{(\mathrm{N}-\mathrm{K}-\mathrm{L}-1)}(\mathrm{I}^{CTiny}) =$$
$$(pa \cdot v_E \cdot S, \mathbf{brk} \cdot \mathbf{pop} \cdot \mathbf{push} \cdot \mathbf{quotep}(pa \diamond \underline{1}) \cdot \mathbf{brk} \cdot c_{e_1} \cdot \ldots) \quad (1)$$

$$\mathtt{M\_next}^{(\mathrm{N}-\mathrm{K}-\mathrm{L})}(\mathrm{I}^{CTiny}) = (pa \diamond \underline{1} \cdot v_E \cdot v_E \cdot S, \mathbf{brk} \cdot c_{e_1} \cdot \mathbf{swap} \cdot \ldots) \quad (2)$$

Apllying Lemma 2 on (2) and (4) below, we obtain:

$$([S_{\mathrm{E}'}, adj \cdot c_{e'}] \cdot v_E \cdot S, \mathbf{swap} \cdot \mathbf{quotep}(pa \diamond \underline{2}) \cdot \mathbf{brk} \cdot c_{e_2} \cdot \ldots) \overset{\text{CAM}^2}{\longrightarrow} \quad (3)$$

$$(pa \diamond \underline{2} \cdot v_E \cdot [S_{\mathrm{E}'}, adj \cdot c_{e'}] \cdot S, \mathbf{brk} \cdot c_{e_2} \cdot \mathbf{cons} \cdot \mathbf{app} \cdot c) \overset{\text{DCAM}^{n-1}}{\Longrightarrow} \overset{\text{CAM}^+}{\longrightarrow} \quad (4)$$

$$(v_2 \cdot [S_{\mathrm{E}'}, adj \cdot c_{e'}] \cdot S, \mathbf{cons} \cdot \mathbf{app} \cdot c) \overset{\text{CAM}^2}{\longrightarrow} \quad (5)$$

$$((v_{E'}, v_2) \cdot S, adj \cdot c_{e'} \cdot c) \xrightarrow{\text{CAM}}{}^{10} \tag{6}$$

$$(pa'' \cdot (v_{E'}, (id, v_2)) \cdot S, \mathbf{brk} \cdot c_{e'} \cdot c) = \texttt{M\_next}^{\text{N}}(\text{I}^{CTiny}) \tag{7}$$

The state $\texttt{M\_next}^{\text{N}}(\text{I}^{CTiny})$ is in the required form, and this case holds. This finishes the proof of the left to right implication of Lemma 3(3). The proof of the right to left implication follows similarly by analysing the compiler rule that generated the sequence of code on the DCAM state. This finishes the proof of Lemma 3. □

The following lemma states that the environment lookup functions `lookup` of *Tiny* and `M_lookup` of *CTiny* are equivalent. Its proof is an immediate consequence of Lemma 1 and is omitted here.

Lemma 4 For all pairs E and v_E, and for all $id \in \text{T}(\Sigma_{\text{OBS}})_{var}$:

$$(\texttt{lookup}(E, id), \texttt{M\_lookup}(v_E, id)) \in \text{G}_{val}$$ □

We are now ready to prove Proposition 2. This proof uses Lemmas 2 and 3.

Proof (of Proposition 2)

For this proof we must show that $\mathcal{B}$ is a Debugger Bisimulation according to Definition 2. First, $(\text{I}^{Tiny}, \text{I}^{CTiny}) \in \mathcal{B}$ follows immediately from the definition of $\mathcal{B}$. Let us prove first that $\mathcal{B}$ obeys Definition 2(1). We must show that for all $(dh, st) \in \mathcal{B}$ and $d \in T(\Sigma^{Tiny})_{command}$:

for all $r \in T(\Sigma^{Tiny})_{result}$ and $dh' \in \text{DH}$, $dh \xrightarrow{d,r}_{Tiny} dh'$ implies that there exist $r' \in T(\Sigma^{CTiny})_{result}$ and $st \in \Gamma_{\text{EV}}$ such that $st \xrightarrow{d,r'}_{CTiny} st'$ and the two requirements hold:

1. $(dh', st') \in \mathcal{B}$.

2. If $r \in T(\Sigma^{Tiny}_{\text{OBS}})_{result}$ then $r = r'$

The proof follows by case analysis on d. In this proof we will not consider the cases where $dh = \langle[\,]\rangle$ and $st = (\varepsilon_S, \varepsilon_C)$ since these cases follows trivially from Lemma 3(1) and Proposition 3.

$\boxed{d = \texttt{reset}}$ In this case $dh' = \mathrm{I}^{Tiny}$ and $r = \textbf{null}$. Then $st' = \mathrm{I}^{CTiny}$, $r' = \textbf{null}$, and the proposition holds.

$\boxed{d = \texttt{step}\ n}$ In this case:

$$(dh', r) = \texttt{step}(dh, n)$$
$$(st', r') = \texttt{M\_step}(st, n)$$

This proof is by induction on n.

Base Case $n = 0$. If $\iota(\texttt{<>}, \texttt{[\_:\ \ E} \vdash \texttt{e, pa} \Rightarrow \texttt{v]}, \ldots, \texttt{[sub :\ \ success]}, \texttt{<>})$ matches dh, then $dh' = dh$ and $r = \texttt{succeed}$. From Lemma 3(2) $(\texttt{v' } \cdot \texttt{ S}, \varepsilon_C)$ matches st. Therefore, $st' = st$ and $r' = \texttt{succeed}$ and $r = r'$. When the pattern

$$\iota(\texttt{<>}, \texttt{[\_:\ \ E} \vdash \texttt{e, ap} \Rightarrow \texttt{v]}, \ldots, \texttt{[sub :\ \ success]}, \texttt{<>})$$

does not match dh we have, from Lemma 3(2), that $(\texttt{v' } \cdot \texttt{ S}, \varepsilon)$ does not match st. In this case, $dh' = dh$ and $r = \texttt{null}$ and $st' = st$ and $r' = \texttt{null}$. Therefore, $r = r'$ and the proposition holds.

Inductive Step $n > 0$. If the pattern

$$\iota(\texttt{<>}, \texttt{[\_:\ \ E} \vdash \texttt{e, pa} \Rightarrow \texttt{v]}, \ldots, \texttt{[sub :\ \ success]}, \texttt{<>})$$

matches dh, this case follows as in the base case. Otherwise, since we are assuming that $\texttt{step}(dh, n)$ terminates, then $\texttt{step}(dh, 1)$ must also terminate. In this case, it is easy to check from the definition of step (page 123) that $\texttt{step}(dh, 1) = \texttt{next}(dh)$ and:

$$\texttt{step}(dh, n) = \texttt{step}(\texttt{step}(dh, 1), n - 1)$$
$$\texttt{M\_step}(st, n) = \texttt{M\_step}(\texttt{M\_next}(st), n - 1)$$

Therefore, we can write:

$$(dh', r) = \texttt{step}(\texttt{step}(dh, 1), n-1)$$
$$(st', r') = \texttt{M\_step}(\texttt{M\_next}(st), n-1)$$

From the definition of $\mathcal{B}$ we have that $(\texttt{step}(dh,1), \texttt{M\_next}(st)) \in \mathcal{B}$. Therefore, we can apply the inductive hypothesis to obtain requirements 1 and 2. This finishes the proof of this case.

$\boxed{d = \texttt{run until } id = n}$ In this case:

$$(dh', r) = \texttt{run\_until}(dh, id, n)$$
$$(st', r') = \texttt{M\_run\_until}(st, id, n)$$

Since we are assuming that the call $\texttt{run\_until}(dh, id, n)$ terminates, there exists a finite number N of recursive calls to the function `run_until` in the call $\texttt{run\_until}(dh, id, n)$. The proof of this case is by induction on N.

Base Case N = 0. We have two cases to analyse:

1. ι(<>, [_: E ⊢ e, pa ⇒ v], ..., [_: success], <>) matches *dh*. This case follows as in the `step` *n* case.

