Observational Equivalence and Compiler Correctness

Fabio Q. B. da Silva
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh,
Scotland, U.K.,
e-mail: fabio@dcs.ed.ac.uk

September 18, 1992

Abstract

In this work, we study the problem of giving proofs of compiler correctness in the framework of observational equivalence [ST87] and correspondences [Sch87, Sch90]. The starting point for correctness is a formal semantics of the programming language given in a form of Structural Operational Semantics [Plo81] wich we call *Relational Semantics* [dS92, Chapter 2]. Intuitively, the problem of compiler correctness 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 start by introducing Relational Semantics. We then define an equivalence relation between Relational Semantics based on the notion of observational equivalence [ST87], and extend the proof method of correspondences [Sch87, Sch90] to a proof method for observational equivalence of Relational Semantics, which we call *Model Correspondence*. The main theoretical result of this work shows that Model Correspondence is sound and complete with respect to observational equivalence and therefore can be used as a proof method.

Finally, we use observational equivalence to establish the equivalence between a (standard) semantics of the programming language and another semantics defined using a compilation of the language into some machine code. This notion of equivalence defines a criterion for compiler correctness and we shall argue why this is a suitable criterion. We also show a proof of compiler correctness that illustrates the use of Model Correspondence.

1 Introduction

The problem of proving the correctness of compilers is obviously of great importance. We need such proofs to acquire the confidence that the result we obtain from executing the compiled code of a program on a machine is consistent with the result of the program according to the language's definition. Since there is 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. In the rest of this section we outline the main approaches to compiler correctness that we find in the literature, and show how our approach extends and improves these previous ideas.

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.

$$\begin{tikzcd} \underline{L} \arrow[r, "\text{compiler}", "\gamma"'] \arrow[d, "\pi"] & \underline{T} \arrow[d, "\mu"] \\ \underline{M} & \underline{U} \arrow[l, "\delta", "\text{decoding}"'] \end{tikzcd}$$

Figure 1: The Morris Diagram

The algebraic approach was further developed in [Mor73], where compiler correctness is characterised as the commutativity of the diagram in Figure 1 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 $\underline{\mathrm{L}}$ 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.

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 functional languages, function expression evaluate to function values. For instance, the function value of the expression **fn x . x + 1** could be represented as $(\varepsilon_E, \mathbf{x}, \mathbf{x}+1)$, and the value of **fn y . y + 1** could be represented as $(\varepsilon_E, \mathbf{y}, \mathbf{y}+1)$. 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. This example will be fully illustrated later on.

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 $\epsilon : \underline{M} \rightarrow \underline{U}$ (of encoding) to replace the homomorphism $\delta : \underline{U} \rightarrow \underline{M}$ 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 $\epsilon : \underline{M} \rightarrow \underline{U}$ 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 our work.

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 as the function values.

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 are discussed in [Sch87, page 255], where Schoett gives a proof that observational equivalence is more general than representation relation. Therefore, we propose to use observational equivalence as the criterion for compiler correctness.

Our approach to correctness is an improvement on previous approaches since it gives a more general, and yet intuitively sufficient, correctness criterion. Furthermore, it is based on a formal definition of equivalence which we can reason 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.

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 our approach to the compiler correctness problem is that we define a proof method based on correspondence relations [Sch87, Sch90]. This proof method 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.

The rest of this paper is organised as follows. In Section 2 we present some basic concepts of Universal Algebras. Section 3 defines the formalism of Relational Semantics. In Section 4 we defined an observational equivalence relation between Relational Semantics and extend the proof method of correspondence [Sch87, Sch90] to a proof method, called *Model Correspondence*, for proofs of equivalence of Relational Semantics. We then prove that Model Correspondence is sound and complete with respect to observational equivalence and therefore can be used as a proof method.

In Section 5 we study the problem of giving proofs of compiler correctness in the framework of Relational Semantics and use observational equivalence as a criterion for compiler correctness. In Section 6 we present a proof of compiler correctness using Model Correspondence. Finally, in Section 7 we present some conclusions and directions for future work.

2 Definitional Preliminaries

The set theory of this work is the axiomatic set theory of Zermelo-Fraenkel, whose formalisation can be found for example in [Hal74, Kun80]. We assume the reader is familiar with the basic concepts of set theory. We also assume familiarity with the basic concepts of the theory of inductive definitions; these concepts can be found in [Acz77]. The algebraic concepts used hereafter are standard definitions in Universal Algebra which can be found for instance in [Coh65, TWW78, Grä79, Wec92]. The definition of first order signature can be found in [GB90].

A (many sorted) *algebraic signature* is a pair (S, F) where S is a set (of *sorts* or *type names*) and F is a $\mathrm{S}^* \times \mathrm{S}$-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 $(\mathrm{S}, \mathrm{F}, \Pi)$, 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*.

Let $\Omega = (\mathrm{S}, \mathrm{F}, \Pi)$ and $\Omega' = (\mathrm{S}', \mathrm{F}', \Pi')$, a *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 $\rho_{\mathrm{F}}(\sigma) : \rho_{\mathrm{S}}(s_1) \times \ldots \times \rho_{\mathrm{S}}(s_n) \rightarrow \rho_{\mathrm{S}}(s)$ and $\rho_{\Pi}(\pi) : \rho_{\mathrm{S}}(s_1) \times \ldots \times \rho_{\mathrm{S}}(s_n)$. 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 $\Omega \sqsubset \Omega'$.

As a notational convention, whenever $\sigma \in \mathrm{F}_{s_1 \ldots s_n, s}$, 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 $s_1 \ldots s_n \in \mathrm{S}^*$ and $s \in \mathrm{S}$ 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 $s \in \mathrm{S}$, the set $T_{\mathrm{X}}(\Sigma)_s$ 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 $x \in \mathrm{X}_s$, $x \in T_{\mathrm{X}}(\Sigma)_s$
2. For all $\sigma \in \mathrm{F}_{s_1 \ldots s_n, s}$, $n \geq 0$:
   if $t_i \in T_{\mathrm{X}}(\Sigma)_{s_i}$, for $i \in [n_+]$, then $\sigma(t_1, \ldots, t_n) \in T_{\mathrm{X}}(\Sigma)_s$

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 $\overline{t}$ to emphasise that it is a ground term. The set $T_{\mathrm{X}}(\Sigma)$ (respectively $T(\Sigma)$) denotes the family $\{\, T_{\mathrm{X}}(\Sigma)_s\}_{s\in\mathrm{S}}$ (respectively $\{\, T(\Sigma)_s\}_{s\in\mathrm{S}}$). A *ground instantiation* of a term t is a ground term obtained from t by simultaneously replacing every meta-variable in t by a ground term.