2. ι(<>, [_: E ⊢ e, pa ⇒ v], ..., [_: success], <>) does not match *dh*. In this case, <h, [E ⊢ e, pa ⇒ v :: _]> matches *dh* with E bound to E and e bound to e. From Lemma 3(3), (pa' · SE · ..., ce · ...) matches *st* with SE bound to v_E and ce bound to c_e. Since this call terminates with no further recursive call to `run_until` then $\texttt{lookup}(\texttt{E}, id) = n$. From Lemma 4 $\texttt{M\_lookup}(\texttt{SE}, id) = n$. Therefore, $dh' = dh$, $st' = st$, $r = r' = \texttt{null}$, and the result holds

Inductive Step N > 0. We have two cases to analyse:

1. <h, [E ⊢ e, pa ⇒ v :: _]> matches *dh* with E bound to E and e bound to e. From Lemma 3(3), (pa' · SE · ..., ce · ...) matches *st* with SE

bound to v_E and ce bound to c_e. Since this call there is at least one further recursive call to `run_until` then `lookup(E,`id`)` $\neq n$. From Lemma 4 `M_lookup(SE,`id`)` $\neq n$. Therefore,

$$(dh', r) = \texttt{run\_until(step(}dh\texttt{,1)},id,n\texttt{)}$$
$$(st', r') = \texttt{M\_run\_until(M\_next(}st\texttt{)},id,n\texttt{)}$$

From the definition of $\mathcal{B}$ $(\texttt{step(}dh\texttt{,1)},\texttt{M\_next(}st\texttt{))} \in \mathcal{B}$. Therefore, we can apply the inductive hypothesis to obtain requirements 1 and 2.

2. In this case, the pattern `<h, [E ⊢ e, pa ⇒ v :: _]>` does not match *dh*. This result follows trivially by applying the inductive hypothesis.

This finishes the proof of this case.

$\boxed{d = \texttt{show } id}$ The proof of this case is an immediate consequence of Lemmas 3 and 4.

$\boxed{d = \texttt{show pos}}$ The proof of this case is an immediate consequence of Lemma 3.

$\boxed{d = \texttt{show res}}$ For the proof of this case notice that, since G_{val} is the identity on the observable terms then the Observational Equivalence requirement in Definition 2(1) can be replaced by $(r, r') \in G_val$. We have to analyse two cases. First, if the pattern

$$\iota(\texttt{<>, [\_: E ⊢ e, pa ⇒ v], ..., [\_: success], <>})$$

matches *dh*, from Lemma 3(2) `(v' · S,` ε_C`)` matches *st* and $(r, r') \in G_{val}$. Second, both patterns do not match *dh* and *st* respectively, and $r = r' =$ **null**.

The proof of the requirements of Definition 2(2) follows similarly. This finishes the proof of Proposition 2 and we conclude that *Tiny* $\approx$ *CTiny* □

7.5 Summary and Conclusions

In this chapter we developed a theory of debugger correctness and studied some problems related to the design of Compiler-debuggers. We started by generalising the notion of Interpreter-debugger to the notion of Debugger-schema, in which the evaluation of programs does not have to be performed by interpretation using the CS system. We then defined a relation of equivalence of Debugger-schemas that expressed the conditions for debugger correctness, and proved a result which relates the equivalence of Interpreter-debuggers and the Observational Equivalence of their underlying Relational Specifications.

We also characterise the concept of Compiler-debuggers and studied various aspects of the specification of such debuggers: the granularity of debugging steps, the access to information about the evaluation, the influences of optimised code in debugging, and the overhead caused by debugging code. This chapter finished with a complete example of the specification of a Compiler-debugger called *CTiny*, and a proof of equivalence between *Tiny* and *CTiny*.

As far as we are aware of, debugger correctness has not yet been addressed in the literature. The results of this chapter provide a general account of this problem that addresses the correctness of a wide class of debuggers, not only those under Definition 6.2. These are the main theoretical contributions of this chapter.

We demonstrated that proofs of correctness of Compiler-debuggers can be done. It remains to show practical applications of the theory. In order to make the proofs scale up to real examples it will probably be necessary to define methodologies and tools to assist in these proofs. Since the theory of debugger correctness is closely related to the theory of compiler correctness, tools developed to assist in compiler correctness proofs can also be used in debugger correctness proofs. We believe that the results developed in this chapter form the basis for practical applications of debugger correctness.

Chapter 8

Concluding Remarks

This thesis has proposed a theory for the specification and correctness proofs of compilers and a related theory for the specification and correctness proofs of debuggers. In both theories, a Relational Semantics of the programming language was used as the reference point for specification and correctness. It has also demonstrated how to apply these theories to the specification and correctness of concrete compilers and debuggers. This chapter summarises the key issues of this thesis and proposes areas for further research.

8.1 Summary

In this section we summarise the main results of this thesis and discuss how they extend and improve related approaches.

In Chapter 2 we defined Relational Semantics as the underlying semantic formalism used throughout this thesis. This definition was carried out in two stages. Section 2.3 defined the syntactic aspects of the formalism by defining the concept of a Relational Specification Syntax. Section 2.4 defined the semantic aspects of the formalism by defining the concept of a Relational Specification and its Declarative Semantics.

In Chapter 3 we examined the problem of equivalence between Relational

Specifications. We applied the definition of observational equivalence presented in [ST87] to first order models and from this we derived an Observational Equivalence relation betwen Relational Specifications. We then extended the notion of strong correspondence in [Sch87,Sch90] to first order models by defining a relation we called Model Correspondence. Model Correspondence yields a practical proof method for Observational Equivalence that can be used in proofs of equivalence of Relational Specifications. The main result of Chapter 3, expressed in Theorem 3.1, is that Model Correspondence is consistent with respect to Observational Equivalence; thus, justifying the use of Model Correspondence in proofs of equivalence.

In Chapter 4 we treated the problem of compiler correctness in the framework of Relational Semantics. We addressed this problem in two stages. First, we gave a characterisation of the process of Evaluation by Compilation. Second, we used the theory of equivalence between Relational Specifications defined in Chapter 3 to provide a criterion for compiler correctness. This criterion is more general than previous proposals in the literature, as discussed in Section 4.1. It provides a natural generalisation of those proposals by using Observational Equivalence. Furthermore, Model Correspondence can be used as a proof method for compiler correctness proofs.

In Chapter 5 we defined the notions of program evaluation and evaluation step that were used in the theory of debugger design of Chapters 6 and 7. These notions were derived from the definition of the Computational Semantics of a Relational Specification. Theorem 5.1 stated that the Computational Semantics of a Relational Specification is sound with respect to the Declarative Semantics of this specification.

We then discussed a notion of evaluation step that was argued to be agree with our intuition. The Computational Semantics of a particular class of Relational Specifications, called Data-driven Specifications, yields an operational interpretation that agrees with that intuitive notion of evaluation step. We formalised this

operational interpretation in Theorem 5.2. The Computational Semantics of any Data-driven Specification is complete with respect to the Declarative Semantics. This completeness result was proved in Theorem 5.3.

In particular, if the Data-driven Specification is Deterministic, in the sense of Definition 5.4, its Computational Semantics is monogenic, as proved in Theorem 5.4. This makes the Computational Semantics of Deterministic Data-driven Specification suitable for use in a system for the prototyping of programming language implementations like the Centaur system [CIK89] and Berry's Animator Generator [Ber91a]. The advantages of the Computational Semantics over those systems is its clear, formal basis, and its correctness with respect to the formalism of Relational Semantics.

In Chapter 6 we studied the problem of how to give formal specifications of debuggers. This problem was addressed in three stages. We first characterised the class of Interpreter-debuggers. Then, we studied two problems that are common to the design of most debuggers: the granularity of debugging steps and the reference to sub-programs. Finally, we defined a notation, called DSL to assist in the specification of debuggers. The definitions of Chapter 6, and the specification notation DSL, were illustrated in Section 6.6 by a full specification of an Interpreter-debugger called *Tiny*.

In Chapter 7 we treated the problem of debugger correctness. We started by generalising the notion of Interpreter-debugger to the notion of Debugger-schema. We then defined a Debugger Bisimulation relation between Debugger-schemas in Definition 7.2 that expresses the conditions for debugger correctness.

As part of the theory of debugger correctness, we studied the relationship between equivalence of debuggers and equivalence of the underlying program evaluation. An important result of this study is that the equivalence of two Interpreter-debuggers implies the equivalence of their underlying Relational Specifications. This result was stated in Theorem 7.1.

Another problem addressed in Chapter 7 was the study of Compiler-debuggers.

We first gave a general characterisation of Compiler-debuggers, and then studied various aspects of the specification of such debuggers: the granularity of debugging steps, the access to information about the evaluation, the influences of optimised code in debugging, and the overhead caused by debugging code. Chapter 7 finished with a complete example of the specification of a Compiler-debugger called *CTiny*, and a proof of equivalence between *Tiny* and *CTiny*.

8.2 Improvements and Extensions to the Theory

In this section we discuss some improvements that can be made to the theoretical parts of this thesis. Some of these aspects are further discussed in Section 8.3 from an implementation point of view.