Every first order signature $\Omega = (\mathrm{S}, \mathrm{F}, \Pi)$ 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 $w, w' \in \mathrm{S}^+$ such that $w \neq w'$, $\Pi_w \cap \Pi_{w'} = \{\ \}$. This does not represent a real restriction; however, its relaxation would make the presentation more complex.

For each relation name $\pi \in \Pi_{s_1\ldots s_n}$, the set $F_{\mathrm{X}}(\Omega)_\pi$ of *well formed atomic formulae* with relation name π, or π-formulae, is defined by: $F_{\mathrm{X}}(\Omega)_\pi = \{(t_1, \ldots, t_n) : t_i \in T_{\mathrm{X}}(\Sigma)_{s_i}\}$. 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 $F_{\mathrm{X}}(\Omega)_\pi$, for some $\pi \in \Pi$; in which case we write $\pi(t_1, \ldots, t_n) \in F_{\mathrm{X}}(\Omega)$ omitting the index π from $F_{\mathrm{X}}(\Omega)_\pi$.

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, where for $w = s_1 \ldots s_n$, A^w denotes the cartesian product $\mathrm{A}_{s_1} \times \ldots \times \mathrm{A}_{s_n}$, and $[\mathrm{A}^w \to \mathrm{A}_s]$ denotes the function space from A^w into A_s. The class of all Σ-algebras is denoted by $Alg(\Sigma)$. 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.

Finally, 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. Let $\Omega \sqsubseteq \Omega'$, and $\mathrm{M} \in Mod(\Omega')$, 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 $\Sigma = (\mathrm{S}, \mathrm{F})$, $\Sigma' = (\mathrm{S}', \mathrm{F}')$, and $\mathrm{A} \in Alg(\Sigma')$: $\Sigma \sqsubseteq \Sigma'$ denotes the algebraic signature inclusion, and A/Σ denotes the reduct of A by Σ. Furthermore, if $\mathrm{N} \in Mod(\Omega)$ then N/Σ denotes the Σ-algebra part of the model N.

3 The Formalism of Relational Semantics

In this section we define the semantic formalism that will be used in the rest of this work to define semantic aspects of programming languages. This formalism, called *Relational Semantics*, is based on Plotkin's Structural Operational Semantics [Plo81]. Our interpretation of Plotkin's ideas is inspired by the work on Natural Semantics [Kah88] and Inductive Semantics [Ast89, Ast91]. In Section 3.1 we define the syntactical aspects of Relational Semantics, and in Section 3.2 we characterise the meta-semantics of this formalism.

3.1 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 more readable.

The basic syntactical objects of a Relational Specification are sorts, terms, formulae, and inductive definitions. Sorts, terms, and formulae were defined in Section 2 and we now defined the concept of a Relational Inductive Definition. A *Relational Inductive Rule* (or simply a *Relational Rule*) is a pair $(\{\mathrm{Pre}_\pi\}_{\pi\in\Pi}, \mathrm{ce}_{\pi'})$ where $\mathrm{Pre}_\pi \in \mathcal{F}(F_{\mathrm{X}}(\Omega)_\pi)$ is a finite set of π-formulae (called the *premisses*) and $\mathrm{ce}_{\pi'} \in F_{\mathrm{X}}(\Omega)_{\pi'}$, for some $\pi' \in \Pi$, is a π'-formula (called the *conclusion*).

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 [Acz77]). 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 3.2, 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 3.1 (Relational Specification Syntax) A *Relational Specification Syntax* is a pair (Ω, ϕ), where ϕ is a Relational Inductive Definition over the signature Ω. $\square$

This definition provides 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 LSL [Ber91], 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 work 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 $\mathrm{P} \subseteq \mathrm{S} \times (\mathrm{S}+\mathrm{F})^*$. Then a suitable transformation on the right hand side of the productions of G defines an $\mathrm{S}^* \times \mathrm{S}$-sorted set Σ of function constants. The initial Σ-algebra $T(\Sigma)$ has a carrier for every sort $s \in \mathrm{S}$ 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 a simple expression language:

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

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

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. We then write a Relational Rule (Pre, ce), where Pre is the set $\{f_1, \ldots, f_n\}$, in the traditional Natural Deduction style, as follows:

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

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 $s \in \mathrm{S}$, 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 3.1 This example presents a complete definition of a Relational Specification Syntax $Env = (\Omega^{Env}, \phi^{Env})$ for a simple functional language without recursion, called *Fun*. 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 components S^{Env} and F^{Env} of Ω^{Env} are defined by the following BNF rules:

$$\begin{array}{lll}
exp & ::= & \mathbf{id}(var) \mid \mathbf{num}(nat) \mid \mathbf{plus}(exp, exp) \mid \mathbf{let}\ var = exp\ \mathbf{in}\ exp \\
 & & \mathbf{fn}\ var\ .\ exp \mid exp(exp) \\
var & ::= & \mathbf{x} \mid \mathbf{y} \mid \cdots \\
nat & ::= & nat + nat \mid 0 \mid 1 \mid \cdots \\
env & ::= & \varepsilon_E \mid var \mapsto val \cdot env \\
funval & ::= & (env, var, exp) \\
val & ::= & \mathbf{N}(nat) \mid \mathbf{F}(funval)
\end{array}$$

The set X of meta-variables used to construct Σ^{Env}-terms and Ω^{Env}-formulae is as defined follows: X_{exp} is generated by e, X_{var} is generated by id, and X_{nat} is generated by n. We will use the set X in the examples throughout this work and new components will be added to X as required.

Typical Σ^{Env}-terms are $\mathbf{num}(10)$, $\mathbf{let}\ \mathbf{x} = \mathbf{num}(2)\ \mathbf{in}\ \mathbf{plus}(\mathbf{x},\mathbf{x})$. It is often tedious to write unary operators that are only used for sort conversion, e.g., **num**, **id**, and so forth. Therefore, we shall omit these operators in the examples. The context and the convention on the names of the meta-variables will be always sufficient to resolve ambiguities. Using this "syntactic sugar" the grammar above is written as follows:

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

The rules below define the semantics of *Env* by an inductive definition of two relations: $\_ \vdash \_ \Rightarrow \_$ between environments, expressions, and natural numbers, and $\longrightarrow_L$ that describes how to lookup the value of variables in an environment. This semantics is called a *environment-model* semantics of *Env* since the values of program variables are kept in an environment. In [dS92], we show a *substitution-model* semantics for *Env*, and also prove that both semantics are equivalent in a sense to be defined in Section 4.

In the rules below, the relation names in Π^{Env} are declared in the boxes, such that $\boxed{env \vdash exp \Rightarrow nat}$ declares $\_ \vdash \_ \Rightarrow \_ \in \Pi^{Env}_{env \times exp \times nat}$ and emphasises that $\_ \vdash \_ \Rightarrow \_$-formulae are written in infix mode. The above BNF rules defined the *abstract syntax* of *Env*. 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 on an Environment $\boxed{env \vdash exp \Rightarrow val}$

$$\frac{}{E \vdash n \Rightarrow n} \tag{1}$$

$$\frac{(E, id) \longrightarrow_L v}{E \vdash id \Rightarrow v} \tag{2}$$

$$\frac{E \vdash e_1 \Rightarrow n_1 \qquad E \vdash e_2 \Rightarrow n_2}{E \vdash \mathbf{plus}(e_1, e_2) \Rightarrow v_1 + v_2} \tag{3}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad id \mapsto v_1 \cdot E \vdash e_2 \Rightarrow v_2}{E \vdash \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \Rightarrow v_2} \tag{4}$$

$$\frac{}{E \vdash \mathbf{fn}\ id\ .\ e \Rightarrow (E, id, e)} \tag{5}$$

$$\frac{E \vdash e_1 \Rightarrow (E', id, e') \qquad E \vdash e_2 \Rightarrow v_2 \qquad id \mapsto v_2 \cdot E' \vdash e' \Rightarrow v'}{E \vdash e_1(e_2) \Rightarrow v'} \tag{6}$$

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

$$\frac{}{(id \mapsto v \cdot E, id) \longrightarrow_L v} \quad (7)$$

$$\frac{id \neq id' \qquad (E, id) \longrightarrow_L v}{(id' \mapsto v' \cdot E, id) \longrightarrow_L v} \quad (8)$$

Comments

Rule (1) defines that an expression n evaluates to a number n. Rule (2) defines that the value of a program variable v is obtained by searching in the environment E using rules (7) and (8). Rule (3) says 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 $\mathbf{plus}(e_1, e_2)$ evaluates to the result of the operation $n_1 + n_2$, where $+$ is the arithmetic sum on Natural Numbers. The remaining rules are informally interpreted in a similar way. This finishes Example 3.1

□

3.2 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.

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.

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.

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

Definition 3.2 (Relational Specification) A *Relational Specification* is a triple $(\Omega, \phi, \mathrm{A})$, where (Ω, ϕ) is a Relational Specification Syntax, $\Omega = (\mathrm{S}, \mathrm{F}, \Pi)$, $\Sigma = (\mathrm{S}, \mathrm{F})$, and A is a Σ-algebra. □

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

Definition 3.3 (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 $\mathrm{M} \models f$.

In some cases, we may be also interested in satisfaction of formulae with meta-variables over an arbitrary set X. For instance, we showed in [dS92] that the search for a satisfying substitution for such a formula can be interpreted as program evaluation under certain conditions. The problem of program evaluation based on Relational Semantics is not addressed in this presentation. The interested reader is referred to [Des88, Chi89, Ber91] and [dS92, Chapter 5] for different approaches to this problem.

The next example defines an algebra for the Relational Specification Syntax defined in Example 3.1; thus it also defines a Relational Specification for *Env* together with its Declarative Semantics.

Example 3.2 Let $(\Omega^{Env}, \phi^{Env})$ be the Relational Specification Syntax defined in Example 3.1. We now define the Σ^{Env}-algebra A^{Env} and extend *Env* to be the Relational Specification $(\Omega^{Env}, \phi^{Env}, \mathrm{A}^{Env})$. To improve the readability, we use A for A^{Env} in the rest of this example. The carriers of A are defined as follows:

$$\begin{aligned} \mathrm{A}_{exp} &= T(\Sigma^{Env})_{exp} \\ \mathrm{A}_{var} &= T(\Sigma^{Env})_{var} \\ \mathrm{A}_{nat} &= \{0, 1, \ldots\} \end{aligned}$$

The algebra A interprets $+$ as the standard sum operation on the natural numbers and $\neq$ as the intuitive inequality between identifiers, and for each *op* in the set

$$\{\mathbf{id}, \mathbf{num}, \mathbf{plus}, \mathbf{let\_=\_in}, 0, 1, \ldots, x, y, \ldots\}$$

the interpretation op^{A} is defined as a follows. For $op : s_1 \ldots s_n \rightarrow s$ and $t_i \in \mathrm{A}_{s_i}$, $i \in [n_+]$, $op^{\mathrm{A}}(t_1, \ldots, t_n) = op(t_1, \ldots, t_n)$. This is the standard *term algebra* interpretation for the function names [Wec92].

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^{Env} using A. In these examples, we use the "desugared" version of the syntax of *Env* to make the examples clearer. In A the evaluation of a term $\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \in T(\Sigma^{Env})_{exp}$ is defined by:

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

The evaluation of the term $1 + 2 \in T(\Sigma^{Env})_{nat}$ is defined by:

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

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 +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^{Env}, one rule of the derived system will have the form:

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

which makes the two formulae:

$$\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow_S 3$$
$$\mathbf{plus}(\mathbf{num}(1), \mathbf{num}(2)) \Rightarrow_S 1 + 2$$

valid in $(\Omega^{Env}, \phi^{Env}, \mathrm{A})$. This finishes the Example 3.2. □

3.3 Directed Relations and Directed Specifications

We now introduce 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 work we need to distinguish between programs and results so that we can characterise compilers from source language programs into target language programs.