Order-sorted Signatures and Algebras

The Relational Semantics formalism may be extended by using order-sorted algebras and order-sorted first order models. An order-sorted algebra extends the expressiveness of many-sorted algebras by adding a partial ordering relation on the sorts of the algebra. Based on the definitions of [Wir89, page 35], an order-sorted algebraic signature is a triple $(\mathrm{S}, \Sigma, \mathrm{O})$ where S is a set of sorts, Σ is a $\mathrm{S}^* \times \mathrm{S}$-sorted family of sets of function names, and O is a set of *subset declarations*.

Then an order-sorted partial Σ-algebra A consists of an S-sorted family of sets $\{\mathrm{A}_s\}$ such that A_s is the *carrier* of sort s, and a partial function $\sigma^{\mathrm{A}} \in [\mathrm{A}_w \rightarrow \mathrm{A}_s]$ for each symbol $\sigma \in \Sigma_{w,s}$, such that for any $s, s' \in \mathrm{S}$, if $s \leq s' \in \mathrm{O}$ then $\mathrm{A}_s \subseteq \mathrm{A}_{s'}$.

The above definitions generalise trivially to first order signatures and first order models. Using order-sorted signatures the definition of the syntax of the language *Exp* of Example 2.1 becomes:

$$
\begin{array}{lll}
exp & ::= & exp + exp \mid \textbf{let } var = exp \textbf{ in } exp \\
var & ::= & \mathbf{x} \mid \mathbf{y} \mid \cdots \\
nat & ::= & \textbf{plus}(nat, nat) \mid 0 \mid 1 \mid \cdots
\end{array}
$$

$var \leq exp$

$nat \leq exp$

This definition is simpler than that of Example 2.1 because the coercion constructors **num** and **id** are no longer necessary. It would not be difficult to redefine the concepts of Relational Specification and Declarative Semantics of Chapter 2 to use order-sorted algebras. Once this is done it is necessary to check its influence on the results of the other chapters.

We have not studied how order-sorted algebras influence the theory developed in Chapters 3 and 4. Nevertheless we believe that the definition of strong correspondence generalises to the case of order-sorted algebras. Furthermore, such a generalisation would make our definition of strong correspondence closer to the strong correspondence defined in [Sch87]. The reason is that we would be able to define observational equivalence with respect to a set of *visible sorts*. The results that could be derived from this fact remain to be studied.

The results of Chapter 5 are not affected by changing the algebraic theory to order-sorted algebras, nor are the definitions of Chapter 6. Theorem 7.3, depend on the algebraic theory in use and it still remains to study how this result is affected by changing to an order-sorted theory.

Conservative Extension and Semantic Transformations

Let $\mathcal{S}$ be a Relational Specification with signature $(\mathrm{S}, \mathrm{F}, \Pi, \pi)$. We define an *extension* of $\mathcal{S}$ to be a Relational Specification $\mathcal{S}'$ whose signature $(\mathrm{S}', \Sigma', \Pi', \pi')$ is *larger* than $(\mathrm{S}, \mathrm{F}, \Pi, \pi)$ in the following sense: $\mathrm{S} \subseteq \mathrm{S}'$, whenever $\sigma \in \Sigma_{w,s}$, then $\sigma \in \Sigma'_{w',s}$, and whenever $\pi \in \Pi_w$ then $\pi \in \Pi'_{w'}$, where $w' \in \mathrm{S}'^*$ and $w \subseteq w'$. In this sense $\mathcal{S}'$ can define more relations, and the operations and relations in $\mathcal{S}'$ can have more components than the corresponding ones in $\mathcal{S}$.

The problem now is to characterise a notion of conservative extension. Intuitively, a *conservative extension* of a Relational Specification $\mathcal{S} = (\Omega, \phi, \mathrm{A}, \Omega_{\mathrm{OBS}})$, is an extension $\mathcal{S}' = \langle \Omega', \phi', \mathrm{A}', \Omega_{\mathrm{OBS}} \rangle$ of $\mathcal{S}$ that preserves satisfaction of the formulae in $F(\Omega)$. Using the theory of equivalence of Relational Specification we can define this notion precisely by saying that a conservative extension of $\mathcal{S}$ is an extension $\mathcal{S}'$ such that $\mathcal{S} \cong \mathcal{S}'$.

However, to prove that an extension is conservative is as difficult as to prove the equivalence of two arbitrary Relational Specifications. An interesting problem for future research is to define transformations on Relational Specification that produce conservative extensions. Once we prove that a transformation only produces conservative extensions we can use the transformed specifications whenever necessary without a further proof of Observational Equivalence.

Transformations can extend the scope of theories and definitions that are only applicable to a particular class of specifications. This is achieved by conservatively transforming a specification that does not belong to a particular class to another specification that does. This problem was discussed in Section 5.4 where the transformation defined by the FAIR algorithm of [dS90] was used to extend the notion of evaluation step to the class of Dynamically-deterministic Specifications.

Another important application of transformations, that is particular to this thesis, is to extend Relational Specifications with debugging informations such as paths, function call stacks, and so forth. We believe that the inclusion of paths to a Relational Specification, as discussed in Section 6.4.2, can be automated. Therefore, another interesting problem would be the definition of a *calculus of transformations* for debugging purposes. This problem could be tackled by first isolating a basic set of transformations and then defining how to compose, or otherwise manipulate, those basic transformations to obtain more sophisticated ones.

Debugging Concurrent and Non-deterministic Languages

The design of debuggers for concurrent and non-deterministic languages is more complex than the design of debuggers for sequential and deterministic languages. In the case of concurrent languages, this complexity is due to problems such as the "probe effect", non-repeatability of the evaluation, and lack of a synchronised global clock.

In a survey paper [MH89], McDowell and Helmbold listed four kinds of techniques that have been used to address the problem of designing debuggers for concurrent languages:

1. The application of *conventional debugging methods* and tools to concurrent languages.

2. *Event-based debugging*, where the execution of a concurrent system is viewed as a sequence (or several parallel sequences) of events. The debugging task is then to collect and analyse such sequences.

3. Techniques for displaying the control flow and distributed data of concurrent systems.

4. Static analysis techniques based on data flow analysis, in which some errors are detected without evaluating the programs.

A conclusion of that survey is that none of the above techniques give entirely satisfactory accounts of concurrent debugging. Therefore, work still remains to be done in integrating these techniques, or finding alternative ones, or both. We believe that an important problem for further research is to provide a theory for the specification and correctness proofs of debuggers for concurrent languages. The framework of this thesis is a step towards such a theory.

In the following paragraphs we shall sketch an extension to the framework defined in this thesis to address debuggers for non-deterministic languages. We also discuss which of the above mentioned techniques would be addressed in such an extension.

Part of our framework already addresses non-deterministic programming languages, namely the Relational Semantics formalism, the notion of equivalence of Relational Specifications and Compiler Correctness, and the Computational Semantics of Relational Specifications. On the other hand, the notion of evaluation step is only defined for Deterministic Relational Specifications; thus only deterministic languages are addressed by these definitions. The definition of Interpreter-debuggers (Definition 6.2) uses this deterministic notion of evaluation step and therefore only expresses debuggers for sequential languages.

From a theoretical point of view, it would not be difficult to change Definition 6.2 such that it considers the evaluation relation $\stackrel{CS}{\Rightarrow}$ (and consequently the relation $\stackrel{EH}{\Longrightarrow}$) as a relation instead of a function. The theory of debugger correctness developed in Chapter 7 does not preclude debuggers for non-deterministic programming languages. Therefore, it should not be difficult to extend the theoretical aspects of this thesis to address debuggers for such languages. The quite general definitions of Chapters 6 and 7 were presented with this extension in mind.

Problems arise from the pragmatic point of view: prototyping and specification of concrete debuggers will become non-trivial. The prototyping of debuggers requires a machine implementation of the CS system. If non-deterministic Data-driven Specifications are allowed we shall need to solve the problem of *selection of Relational Rules* discussed in Section 5.4.

However, the most important problem that arises is the specification of debuggers for non-deterministic languages. A theory for design of such debuggers must address the problem of non-repeatability of evaluation. Clearly, this problem is not addressed by the framework of this thesis even when extended to deal with non-deterministic programming languages.

The general approach to this problem in the literature is to record the entire evaluation history of a program, and to use this record to *replay* the evaluation when necessary. The basic drawback of this solution is that the evaluation history of real systems can grow unmanageably large. Various proposals have been made

trying to make this method applicable in practice.

We propose to extend the $\mathcal{CS}$ system to support replay of evaluations. The basic solution we propose is to record the choices of Relational Rules at each point when more than one rule could be used in the evaluation. We then use this record to guide the replay of the evaluation, making the replayed evaluation deterministic. We believe that a record of the choices of rules in an evaluation is much smaller than the entire evaluation history and possibly of a manageable size.