The notion of directed relations appear implicitly in [Ber91] and is defined explicitly for a more restricted form of relational semantics in [Chi89]. Furthermore, a similar concept is introduced in [DM85] for Definite Clause Programs. A comparison between these definitions of directed relations can be found in [dS92].

A *directed first order signature* (or a *directed signature*) is a quadruple $(\mathrm{S}, \mathrm{F}, \Pi, \pi)$ such that (S, F) is an algebraic signature, Π is a $\mathrm{S}^+ \times \mathrm{S}^*$-sorted family of sets, and $\pi \in \Pi$ is a relation name called the *initial relation* of the signature. A directed signature $(\mathrm{S}, \mathrm{F}, \Pi, \pi)$ is a sub-signature of $(\mathrm{S}', \mathrm{F}', \Pi', \pi')$ if $(\mathrm{S}, \mathrm{F}, \Pi) \sqsubseteq (\mathrm{S}', \mathrm{F}', \Pi')$ and $\pi = \pi'$. A relation name $\pi \in \Pi_{w \times w'}$, for some $w \in \mathrm{S}^+$ and $w' \in \mathrm{S}^*$, 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 $t_i \in T_{\mathrm{X}}(\Sigma)_{s_i}$, 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 $F_{\mathrm{X}}(\Omega)_\pi$ 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, $t_w \in T_{\mathrm{X}}(\Sigma)$ is an abbreviation for $t_{s_i} \in T_{\mathrm{X}}(\Sigma)_{s_i}$ for $i \in [n_+]$. The application of a S-sorted function to t_w or $t_{w'}$ is interpreted as the componentwise application. In a formula $\pi(t_w, t_{w'})$, t_w is called the *input*, and $t_{w'}$ is called the *output*.

For example, we can define a directed signature $\Omega = (\mathrm{S}, \mathrm{F}, \Pi, \_ \vdash \_ \Rightarrow \_)$ based on the signature Ω^{Env}, defined in Example 3.1, as follows: $\mathrm{S} = \mathrm{S}^{Env}$, $\mathrm{F} = \mathrm{F}^{Env}$, $\longrightarrow_L$: $env \times exp \leftrightarrow val$, and $\_ \vdash \_ \Rightarrow \_ : env \times exp \leftrightarrow val$. Hereafter, we will use Ω as the new definition of Ω^{Env} in *Env*. This directed signature will be used in Section 6 in the example of a compiler correctness proof.

4 Equivalence between Relational Specifications

In Section 3 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 work.

Whatever the reason we provide different definitions of the same semantic aspects of a programming language, it is necessary for the definitions to be equivalent 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. Furthermore, in the case where one of the semantics is defined in terms of the compilation of programs into some machine code, the equivalence between the semantics give a criterion for compiler correctness.

Towards the definition of an equivalence relation between Relational Specifications, we start in Section 4.1 by defining 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 work proves that two models are observationally equivalent if and only if there exists a Model Correspondence between them. We then use observational equivalence to define an equivalence relation between Relational Specifications in Section 4.2. In Section 5 we show that this equivalence provides a suitable criterion for compiler correctness. In Section 6, we then show how to use Model Correspondence as a proof method for compiler correctness.

4.1 Observational Equivalence between First Order Models

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 according to some Relational Specification. For an arbitrary Relational Specification $(\Omega, \phi, \mathrm{A})$ the set of ground formula $F(\Omega)$ defines all the observations we can make. However, 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 section let $\Omega \in \mathit{Sig}$ with algebraic signature Σ, $\Omega_{\mathrm{OBS}} \sqsubseteq \Omega$ be an observation signature of Ω such that $\Omega_{\mathrm{OBS}} = (\mathrm{S}_{\mathrm{OBS}}, \mathrm{F}_{\mathrm{OBS}}, \Pi_{\mathrm{OBS}})$, and let $\Sigma_{\mathrm{OBS}} \sqsubseteq \Sigma$ denote the algebraic signature $(\mathrm{S}_{\mathrm{OBS}}, \mathrm{F}_{\mathrm{OBS}})$ 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 4.1 (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 $s \in \mathrm{S}_{\mathrm{OBS}}$ 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 $f \in F(\Omega_{\mathrm{OBS}})_\pi$:

   $f \in \mathrm{dom}\ \Psi_\pi^{\mathrm{A}}$ if and only if $f \in \mathrm{dom}\ \Psi_\pi^{\mathrm{B}}$
   and if both sides of the above equivalence are true then
   $\Psi_\pi^{\mathrm{A}}(f) \in \pi^{\mathrm{M}}$ 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 4.1 For any signature Ω and $\Omega_{\mathrm{OBS}} \sqsubseteq \Omega$, $\equiv_{\Omega_{\mathrm{OBS}}} \in \mathit{Mod}(\Omega) \times \mathit{Mod}(\Omega)$ is an equivalence relation on $\mathit{Mod}(\Omega)$, i.e., $\equiv_{\Omega_{\mathrm{OBS}}}$ is transitive, reflexive, and symmetric. □

Fact 4.2 For any signature Ω, and observation signatures $\Omega_{\mathrm{OBS}}, \Omega'_{\mathrm{OBS}} \sqsubseteq \Omega$, if $\Omega_{\mathrm{OBS}} \sqsubseteq \Omega'_{\mathrm{OBS}}$ then $\equiv_{\Omega'_{\mathrm{OBS}}} \subseteq \equiv_{\Omega_{\mathrm{OBS}}}$. $\square$

Among the properties of $\equiv_{\Omega_{\mathrm{OBS}}}$ 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. For instance, 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 following sections. 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 [Hen90] 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 paper.

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 17 that, for some $\Omega \in Sig$ with algebraic signature Σ, $\Sigma_{\mathrm{OBS}} \sqsubseteq \Sigma$ and $\mathrm{A}, \mathrm{B} \in Alg(\Sigma)$.

Definition 4.2 (Strong Correspondence)
A *strong* Σ_{OBS}*-correspondence*[1] between $\mathrm{A}/\Sigma_{\mathrm{OBS}}$ and $\mathrm{B}/\Sigma_{\mathrm{OBS}}$ is an $\mathrm{S_{OBS}}$-sorted relation $\mathrm{G} = \{\mathrm{G}_s\}_{s \in \mathrm{S_{OBS}}}$, where $\mathrm{G}_s \subseteq \mathrm{A}_s \times \mathrm{B}_s$ for all $s \in \mathrm{S_{OBS}}$, such that all $\sigma \in \mathrm{F_{OBS}}$ are *compatible* with G in the following sense:

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

$$\begin{array}{rl} \text{whenever} & (\mathrm{a}_i, \mathrm{b}_i) \in \mathrm{G}_{s_i}, \text{ for } i \in [m_+], \\ \text{then} & (\mathrm{a}_1, \ldots, \mathrm{a}_m) \in \mathrm{dom}\, \sigma^{\mathrm{A}} \text{ if and only if } (\mathrm{b}_1, \ldots, \mathrm{b}_m) \in \mathrm{dom}\, \sigma^{\mathrm{B}} \\ & \text{and if both sides of the above equivalence are true then} \\ & (\sigma^{\mathrm{A}}(\mathrm{a}_1, \ldots, \mathrm{a}_m), \sigma^{\mathrm{B}}(\mathrm{b}_1, \ldots, \mathrm{b}_m)) \in \mathrm{G}_s \end{array}$$

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

[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 paper.

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 4.1.

Proposition 4.1 (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 $t \in T(\Sigma_{OBS})_s$:

$$t \in \text{dom}\ \psi_s^A \text{ if and only if } t \in \text{dom}\ \psi_s^B \tag{9}$$
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 $\sigma : s_1 \times \ldots \times s_n \to s \in \Sigma_{OBS}$ 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,

$$t_i \in \text{dom}\ \psi_{s_i}^A \text{ if and only if } t_i \in \text{dom}\ \psi_{s_i}^B, \tag{10}$$
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 $(\psi_{s_i}^A(t_i), \psi_{s_i}^B(t_i)) \in G_{s_i}$. Since G is a strong Σ_{OBS}-correspondence then:

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

   and when both sides of this equivalence are true then $(\psi_s^A(t), \psi_s^B(t)) \in G_s$.

This finishes the proof of Proposition 4.1. $\square$

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 17 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 4.3 (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 $t_i \in \operatorname{dom} \psi^{\mathrm{A}}_{s_i}$ and $t_i \in \operatorname{dom} \psi^{\mathrm{B}}_{s_i}$ 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:

   $$\begin{array}{rl} \text{whenever} & ((\mathrm{a}_1, \ldots, \mathrm{a}_n), (\mathrm{b}_1, \ldots, \mathrm{b}_n)) \in \mathrm{H}_\pi \\ \text{then} & (a_1, \ldots, a_n) \in \pi^{\mathrm{M}} \text{ if and only if } (b_1, \ldots, b_n) \in \pi^{\mathrm{N}} \end{array}$$

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

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 4.3(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 4.3(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 $T(\Sigma_{\mathrm{OBS}})$, and consequently $F(\Omega_{\mathrm{OBS}})$, 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 6 shows that this method yields practical proofs of equivalence between Relational Specifications.

Theorem 4.1 (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.:

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

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 $(\mathrm{G}, \mathrm{H}) : \mathrm{M} \longleftrightarrow_{\Omega_{\mathrm{OBS}}} \mathrm{N}$ then $\mathrm{M} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{N}$. Since G is a strong Σ_{OBS}-correspondence between $\mathrm{A}/\Sigma_{\mathrm{OBS}}$ and $\mathrm{B}/\Sigma_{\mathrm{OBS}}$ then Proposition 4.1 apllies. Therefore, Definition 4.1(1) is trivially satisfied. It remains to prove that Definition 4.1(2) is satisfied. From Proposition 4.1 it follows that for all $\pi : s_1 \times \ldots s_n \in \Pi_{\mathrm{OBS}}$ and $\pi(t_1, \ldots, t_n) \in F(\Omega_{\mathrm{OBS}})_\pi$,

$$t_i \in \mathrm{dom}\ \psi^{\mathrm{A}}_{s_i} \qquad \text{if and only if} \qquad t_i \in \mathrm{dom}\ \psi^{\mathrm{B}}_{s_i}$$

Therefore,

$$\pi(t_1, \ldots, t_n) \in \mathrm{dom}\ \Psi^{\mathrm{A}}_\pi \text{ if and only if } \pi(t_1, \ldots, t_n) \in \mathrm{dom}\ \Psi^{\mathrm{B}}_\pi$$

Whenever both sides of the above equivalence are true it follows that for all $i \in [n_+]$, $t_i \in \mathrm{dom}\ \psi^{\mathrm{A}}_{s_i}$ and $t_i \in \mathrm{dom}\ \psi^{\mathrm{B}}_{s_i}$. From the fact that H satisfies the requirement of Definition 4.3(2.ii) we obtain that:

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

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

$$(\psi^{\mathrm{A}}_{s_1}(t_1), \ldots, \psi^{\mathrm{A}}_{s_n}(t_n)) \in \pi^{\mathrm{M}} \text{ if and only if } (\psi^{\mathrm{B}}_{s_1}(t_1), \ldots, \psi^{\mathrm{B}}_{s_n}(t_n)) \in \pi^{\mathrm{N}}$$

Therefore, from the definition of formula evaluation (Section 2, page 7):

$$\Psi^{\mathrm{A}}_\pi(\pi(t_1, \ldots, t_n)) \in \pi^{\mathrm{M}} \text{ if and only if } \Psi^{\mathrm{B}}_\pi(\pi(t_1, \ldots, t_n)) \in \pi^{\mathrm{N}}$$

Therefore, Definition 4.1(2) is satisfied and we conclude that $\mathrm{M} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{N}$. 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 $\mathrm{M} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{N}$.