The changes in the theory to deal with non-deterministic evaluation step and the extension of $\mathcal{CS}$ to support replay of evaluation are just the initial stages towards a theory of debuggers design for non-deterministic languages. It is necessary to investigate other problems, for instance whether it is necessary to extend DSL with features specific to the design of such debuggers, and if this is necessary, what features should be included in the specification notation.

Further Extensions

The use of negative premisses extend the expressiveness of inductive definitions by allowing assertions of the form $a \notin A$. In [Gro89,BG90] Groote et al. proposes an extension to Structural Operational Semantics [Plo81] with negative premisses. An interesting investigation would be to extend Relational Semantics with negative premisses, and the aforementioned works offers a starting point.

The Relational Semantics formalism does not offer any facilities for modularisation of Relational Specifications. Such a facility is desirable when dealing with semantics of real programming languages where the number of rules can be large, e.g., the definition of Standard ML [HMT89]. Therefore, to include some notion of modules in the formalism is an interesting problem for future investigation. Once such a notion is defined, we foresee that the definition of equivalence between Relational Specifications will have to be extended to cope

with the modular structure of the specifications. The work of Schoett [Sch87] can be used as a starting point to investigate the required extensions.

The definition of Relational Specification can be generalised by replacing the Σ-algebra A in $(\Omega, \phi, \mathrm{A})$ by a *specification Sp* of a class of Σ-algebra containing A. This generalisation would allow any Σ-algebra that satisfies *Sp* to be used in the Relational Specification. This generalisation involves to define first how to write the specification *Sp* and then what it means for a Σ-algebra to satisfy a specification. There are many alternatives for these definitions, as discussed in [ST88, Wir89,BKL$^+$91]. We have not assessed the advantages and disadvantages of such a generalisation, but it seems an interesting problem for future investigation.

8.3 An Implementation of the Framework

A natural problem to be addressed is to give a machine implementation of the framework defined in this thesis. This framework is rich enough to give scope for various levels of implementation. We incrementally describe how such levels can be built starting with a simple system for prototyping of language implementations, and finishing with an integrated environment for language development that includes mechanised formal reasoning. In the presentation below, *the Framework* refers to the hypothetical implementation that is being suggested.

Language Prototyping System

The first component of the Framework is the implementation of a system to evaluate programs based on a Relational Specification. This would involve two main aspects. First, the syntax analysis of Relational Specifications according to Definition 2.1. Second, the implementation of the $\mathcal{CS}$ system according to Definition 5.1.

The resulting system would be similar to the prototyping facilities of systems like Centaur [CIK89] and the AnimatorGenerator [Ber91a]. The Framework

would depart from those approaches in that the *CS* system has a formal proof of correctness with respect to the underlying meaning of Relational Specifications given by the Declarative Semantics. Therefore, the prototyping facility of the Framework would also be correct with respect to this underlying meaning.

Meta-semantic Analysis and Transformations

A natural improvement on the prototyping facility described above is to include a *meta-semantic checker* that determines whether a Relational Specification is Data-driven and Deterministic in the sense of Definitions 5.2 and 5.4 respectively. Such a meta-semantic checker decides whether the evaluation of program formulae in the Framework is complete in the sense of Theorem 5.3.

Moreover, there are other static analysis that can be performed on Relational Specifications. For instance, checking whether a specification is Dynamically-deterministic in the sense of Definition 5.6. It is possible to include in the Framework a set of useful meta-semantic checkers which can be used to classify Relational Specifications.

Once we have the tools to automatically classify Relational Specification, an extension to the Framework is the implementation of transformation algorithms among classes of specifications. Such transformations are useful to extend the applicability of other tools in the Framework that only work over a given class, as discussed in Section 8.2.

In a series of papers, Hannan et al. described various transformations on semantic specifications [HM90,Han91b,Han91a]. It would also be interesting to investigate which, if any, of these transformations can be automated, and include their implementation in the Framework.

An Implementation of DSL

An implementation of the Framework as described above would possess useful features for language prototyping but no special facilities for debugger design.

The next stage in developing the Framework is to implement the notation DSL. This would add two features to the Framework: a powerful debugging language that can be used to debug Relational Specifications, and the capability of specifying and prototyping Interpreter-debuggers and Compiler-debuggers.

A General Purpose Implementation of DSL

A related problem is to study an implementation of DSL as a general purpose language for sequence manipulation. This implementation would extended the ideas of SNOBOL [FGP64,GPP68] for the manipulation of strings to the manipulation of arbitrary sequences. A first stage in the study of this problem is to generalise DSL sequence patterns to higher order sequences, so that we can express sequences of sequences. We believe it is possible to include DSL patterns in an existing functional language such as Standard ML [HMT89]. The implications of adding sequence patterns to ML would have to be studied.

8.4 Conclusions

In this thesis, we set out to examine the problems in the specification and the correctness proofs of compilers, and in the specification and the correctness proofs of debuggers. We proposed a framework that addresses specification, prototyping, implementation – in the sense of low level specification, and correctness proofs of such tools using a particular characterisation of Structural Operational Semantics as the underlying formalism. Although this framework was presented as an integrated theory, it is in fact composed of the Theory and its Application.

The Theory addresses the problem of correctness independently of the semantic formalism used in compiler specifications and specification language used in debugger specifications. It is composed of the material developed in Sections 3.3, and 7.2. There are two main results of the Theory. First, the extension of the notion of strong correspondence relation of [Sch87] to Model Correspondence between first order models, which is expressed by Theorem 3.1. This exten-

sion provides a proof method for Observational Equivalence between first order models. Second, the theory of debugger correctness of Section 7.2.

The Application part of this thesis addresses specification of programming language semantics, specification of compilers, correctness proofs of compiler, specification of Interpreter-debuggers and Compiler-debuggers, and correctness proofs of debuggers. A Relational Semantics of the programming language is used as a reference for compiler correctness and as a description of program evaluation on which the specifications of debuggers are based.

Two important results related to the Application are the definition of an Evaluation Model of programs based on Relational Semantics, and the definition of a debugger specification notation, called DSL, which can be used in the specification of concrete debuggers. The Evaluation Model and DSL can have actual implementations, yielding a prototyping system for programming languages, compilers, and debuggers. These implementations are not addressed in this thesis.

The main contributions and advances of this thesis are related to debugger correctness. We demonstrated that it is possible to give formal specifications of debuggers based on a formal semantics of the programming language. In practice, a given debugger may have more than one specification, tailored to different purposes. Related to this topic we examined the problem of Compiler-debuggers as an alternative specification of debuggers. We demonstrated that it is possible to prove the equivalence of specifications of a debugger, and, in particular, that it is possible to prove the correctness of Compiler-debuggers.

As far as we are aware of, the problems of specifying Compiler-debuggers, and of debugger correctness, have not been addressed before in the literature. Moreover, no other related work proposes a theory that addresses compiler and debugger correctness uniformly. These are the main, novel results of this thesis. We hope that these results will assist in the design of compilers and debuggers in practice, and also provide the basis of further research on the specification and correctness of other programming tools.

Bibliography

[AC90] I. Attali and J. Chazarain. Functional evaluation of natural semantics specifications. In *Proceedings International Conference WAGA on Attribute Grammar and their Applications. Lecture* Notes in Computer Science 461, pages 157–176. Springer-Verlag, September 1990.

[Acz77] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pages 739–782. North-Holland Publishing Company, 1977.

[AFZ88] I. Attali and P. Franchi-Zannettacci. Unification-free execution of TYPOL programs by semantic attribute evaluation. In *Proceedings Fifth International Conference on Logic Programming, Seattle.* MIT Press, August 1988.

[And91] James H. Andrews. *Logic Programming: Operational Semantics and Proof Theory.* PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, July 1991.

[Ast89] E. Astesiano. Inductive semantics. In *Lecture Notes of the State of the Art Seminar on Formal Description of Programming Concepts, Petropolis, Brazil.* IFIP TC2 WG 2.2, 1989.

[Ast91] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts.* Springer Verlag, 1991. To appear.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Publishing Company, 1986.

[Bar85] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition, 1985.

[BCG86] G. Berry, Ph. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to ESTEREL. In *Proceedings of the First France-Japan Symposium on Artificial Intelligence and Computer Science, Tokyo.* North-Holland, 1986. Also available as INRIA Research Report 647.

[Ber91a] D. Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, June 1991.

[Ber91b] Yves Bertot. Occurrences in debugger specifications. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, pages 327–337. ACM Press, June 1991.

[Ber92] Yves Bertot. Origin functions in λ-calculus and term rewriting systems. In *Proceedings 17th Colloquium on Trees in Algebra and Programming, Rennes, France*, pages 49–65. Springer-Verlag, February 1992. Lecture Notes in Computer Science 581.