Completeness We need to prove that $\mathrm{M} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{N}$ implies that there exists a pair of relations (G, H), such that $(\mathrm{G}, \mathrm{H}) : \mathrm{M} \longleftrightarrow_{\Omega_{\mathrm{OBS}}} \mathrm{N}$. For this proof we will show first that G defined as $\mathrm{G}_s = (\psi_s^{\mathrm{A}})^{-1} \circ \psi_s^{\mathrm{B}}$, for each $s \in \mathrm{S}_{\mathrm{OBS}}$ is a strong Σ_{OBS}-correspondence between $\mathrm{A}/\Sigma_{\mathrm{OBS}}$ and $\mathrm{B}/\Sigma_{\mathrm{OBS}}$.

Let $(\mathrm{a}_i, \mathrm{b}_i) \in \mathrm{G}_i$ for $n \geq 0$ and all $i \in [n_+]$. Then, choose $t_i \in \mathrm{dom}\ \psi_{s_i}^{\mathrm{A}} \cap \mathrm{dom}\ \psi_{s_i}^{\mathrm{B}}$ in $T(\Sigma_{\mathrm{OBS}})_{s_i}$, such that $\psi_{s_i}^{\mathrm{A}}(t_i) = \mathrm{a}_i$, $\psi_{s_i}^{\mathrm{B}}(t_i) = \mathrm{b}_i$, and $t = \sigma(t_1, \ldots, t_n)$ belongs to $T(\Sigma_{\mathrm{OBS}})_s$ for some $s \in \mathrm{S}_{\mathrm{OBS}}$. From Definition 4.1(1) we have that:

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

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

$$\begin{array}{l} (\sigma^{\mathrm{A}}(\psi_{s_1}^{\mathrm{A}}(t_1), \ldots, \psi_{s_n}^{\mathrm{A}}(t_n)), \sigma^{\mathrm{B}}(\psi_{s_1}^{\mathrm{B}}(t_1), \ldots, \psi_{s_n}^{\mathrm{B}}(t_n))) = \\ (\psi_s^{\mathrm{A}}(t), \psi_s^{\mathrm{B}}(t)) \in (\psi_s^{\mathrm{A}})^{-1} \circ \psi_s^{\mathrm{B}} = \mathrm{G}_s \end{array}$$

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

We will show that H defined as $\mathrm{H}_\pi = (\Psi_\pi^{\mathrm{A}})^{-1} \circ \Psi_\pi^{\mathrm{B}}$, for all $\pi \in \Pi_{\mathrm{OBS}}$, satisfies the requirements of Definition 4.3(2). From this definition of H_π it follows that for all $\pi(t_1, \ldots, t_n) \in F(\Omega_{\mathrm{OBS}})_\pi$, if $t_i \in \mathrm{dom}\ \psi_{s_i}^{\mathrm{A}}$ and $t_i \in \mathrm{dom}\ \psi_{s_i}^{\mathrm{B}}$ for every $i \in [n_+]$, then:

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

The requirements of Definition 4.3(2.i) follows trivially from the definition of H_π and Proposition 4.1. Definition 4.3(2.ii) is trivially satisfies by the definition of H_π. Now, we must show that each $\pi \in \Pi_{\mathrm{OBS}}$ is compatible with H_π. For all $\pi : s_1 \times \ldots \times s_n \in \Pi_{\mathrm{OBS}}$, whenever $((\mathrm{a}_1, \ldots, \mathrm{a}_n), (\mathrm{b}_1, \ldots, \mathrm{b}_n)) \in \mathrm{H}_\pi$, for all $i \in [n_+]$, $s_i \in \mathrm{S}_{\mathrm{OBS}}$ take $t_i \in \mathrm{dom}\ \psi_{s_i}^{\mathrm{A}} \cap \mathrm{dom}\ \psi_{s_i}^{\mathrm{B}}$ in $T(\Sigma_{\mathrm{OBS}})_{s_i}$ such that $\psi_{s_i}^{\mathrm{A}}(t_i) = \mathrm{a}_i$ and $\psi_{s_i}^{\mathrm{B}}(t_i) = \mathrm{b}_i$. It is clear that $\pi(t_1, \ldots, t_n) \in F(\Omega_{\mathrm{OBS}})_\pi$. Now, from the fact that $\mathrm{M} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{N}$, it follows that:

$$\begin{array}{l} \pi(t_1, \ldots, t_n) \in \mathrm{dom}\ \Psi_\pi^{\mathrm{A}} \text{ if and only if } \pi(t_1, \ldots, t_n) \in \mathrm{dom}\ \Psi_\pi^{\mathrm{B}} \\ \text{and if both sides of the above equivalence are true then} \\ \Psi_\pi^{\mathrm{A}}(\pi(t_1, \ldots, t_n)) \in \pi^{\mathrm{M}} \text{ if and only if } \Psi_\pi^{\mathrm{B}}(\pi(t_1, \ldots, t_n)) \in \pi^{\mathrm{N}} \end{array}$$

It is obvious that $\pi(t_1, \ldots, t_n) \in \mathrm{dom}\ \Psi_\pi^{\mathrm{A}}$ and $\pi(t_1, \ldots, t_n) \in \mathrm{dom}\ \Psi_\pi^{\mathrm{B}}$. Therefore, from the definition of evaluation of formulae (Section 2, page 7):

$$(\psi_{s_1}^{\mathrm{A}}(t_1), \ldots, \psi_{s_n}^{\mathrm{A}}(t_n)) \in \pi^{\mathrm{M}} \text{ if and only if } (\psi_{s_1}^{\mathrm{B}}(t_1), \ldots, \psi_{s_n}^{\mathrm{B}}(t_n)) \in \pi^{\mathrm{N}}$$

Finally, we conclude that:

$$(\mathrm{a}_1, \ldots, \mathrm{a}_n) \in \pi^{\mathrm{M}} \text{ if and only if } (\mathrm{b}_1, \ldots, \mathrm{b}_n) \in \pi^{\mathrm{N}}$$

This proves that π is compatible with H_π, for all $\pi \in \Pi_{\mathrm{OBS}}$, and therefore that (G, H) is a Ω_{OBS}-Model Correspondence between M and N. This finishes the proof of Theorem 4.1. $\square$

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 4.1 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 4.3(2.i) allows non observable objects to be related in H. In Section 6, we illustrate the use of Model Correspondences.