[BF90] Clement A. Baker-Finch. The (algebraic) congruence of two programming language semantics. *Science of Computer Programming*, 14(1):81–96, June 1990.

[BG90] R. N. Bol and J. F. Groote. The meaning of negative premisses in transition system specifications. Technical Report CS-R9054,

Centre for Mathematics and Computer Science (CWI), Computer Science/Department of Software Technology, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, October 1990.

[BH83] B. Bruegge and P. Hibbard. Generalized path expressions: a high level debugging mechanism. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 34–44, August 1983. ACM Software Engineering Notes 8(4); ACM SIGPLAN Notices 18(8).

[BKL+91] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development*, volume 501 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BL69] R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: an experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, 1980.

[BMS87] R. Bahlke, B. Moritz, and G. Snelting. A generator for language-specific debugging systems. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques, SIGPLAN Notices 22(7)*, pages 92–101, July 1987.

[BOM81] B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting concepts to novices. *IJMMS*, 14:237–249, 1981.

[Bou85] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*. Cambridge University Press, 1985.

[Bov87] J. D. Bovey. A debugger for a graphical workstation. *Software-practice and Experience*, 17(9):647–662, September 1987.

[Boy91a] Johan Boye. Operational completeness of logic programs with external procedures. Master's thesis, The University of Linköping, S-581 83, Linköping, Sweden, January 1991. Reg nr: LiTH-IDA-Ex-9104.

[Boy91b] Johan Boye. S-SLD-resolution: An operational semantics for logic programming with external procedures. Research Report LiTH-IDA-R-91-18, The University of Linköping, Department of Computer and Information Science, S-581 83, Linköping, Sweden, June 1991. Also in Proceedings 3rd International Symposium on Programming Language Implementation and Logic Programming, Passau, Germany, August 26-28, 1991.

[BS86] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transaction on Programming Languages and Systems*, 8(4):547–576, October 1986.

[Bun88] A. Bundy. The use of explicit plan to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer Verlag, 1988. Longer version available from Edinburgh as DAI Reasearch Paper No. 349.

[BvHHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[CCM84] G. Cousineau, P-L. Curien, and M. Mauny. The Categorical Abstract Machine. In *Proceedings International Symposium on Semantics of Data Types, Sophia-Antipolis, France*, pages 50–64. Springer-Verlag, 1984. Lecture Notes in Computer Science, 173.

[CCP91] B. A. Cruz, G. L. M. Chaves, and A. Palma. The CPqD-Telebrás CHILL symbolic debugger. In *Proceedings 5th CHILL Conference, Rio de Janeiro, Brazil.* North-Holland, 1991.

[CH87] Benjamin B. Chase and Robert T. Hood. Selective interpretation as a technique for debugging computationally intensive programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques, SIGPLAN Notices 22(7)*, pages 113–124, July 1987.

[Chi89] C. Chin. A support tool for operational semantics. Undergraduate Project Report, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, May 1989.

[CIK89] Dominique Clément, Janet Incerpi, and Gilles Kahn. CENTAUR: Towards a "software tool box" for programming environments. In *Proceedings International Workshop on Environments, Chinon, France.* Springer-Verlag, September 1989. Lecture Notes in Computer Science 467.

[Cla79] K. L. Clark. Predicate logic as a computational formalism. Research monograph 79/59 TOC, Department of Computing, Imperial College, England, December 1979.

[CM86] L. M. Chirica and D. F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, third edition, 1987.

[Coh65] Paul M. Cohn. *Universal Algebra.* New York Harper and Row, 1965.

[Coh78] Avra Cohn. High level proof in LCF. Internal Report CSR-35-78, Department of Computer Science, University of Edinburgh, November 1978.

[dB72] N. G. de Bruijn. Lambda-calculus notion with nameless dummies. *Indag Math.*, pages 381–392, 1972.

[Des86] J. Despeyroux. Proof of translation in Natural Semantics. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 193–205. IEEE Computer Society Press, 1986.

[Des88] T. Despeyroux. TYPOL: a formalism to implement Natural Semantics. Technical Report 94, INRIA, Sophia-Antipolis, France, March 1988.

[DF87] Pierre Deransart and Gérard Ferrand. An operational formal definition of PROLOG. In *Proceedings of the Symposium on Logic Programming, San Francisco, California*, pages 162–172. IEEE Press, August 1987.

[Dia76] Cora Diamond, editor. *Wittgenstein's Lectures on the Foundations of Mathematics, Cambridge 1939.* The University of Chicago Press, 1976.

[DM85] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2:119–155, 1985.

[dS90] Fabio Q. B. da Silva. Towards a formal framework for the evaluation of operational semantics specifications. Technical Report ECS-LFCS-90-126, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, November 1990. Available from Lorraine Edgar (`lme@dcs.ed.ac.uk`) or in writing to the Department of Computer Science.

[FGP64] D. J. Farber, R. E. Griswold, and I. P. Polonsky. SNOBOL, a string manipulation language. *Journal of the ACM*, 11:21–30, January 1964.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* International Computer Science Series. Addison-Wesley Publishing Company, 1988.

[GB85] M. E. Garcia and W. J. Berman. An approach to concurrent systems debugging. In *Proceedings Fifth International Conference Distributed Computing Systems, Denver, CO*, pages 507–514, May 1985.

[GB90] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. Technical Report ECS-LFCS-90-106, LFCS, Department of Computer Science, University of Edinburgh, January 1990.

[Gen69] G. Gentzen. *The Collected Papers of Gerard Gentzen.* North-Holland, Amsterdam, 1969. E. Szabo (ed.).

[GGM76] V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specification. In *Proceedings 5th Symposium on Mathematical Foundations of Computer Science, Gdansk.* Springer-Verlag, 1976. Lecture Notes in Computer Science, 45.

[GH77] R. E. Griswold, , and D. R. Hanson. An overview of SL5. *SIGPLAN Notices*, 12(5):40–50, April 1977.

[GH80] R. E. Griswold and D. R. Hanson. An alternative to the use of patterns in string processing. *ACM Transactions on Programming Languages and Systems*, 2(2):153–172, April 1980.

[GM86] Narain Gehani and Andrew McGettrick, editors. *Software Specification Techniques.* International Computer Science Series. Addision-Weslley Publishing Company, 1986.

[GPP68] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language.* Englewood Cliffs (N.J.), Prentice Hall, 1968.

[Grä79] G. Grätzer. *Universal Algebra.* Springer-Verlag, second edition, 1979.

[Gri71] David Gries. *Compiler Construction for Digital Computers.* Wiley International Editions. John Wiley and Sons, 1971.

[Gro89] J. F. Groote. Transition system specifications with negative premisses. Technical Report CS-R8950, Centre for Mathematics and Computer Science (CWI), Computer Science/Department of Software Technology, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, December 1989.

[GTWW77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.

[Hal74] Paul R. Halmos. *Naive Set Theory.* Undergraduate Texts in Mathematics. Springer-Verlag, second edition, 1974.

[Han91a] John Hannan. Making abstract machines less abstract. In *Proceeding 5th ACM Conference on Functional Programming and Computer Architecture*, pages 618–635. Springer-Verlag, 1991. Lecture Notes in Computer Science, 523.

[Han91b] John Hannan. Staging transformations for abstract machines. In J. Hughes, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulation.* ACM Press, 1991.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *ACM Transaction on Programming Languages and Systems*, 4(3):324–344, July 1982.

[Hen90a] Matthew Hennessy. *The Semantics of Programming Languages: an elementary introdcution using structural operational semantics.* John Wiley and Sons Ltd., 1990.

[Hen90b] Rolf Hennicker. Context induction: a proof principle for behavioural abstraction. In A. Miola, editor, *Proceedings International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 101–110. Springer-Verlag, April 1990. Lecture Notes in Computer Science, 429.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM, to appear*, 1987. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[HM90] John Hannan and Dale Miller. From operational semantics to abstract machines: preliminary results. In *ACM Conference on LISP and Functional Programming*, pages 323–332, July 1990.

[HMT89] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML (version 3). Technical Report ECS-LFCS-89-81, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, May 1989.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418. IEEE Computer Society Press, 1992.

[HW90] P. Hudak and P. Wadler. Report on the programming language haskell - a non-strict purely function language. Technical Report YALEU/DCS/RR-777, Yale University, April 1990.

[Jon77] C. Jones. Program quality and programmer productivity. Technical Report IBM TR 02.764, IBM, Santa Teresa Laboratory, San Jose, CA, 1977.

[Joy89] Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, Computer Laboratory, University of Cambridge, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, March 1989.

[Kah87] G. Kahn. Natural Semantics. Gipe project second annual review report, INRIA, Sophia-Antipolis, France, January 1987.

[Kah88] G. Kahn. Natural Semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. North-Holland Publishing Company, 1988.

[KHC91] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring Semantics: a formal framework for specifying, implementing and reasoning about execution monitors. In *Proceedings ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, pages 338–352, June, 26–28 1991.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N. J., 1978.

[Kun80] Kenneth Kunen. *Set Theory: an introduction to independence proofs*, volume 102 of *Studies in Logic and the Foundations of Mathematics.* North Holland Publishing Company, 1980.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lau79] Søren Lausen. Debugging techniques. *Software-practice and experience*, 9:51–63, 1979.

[Lew82] T. G. Lewis. *Software Engineering: Analysis and Verification.* Reston Publishing Company Inc., 1982.

[LL89] Beatrice Lazzerini and Lanfranco Lopriore. Abstraction mechanism for event control in program debugging. *IEEE Transactions on Software Engineering*, 15(7):890–901, July 1989.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer Series Symbolic Computation - Artificial Intelligence. Springer-Verlag, second edition, 1987.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[Mic] Sun Microsystems. Dbx - source level debugger. Manual pages, Sun Release 4.1.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *Second International Joint Conference on Artificial Intelligence*, pages 481–489, London, 1971. The British Computer Society.

[Mil76] Robin Milner. Program semantics and mechanised proof. In *Foundations of Computer Science II*, pages 3–44. Math. Centre Amsterdam Tracts 82, 1976.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[Moh88] Thomas G. Moher. PROVIDE: a process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[Mor73] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, October 1973.

[Mos79] P. D. Mosses. A construtive approach to compiler correctness. Technical report, Aarhus University, 1979.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. Schwartz, editor, *Proceedings Symposium on Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[MW72] Robin Milner and R. Weyhauch. Proving compiler correctness in a mechanised logic. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, chapter 3, pages 51–70. Edinburgh University Press, 1972.

[Nai83a] Lee Naish. Automatic generation of control for logic programs. Technical Report 83/6, Department of Computer Science, Melbourne University, 1983.

[Nai83b] Lee Naish. Mu-prolog 3.0 reference manual. Technical report, Department of Computer Science, Melbourne University, July 1983.

[Nai85] Lee Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer Verlag, 1985.

[Nip86] Tobias Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[NO88] P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specifications. In *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types, Gullane, Scotland*, pages 184–207. Springer-Verlag, 1988. Lecture Notes in Computer Science, 332.

[Nor82] D. A. Norman. Some observations on mental models. In A. Stevens, editor, *Mental Models*, pages 7–19. Erlbaum, 1982.

[Par81] D. M. R. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference, Karlsuhe*, pages 167–183. Springer-Verlag, March 1981. Lecture Notes in Computer Science 104.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[Pol81] Wolfgang H. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science*. Springer Verlag, 1981.

[PP92] Thomas Pittman and James Peters. *The Art of Compiler Design: Theory and Practice*. Prentice-Hall, Inc., 1992.

[Pra65] D. Prawitz. *Natural Deduction, a proof-theoretical study*. Almqvist & Wiksell, Stockholm, 1965.

[Rei81] H. Reichel. Behavioural equivalence: a unifying concept for initial and final specification methods. In *Proceedings 3rd Hungarian Computer Science Conference*, pages 27–39, 1981.

[Rob65] J. A. Robinson. A machine-oriented logic based on resolution principle. *Journal of the ACM*, pages 23–41, January 1965.

[Sch87] Oliver Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, February 1987.

[Sch90] Oliver Schoett. Behavioural correctness of data representation. *Science of Computer Programming*, 14:43–57, 1990.

[Sev87] Rudolph E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4(3):20–32, 1987 1987.

[Sho83] M. L. Shooman. *Software Engineering.* New York: McGraw-Hill, 1983.

[Sim90] Todd Simpson. Correctness of a compiler specification for the SECD machine. Research Report 90/410/34, Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada, T2N 1N4, October 1990.

[ST87] D. Sannella and A. Tarleck. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[ST88] D. Sannella and A. Tarlecki. Specification in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, February/March 1988.

[Sto81] Joseph E. Stoy. The congruence of two programming language semantics. *Theoretical Computer Science*, 13:151–174, 1981.

[Sto89] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey approach to programming language theory.* The MIT Press series in Computer Science. The MIT Press, 1989.

[SYO87] S. K. Skedzielewski, R. K. Yates, and R. R. Oldehoeft. DI: an interactive debugging interpreter for applicative languages. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques, SIGPLAN Notices 22(7)*, pages 102–112, July 1987.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* Computer Science Series. McGraw-Hill International Editions, 1985.

[TWW78] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Notes on algebraic fundamentals for theoretical computer science. In *Proceedings FAICS '78*, May 1978.

[TWW81] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More on advice on structuring compilers and proving them correct. *Theorectical Computer Science*, 15:223–249, 1981.

[vdLW85] Frits van der Linden and Ian Wilson. An interactive debugging environment. *IEEE Micro*, 5(4):18–31, August 1985.

[vW69] A. van Wijngaarden. Report on the algorithmic language ALGOL 68. *Numer. Math.*, 14:19–218, 1969.

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892, April 1991.

[Wir89] Martin Wirsing. Algebraic specification. Technical Report MIP-8914, Universität Passau, Fakultät für Mathematik und Informatik, Universität Passau, Postfach 2540, D-8390, Germany, 1989.

[Zel84] Polle Trescott Zellweger. Interactive source-level debugging of optimized programs. Technical Report CSL-84-5,[P84-00047], Xerox PARC, Palo Alto, California, May 1984.

[ZJ91] Lawrence W. Zurawski and Ralph E. Johnson. Debugging optimized code with expected behaviour. University of Illinois at Urbana-Champaign., August 1991.

Appendix A

The Semantics of DSL

A.1 Introduction

In this appendix we present the formal semantics of DSL in Relational Semantics. This semantics defines the pattern matching of sequence patterns on sequences of basic values. Basic values, and consequently basic patterns, are not defined since their definitions depend on the application in which DSL is used.

In this thesis, DSL is used in the specification of debuggers based on a Relational Specification. In this application, the basic values and basic patterns of DSL are defined by the Relational Specification. As described in Section 6.5, these basic values are constructor values and the basic patterns are constructor patterns whose semantics are standard and can be found, for instance, in [FH88, HMT89].

This appendix is organised as follows:

Section A.2 defines the *syntax* of DSL.

Section A.3 defines the *dynamic semantics* of the pattern matching.

atpat	::=	_	wildcard
		id	variable
baspatseq	::=	[*baspatseq*,] *baspat*	basic pattern sequence
spat	::=	...	wildcard
		< [*baspatseq*] >	sequence of basic patterns
		$\iota(spat_1, spat_2, spat_3, spat_4, spat_5)$	interval
		$\beta(spat_1, spat_2, spat_3, spat_4, spat_5)$	balanced interval
		id **as** *spat*	layered

Figure 1: The Syntax of Sequence Patterns

A.2 The Syntax

Reserved Words

These are the reserved used in sequence patterns of DSL.

```
as    (    )    ,    ...    _    <    >
```

Identifiers

A DSL identifier is any sequence of letters, digits, and "_" (underbars) starting with a letter. The class of DSL identifiers is called Identifier and is ranged over by the meta-variable *id*.

The Grammar

Figure 1 presents the BNF rules that define *sequence patterns*. The grammar for *baspat* is not defined since it depends on the application of DSL. When DSL is used for specification of debuggers based on a Relational Specification the basic patterns are constructor patterns, whose syntax is defined by the signature of the specification. In grammar rules, square brackets [ ] enclose optional phrases.

v	$\in$	Val=BasVal+BasVal*	basic value
bv	$\in$	BasVal	basic value
w	$\in$	BasVal*	sequence of basic values
E	$\in$	Env $= [id \xrightarrow{\text{fin}} \text{Val}]$	dynamic environment
ES	$\in$	EStack	environment stack
FAIL			matching failure

Figure 2: The Semantic Objects of the Dynamic Semantics

A.3 The Dynamic Semantics

The dynamic semantics defines how to match a pattern to a value. We first define the semantic objects involved. As for the syntax, the basic values are not define. We assume the existence of a semantic class BasVal of basic values.

When DSL is used for specification of debuggers, basic values are the states of the Computational Semantics of the Relational Specification under consideration. Therefore, basic values are built from substitutions, goal stacks, formulae, and terms. For an arbitrary Relational Specification $\mathcal{S}$ with Computational Semantics $\mathcal{CS} = (\Gamma_{\mathcal{CS}}, \overset{\mathcal{CS}}{\Rightarrow}, \mathcal{T}_{\mathcal{CS}})$, BasVal $= \Gamma_{\mathcal{CS}}$. In this case, sequence values are debugging histories in DH.