4.2 Observational Equivalence between Relational Specifications

In this section we apply the results of Section 4.1 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 $\equiv_{\Omega_{\mathrm{OBS}}}$ 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 3.2) 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 $\mathit{Spec}(\Omega_{\mathrm{OBS}})$. Notice that in general the specifications in $\mathit{Spec}(\Omega_{\mathrm{OBS}})$ have different signatures. The notion of Observational Equivalence between Relational Specifications will account for this.

For instance, let $\Omega^{\mathit{Env}}_{\mathrm{OBS}}$ be the observation signature for the Relational Specification *Env* defined as follows. The set $\Pi^{\mathit{Env}}_{\mathrm{OBS}}$ has only the relation $\_ \vdash \_ \Rightarrow \_$, and the sets $\mathrm{S}^{\mathit{Env}}_{\mathrm{OBS}}$ and $\mathrm{F}^{\mathit{Env}}_{\mathrm{OBS}}$ are defined by the following BNF rules:

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

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. We now extend *Env* with $\Omega_{\mathrm{OBS}}^{Env}$ such that the new Relational Specification is written $(\Omega^{Env}, \phi^{Env}, \mathrm{A}^{Env}, \Omega_{\mathrm{OBS}}^{Env})$. This new definition of *Env* will be use in the rest of this presentation.

We can now define a relation of Observational Equivalence on $Spec(\Omega_{\mathrm{OBS}})$, for an arbitrary observation signature Ω_{OBS}.

Definition 4.4 (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:

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

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 4.3 The relation $\cong \subseteq Spec(\Omega_{\mathrm{OBS}}) \times Spec(\Omega_{\mathrm{OBS}})$ is an equivalence relation on the class $Spec(\Omega_{\mathrm{OBS}})$, for any observation signature Ω_{OBS}. $\square$

5 Evaluation by Compilation

In this section we study compiler correctness in the context of the theory of Observational Equivalence developed in Section 4. The generality of the framework developed in Section 4 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 2) 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

$$
\begin{array}{ccc}
 & & \underline{L} \\
 & & \gamma \downarrow \ \text{compiler} \\
\underline{L} & & \underline{T} \\
\text{programming language}\ \updownarrow \pi & \equiv_{\Omega_{\mathrm{OBS}}} & \mu \updownarrow \ \text{machine language} \\
\underline{M} & & \underline{U} \\
 & & \delta \updownarrow \ \text{unloading} \\
 & & \underline{M'}
\end{array}
$$

Figure 2: Evaluation by Compilation

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 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* $\gamma : \underline{L} \rightarrow \underline{T}$, 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 section 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, our starting point is a Relational Specification *Env* of Example 3.1 extended with the observation signature Ω_{OBS} in Section 4.2. Hereafter, let M^{Env} denote the Declarative Semantics of *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.

Example 5.1 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}:

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

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 with the following components: X_{state} generated by *st*, X_{stack} generated by *S*, X_{code} generated by *c*, X_{inst} generated by *op*.

We now present the rules that define the directed relation $\stackrel{\mathrm{CAM}}{\longrightarrow}$: $\mathit{state} \leftrightarrow \mathit{state}$. This relation defines the evaluation of a single CAM instruction. The transitive-reflexive closure of $\stackrel{\mathrm{CAM}}{\longrightarrow}$, written $\stackrel{\mathrm{CAM}^*}{\longrightarrow}$, defines the evaluation of arbitrary sequences of instructions. The definition of $\stackrel{\mathrm{CAM}^*}{\longrightarrow}$ by a set of Relational Rules is straightforward and is omitted in this presentation.

The Transition Relation $\boxed{\mathit{state} \stackrel{\mathrm{CAM}}{\longrightarrow} \mathit{state}}$

$$\frac{}{(v \cdot S, \mathbf{quote}(n) \cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} (n \cdot S, c)} \tag{1}$$

$$\frac{}{(v \cdot S, \mathbf{push} \cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} (v \cdot v \cdot S, c)} \tag{2}$$

$$\frac{}{((v_1, v_2) \cdot S, \mathbf{car} \cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} (v_1 \cdot S, c)} \tag{3}$$

$$\frac{}{((v_1, v_2) \cdot S, \mathbf{cdr} \cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} (v_2 \cdot S, c)} \tag{4}$$

$$\frac{}{(v_1 \cdot v_2 \cdot S, \mathbf{cons} \cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} ((v_2, v_1) \cdot S, c)} \tag{5}$$

$$\frac{}{(v_1 \cdot v_2 \cdot S, \mathbf{swap} \cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} (v_2 \cdot v_1 \cdot S, c)} \tag{6}$$

$$\frac{}{(v \cdot S, \mathbf{cur}(c') \cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} ([v, c'] \cdot S, c)} \tag{7}$$

$$\frac{}{(([v,c],v')\cdot S, \mathbf{app}\cdot c') \stackrel{\mathrm{CAM}}{\longrightarrow} ((v,v')\cdot S, c@c')} \tag{8}$$

$$\frac{}{(v_1\cdot v_2\cdot S, \mathbf{add}\cdot c) \stackrel{\mathrm{CAM}}{\longrightarrow} (v_2+v_1\cdot S, c)} \tag{9}$$

The directed signature Ω^{Cam} is defined by $(\mathrm{S}^{Cam}, \mathrm{F}^{Cam}, \Pi^{Cam}, \stackrel{\mathrm{CAM}}{\longrightarrow})$, where $\stackrel{\mathrm{CAM}}{\longrightarrow}$: *state* $\times$ *state*. For all $s \in \mathrm{S}^{Cam}$, $\mathrm{A}^{Cam}_s = T(\Sigma^{Cam})_s$, except that $\mathrm{A}^{Cam}_{nat} = \{0,1,\ldots\}$. Moreover, the Σ^{Cam}-algebra A^{Cam} interprets $+$ as the standard sum operation on the natural numbers and all other function names are given the term algebra interpretation. The observational signature $\Omega^{Cam}_{\mathrm{OBS}}$ of *Cam* is just the signature Ω^{Cam}. Therefore, every object in this Relational Specification is observable. This finishes Example 5.1. $\square$

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

Example 5.2 In this example we define a compiler for *Fun* into CAM code by presenting a Relational Specification $Trans = (\Omega^{Trans}, \phi^{Trans}, \mathrm{A}^{Trans}, \Omega^{Trans}_{\mathrm{OBS}})$. 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 3.1. 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 with the component X_{comp_env} generated by *CE*. The rules below define the compiler $\stackrel{Comp}{\longrightarrow}$: *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 $\_ \vdash \_ \stackrel{Comp}{\longrightarrow} \_$: $env \times exp \leftrightarrow code$ generates code for expressions.