Semantic Objects

The semantic objects of the dynamic semantics are shown in Figure 2. FAIL is a semantic object used in the Relational Rules to indicate an unsuccessful matching; it is not a semantic value. Environment stacks are defined as follows:

ES	::=	ε	empty stack
		$ES \bullet E$	environment stack

Semantic Operations on Environments and Sequences

If E and E' are enviroment the environment $E \oplus E'$, called E *modified by* E', is the environment with domain dom $E \cup$ dom E' such that:

$$(E \oplus E')(id) = \begin{cases} E'(id) & \text{if } id \in \text{dom } E' \\ E(id) & \textit{otherwise} \end{cases}$$

We extend environment modification to account for matching failure as follows: $E \oplus \text{FAIL} = \text{FAIL}$. We also use the operations on finite sequences "$\diamond$" suffix cons, "$\frown$" concatenation, and "-" difference.

Relational Rules

The Relational Rules below define the Dynamic Semantics of sequence patterns. To simplify the presentation we made some concessions with respect to the definition of Relational Rules in Chapter 2:

1. The relation name $\_ \vdash \_ \Rightarrow \_$ is overloaded for all syntactical classes.

2. Phrases within square brackets [ ] are *optional* and for them the following convention holds: in each instance of every Relational Rule the optional phrases must be either all present or all absent.

3. Similarly, a phrase enclosed by [[ ]] is a *second optional phrase.* As for the first optional phrases, in each instance of every rule the second optional phrases must be all present or all absent.

4. We allow y/FAIL to range over Y $\cup$ {FAIL}, where Y is one of the above defined semantic classes and y ranges over Y.

Comments explaining the semantic rules are given after each subset of the rules. The reader is referred to the schematic representation of a sequence in Figure 3 to understand how the matching of interval and balanced interval patterns is defined.

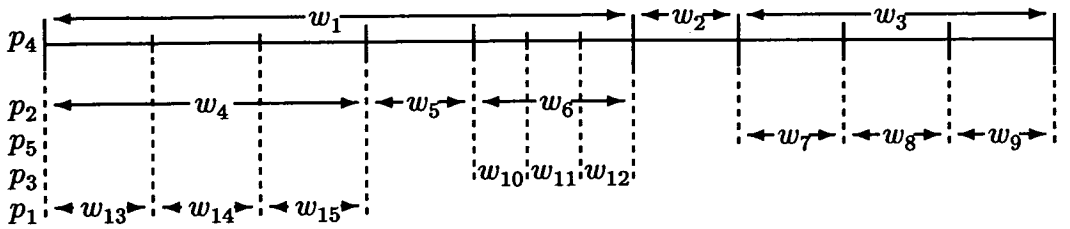

Figure 3: The Matching of Interval Patterns

Atomic Patterns $\boxed{E, v \vdash atpat \Rightarrow E'/FAIL}$

$$\frac{}{E, v \vdash \_ \Rightarrow \{\,\}} \tag{1}$$

$$\frac{id \notin \text{dom } E}{E, v \vdash id \Rightarrow \{id \mapsto v\}} \tag{2}$$

$$\frac{id \in \text{dom } E \qquad E(id) = v}{E, v \vdash id \Rightarrow \{\,\}} \tag{3}$$

$$\frac{id \in \text{dom } E \qquad E(id) \neq v}{E, v \vdash id \Rightarrow \text{FAIL}} \tag{4}$$

Comments

(1)–(4) These rules define the matching of an atomic pattern *atpat* on a value v. The matching is defined on an environment E which keeps the bindings of identifiers to values that resulted from previous matchings. If the matching is successful, the result is an environment E' with the new binding, if applicable. If the matching fails the result is FAIL.

(2) A finite mapping is denoted by $\{id_1 \mapsto v_1, \ldots, id_n \mapsto v_n\}$, $n \geq 0$

(3)–(4) Identifiers can occur more than once in a pattern. These rules ensure the same identifier matches syntactically equal values.

Sequence of Basic Patterns $\boxed{E, w \vdash \mathit{baspatseq} \Rightarrow (E', w')/\mathrm{FAIL}}$

$$\frac{}{E, \langle\,\rangle \vdash [\mathit{baspatseq},]\mathit{baspat} \Rightarrow \mathrm{FAIL}} \tag{5}$$

$$\frac{E, v \vdash \mathit{baspat} \Rightarrow \mathrm{FAIL}}{E, [[w\diamond]]v \vdash [\mathit{baspatseq},]\mathit{baspat} \Rightarrow \mathrm{FAIL}} \tag{6}$$

$$\frac{E, v \vdash \mathit{baspat} \Rightarrow E'}{E, [w\diamond]v \vdash \mathit{baspat} \Rightarrow (E', \langle v\rangle)} \tag{7}$$

$$\frac{E, v \vdash \mathit{baspat} \Rightarrow E' \qquad E \oplus E', w \vdash \mathit{baspatseq} \Rightarrow (E'', w')/\mathrm{FAIL}}{E, w \diamond v \vdash \mathit{baspatseq}, \mathit{baspat} \Rightarrow (E' \oplus E'', w' \diamond v)/\mathrm{FAIL}} \tag{8}$$

Comment

(5)–(8) These rules define the matching of a sequence of basic patterns *baspatseq* on a sequence of basic values w. If the matching is successful the results are the new environment and the sufix of w that matched *baspatseq*. The sub-sequence resulting from this matching is used in rule (13).

Sequence Patterns $\boxed{E, w \vdash \mathit{spat} \Rightarrow (E', (w_1, w_2, w_3))/\mathrm{FAIL}}$

$$\frac{}{E, w \vdash \ldots \Rightarrow (\{\,\}, (\langle\,\rangle, w, \langle\,\rangle))} \tag{9}$$

$$\frac{}{E, \langle\,\rangle \vdash \mathrm{<>} \Rightarrow (\{\,\}, (\langle\,\rangle, \langle\,\rangle, \langle\,\rangle))} \tag{10}$$

$$\frac{}{E, w \diamond v \vdash \mathrm{<>} \Rightarrow \mathrm{FAIL}} \tag{11}$$

$$\frac{\begin{array}{c} E, w \diamond v \vdash \mathit{baspatseq} \Rightarrow \mathrm{FAIL} \\ E, w \vdash \mathrm{<}\mathit{baspatseq}\mathrm{>} \Rightarrow (E', (w_1, w_2, w_3))/\mathrm{FAIL} \end{array}}{E, w \diamond v \vdash \mathrm{<}\mathit{baspatseq}\mathrm{>} \Rightarrow (E', (w_1, w_2, w_3 \diamond v))/\mathrm{FAIL}} \tag{12}$$

$$\frac{E, w \vdash \mathit{baspatseq} \Rightarrow (E', w')/\mathrm{FAIL}}{E, w \vdash \mathrm{<}\mathit{baspatseq}\mathrm{>} \Rightarrow (E', w - w', w', \langle\,\rangle))/\mathrm{FAIL}} \tag{13}$$

Comments

(9)–(13) These rules define the simple sequence patterns ... and <[*baspatseq*]>. The results of a succesful pattern matching are the environment and a triple (w_1, w_2, w_3) of consecutive sub-sequences of w, which is interpreted as follows: w_2 is the sub-sequence of w that matched the pattern, w_1 is the sub-sequence on the left of w_2, and w_3 is the sub-sequence on the right of w_2, such that $w = w_1 \hat{} w_2 \hat{} w_3$. The three sub-sequences are necessary in the definition of the interval patterns.

(12) If the first attempt to match the *spat* fails, then the matching is moved to one element to the left on the original sequence.

Interval Patterns

In the following rules we use p (with indexes) ranging over *spat* in order to simplify the presentation.

$$\frac{E, w \vdash p_4 \Rightarrow \text{FAIL}}{E, w \vdash \iota(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{14}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E \oplus E_1, w_1 \vdash p_2 \Rightarrow \text{FAIL} \end{array}}{E, w \vdash \iota(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{15}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E \oplus E_1, w_1 \vdash p_2 \Rightarrow (E_2, (w_4, w_5, w_6)) \\ E \oplus E_1 \oplus E_2, w_3 \vdash p_5 \Rightarrow \text{FAIL} \end{array}}{E, w \vdash \iota(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{16}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E \oplus E_1, w_1 \vdash p_2 \Rightarrow (E_2, (w_4, w_5, w_6)) \\ E \oplus E_1 \oplus E_2, w_3 \vdash p_5 \Rightarrow (E_3, (w_7, w_8, w_9)) \\ E \oplus E_1 \oplus E_2 \oplus E_3, w_6 \vdash p_3 \Rightarrow \text{FAIL} \end{array}}{E, w \vdash \iota(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{17}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E \oplus E_1, w_1 \vdash p_2 \Rightarrow (E_2, (w_4, w_5, w_6)) \\ E \oplus E_1 \oplus E_2, w_3 \vdash p_5 \Rightarrow (E_3, (w_7, w_8, w_9)) \\ E \oplus E_1 \oplus E_2 \oplus E_3, w_6 \vdash p_3 \Rightarrow (E_4, (w_{10}, w_{11}, w_{12})) \\ E \oplus E_1 \oplus E_2 \oplus E_3 \oplus E_4, w_4 \vdash p_1 \Rightarrow (E_5, (w_{13}, w_{14}, w_{15}))/\text{FAIL} \end{array}}{\begin{array}{l} E, w \vdash \iota(p_1, p_2, p_3, p_4, p_5) \Rightarrow \\ \quad (E_1 \oplus E_2 \oplus E_3 \oplus E_4 \oplus E_5, (w_{13}, w_{14}\hat{}w_{15}\hat{}w_5\hat{}w_6\hat{}w_2\hat{}w_7\hat{}w_8, w_9)/\text{FAIL} \end{array}} \tag{18}$$

Comments

(14)–(18) These rules define the interval patterns. This definition guarantees that the pattern matching is deterministic because it forces the patterns to be tried in a fixed order. In the rules, this order is given by the input/output relation on the premisses, as defined in Section 5.4 by the relation $\preceq$.

(14) The pattern matching must start from the pattern p_4. If the matching of p_4 fails then the whole matching fails.

(15) Once the matching of p_4 is fixed and results in (w_1, w_2, w_3) then p_2 is tried on w_1. If the matching of p_2 fails the whole matching fails. Notice that when this happens there is no attempt to try the matching of p_4 again.

(18) This rules describes the successful matching. Once the matching of p_2 and p_4 are fixed, the order in which the other patterns are tried is not important, since w_3, w_4, and w_6 are fixed. The environment resulting from the

matching is the union of the evaluation resulting from each sub-matching. The resulting triple of sub-sequences is better understood by referring to Figure 3.

Balanced Interval Patterns

$$\frac{E, w \vdash p_4 \Rightarrow \text{FAIL}}{E, w \vdash \beta(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{19}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E, \varepsilon \bullet E_1, w_1 \vdash_{\text{B}} p_2, p_4 \Rightarrow \text{FAIL} \end{array}}{E, w \vdash \beta(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{20}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E, \varepsilon \bullet E_1, w_1 \vdash_{\text{B}} p_2, p_4 \Rightarrow (E_2, (w_4, w_5, w_6)) \\ E \oplus E_1 \oplus E_2, w_3 \vdash p_5 \Rightarrow \text{FAIL} \end{array}}{E, w \vdash \beta(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{21}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E, \varepsilon \bullet E_1, w_1 \vdash_{\text{B}} p_2, p_4 \Rightarrow (E_2, (w_4, w_5, w_6)) \\ E \oplus E_1 \oplus E_2, w_3 \vdash p_5 \Rightarrow (E_3, (w_7, w_8, w_9)) \\ E \oplus E_1 \oplus E_2 \oplus E_3, w_6 \vdash p_3 \Rightarrow \text{FAIL} \end{array}}{E, w \vdash \beta(p_1, p_2, p_3, p_4, p_5) \Rightarrow \text{FAIL}} \tag{22}$$

$$\frac{\begin{array}{c} E, w \vdash p_4 \Rightarrow (E_1, (w_1, w_2, w_3)) \\ E, \varepsilon \bullet E_1, w_1 \vdash_{\text{B}} p_2, p_4 \Rightarrow (E_2, (w_4, w_5, w_6)) \\ E \oplus E_1 \oplus E_2, w_3 \vdash p_5 \Rightarrow (E_3, (w_7, w_8, w_9)) \\ E \oplus E_1 \oplus E_2 \oplus E_3, w_6 \vdash p_3 \Rightarrow (E_4, (w_{10}, w_{11}, w_{12})) \\ E \oplus E_1 \oplus E_2 \oplus E_3 \oplus E_4, w_4 \vdash p_1 \Rightarrow (E_5, (w_{13}, w_{14}, w_{15}))/\text{FAIL} \end{array}}{\begin{array}{l} E, w \vdash \beta(p_1, p_2, p_3, p_4, p_5) \Rightarrow \\ \quad (E_1 \oplus E_2 \oplus E_3 \oplus E_4 \oplus E_5, (w_{13}, w_{14} \hat{} w_{15} \hat{} w_5 \hat{} w_6 \hat{} w_2 \hat{} w_7 \hat{} w_8, w_9)/\text{FAIL} \end{array}} \tag{23}$$

Comment

(14)–(18) These rules define the matching of balanced interval patterns. The major difference between these rules and those for the interval patterns is that the matching of p_2 on w_1 is defined by rules (24)–(27) which check if the sequence w_1 is balanced with respect to p_2 and p_4.

Balanced Matching $\boxed{E, ES, w \vdash_{\mathrm{B}} spat_1, spat_2 \Rightarrow (E', (w_1, w_2, w_3))/\mathrm{FAIL}}$

$$\frac{ES \neq \varepsilon}{E, ES, \langle\,\rangle \vdash_{\mathrm{B}} spat_1, spat_2 \Rightarrow \mathrm{FAIL}} \tag{24}$$

$$\frac{\begin{array}{c} E, w \diamond v \vdash spat_2 \Rightarrow \mathrm{FAIL} \\ E \oplus E', w \diamond v \vdash spat_1 \Rightarrow (E'', (w_1, w_2, w_3)) \end{array}}{E, \varepsilon \bullet E', w \diamond v \vdash_{\mathrm{B}} spat_1, spat_2 \Rightarrow (E'', (w_1, w_2, w_3))/\mathrm{FAIL}} \tag{25}$$

$$\frac{\begin{array}{c} ES \neq \varepsilon \qquad E, w \diamond v \vdash spat_2 \Rightarrow \mathrm{FAIL} \\ E \oplus E', w \diamond v \vdash spat_1 \Rightarrow (E'', (w_1, w_2, w_3)) \\ E, ES, w_1 \vdash_{\mathrm{B}} spat_1, spat_2 \Rightarrow (E''', (w_4, w_5, w_6))/\mathrm{FAIL} \end{array}}{E, ES \bullet E', w \diamond v \vdash_{\mathrm{B}} spat_1, spat_2 \Rightarrow (E''', (w_4, w_5, w_6 \hat{} w_2 \hat{} w_3))/\mathrm{FAIL}} \tag{26}$$

$$\frac{\begin{array}{c} E \oplus E', w \diamond v \vdash spat_2 \Rightarrow (E'', (w_1, w_2, w_3)) \\ E, ES \bullet E' \bullet E'', w_1 \vdash_{\mathrm{B}} spat_1, spat_2 \Rightarrow (E''', (w_4, w_5, w_6))/\mathrm{FAIL} \end{array}}{E, ES \bullet E', w \diamond v, \vdash_{\mathrm{B}} spat_1, spat_2 \Rightarrow (E''', (w_4, w_5, w_6 \hat{} w_2 \hat{} w_3))/\mathrm{FAIL}} \tag{27}$$

Comments

(24)–(27) These rules define the matching of $spat_1$ on w with the extra-condition that the matching is balanced with respect to $spat_2$ in the usual sense of balanced bracketing. For this definition we use a stack to keep the environments resulting from the matchings of $spat_2$. This stack is necessary because the matching of $spat_1$ must be done on the environment resulting from the matching of a corresponding $spat_2$.

(25) If $spat_1$ matches successfully and the environment stack has only one element then the matching is balanced and the overall matching succeeds.

(26) If $spat_1$ matches successfully but the stack has more than one element then the top of the stack is removed and the matching continues with the new stack.

(27) If $spat_2$ matches successfully then the environment resulting from its matching is pushed on the stack and the matching for the corresponding $spat_1$ is attempted.

A.4 Summary and Conclusions

In this appendix, we defined the semantics of DSL. We presented the syntax of *sequence patterns* and the Relational Rules for the dynamic semantics of the pattern matching. We did not define the syntax and semantics of the basic values and basic patterns. These definitions depend on the application in which DSL is used. We informally discussed the definition of the basic values for the application of DSL in the specification of debuggers based on a Relational Specification.

An actual implementation of DSL could be achieved by adding sequence patterns to an existing functional language. In particular, we believe that it is simple to include these patterns in the ML language [HMT89]. The implications of adding sequence patterns to ML would have to be studied.