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

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

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

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

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 $\boxed{comp\_env \vdash exp \stackrel{Comp}{\longrightarrow} code}$

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

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

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

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

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

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

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

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

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 5.2. □

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 of *Fun* expressions.

Example 5.3 This example defines the remaining components of a Compilation for *Fun* programs by presenting a Relational Specification $Comp = (\Omega^{Comp}, \phi^{Comp}, \mathrm{A}^{Comp}, \Omega^{Env}_{\mathrm{OBS}})$. The BNF rules which define the sets S^{Comp} and F^{Comp} of Ω^{Comp} are the same as in Example 5.2. 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 5.1 and 5.2.

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

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

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

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

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 $\stackrel{\mathrm{CAM}}{\longrightarrow}$, and unloading the result from the final CAM state. The Σ-algebra A^{Comp} agrees with the algebras defined in Examples 5.1 and 5.2 on all function names. The observation signature $\Omega^{Env}_{\mathrm{OBS}}$ is the same as for the Relational Specification *Env* defined above. Hereafter, let M^{Comp} denote the Declarative Semantics of *Comp*. This finishes Example 5.3. □

In Examples 5.1, 5.2, and 5.3 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. The following section shows a proof of compiler correctness using Model Correspondence.

6 A Proof of Compiler Correctness

In Section 5 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 Section 4, 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 $\stackrel{Comp}{\longrightarrow}$ defined in Example 5.2 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.

To simplify the presentation, we adopt the following convention: whenever $E \in \mathrm{A}^{Env}_{env}$, $e \in \mathrm{A}^{Env}_{exp}$, and $v \in \mathrm{A}^{Env}_{val}$ 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 $\stackrel{Comp}{\longrightarrow}$ defines a function. This is stated in the following proposition.

Proposition 6.1 For all $E \in \mathrm{A}^{Comp}_{env}$, $e \in \mathrm{A}^{Comp}_{exp}$, and $st, st' \in \mathrm{A}^{Comp}_{state}$, if $(E, e) \stackrel{Comp}{\longrightarrow} st$ and $(E, e) \stackrel{Comp}{\longrightarrow} st'$ then $st = st'$.

Proof (Sketch) The definition of $\stackrel{Comp}{\longrightarrow}$ in rule 5.2(12) is given in terms of the relations $\longrightarrow_E$ and $\_ \vdash \_ \stackrel{Comp}{\longrightarrow} \_$. This proof follows easily by induction on the height of the proof trees for the relation $\_ \vdash \_ \stackrel{Comp}{\longrightarrow} \_$. 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 6.2 The Relational Specifications *Env* and *Comp* are observationally equivalent, i.e., $Env \cong Comp$. □

Hereafter, we drop the superscript *Env* from the signature $\Omega^{Env}_{\mathrm{OBS}}$ whenever it does not introduce ambiguities. For the proof of Proposition 6.2 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 6.1 (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 $\mathrm{AE}_s \times \mathrm{AC}_s$.

$\mathrm{G}_{funval} = \{((\mathrm{E}, id, e), [S, c]) : E \longrightarrow_E (CE, \mathrm{S}) \text{ and } (CE, id) \vdash e \stackrel{Comp}{\longrightarrow} c\}$

$\mathrm{G}_{val} = \mathrm{G}_{funval} + \mathrm{G}_{nat}$ □

Definition 6.2 (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 6.3, and the proof of Proposition 6.2 then follows from Theorem 4.1.

Proposition 6.3 The pair (G, H) of Definitions 6.1 and 6.2 is an Ω_{OBS}-Model Correspondence between ME and MC, i.e., $(\mathrm{G}, \mathrm{H}) : \mathrm{ME} \longleftrightarrow_{\Omega_{\mathrm{OBS}}} \mathrm{MC}$. □

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 6.2 and carried out in the same way. Therefore, we omit the proof of Lemma 6.1 from this presentation.

Lemma 6.1 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 $E \longrightarrow_E (CE, v'')$, $CE \vdash id \overset{Comp}{\longrightarrow} c$,
   $(v'' \cdot S', c@c') \overset{\mathrm{CAM}^*}{\longrightarrow} (v' \cdot S', c')$ and $(v, v') \in \mathrm{G}_{val}$
2. and conversely $\square$

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 6.2 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 6.2 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 $E \longrightarrow_E (CE, v'')$, $CE \vdash e \overset{Comp}{\longrightarrow} c$,
   $(v'' \cdot S', c@c') \overset{\mathrm{CAM}^*}{\longrightarrow} (v' \cdot S', c')$, and $(v, v') \in \mathrm{G}_{val}$
2. And conversely

Proof We prove 6.2(1) and 6.2(2) separately.

6.2(1) This proof is by induction on the height of the proof tree for $E \vdash e \Rightarrow v$ in *Env*.

Rule 3.1(1) 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 3.1(2) The proof of this case follows immediately from Lemma 6.1.

Rule 3.1(3) The proof of this case is simpler than for the other cases and we omit it here.

Rule 3.1(4) The proof of this case is similar to the case of rule 3.1(6). It is therefore omited.

Rule 3.1(5) In this case $e = \textbf{fn}\ id\ .\ e'$, and the proof tree for this expression is as follows:

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

In *Comp* the corresponding proof trees are as follows:

$$\frac{}{\mathrm{E} \longrightarrow_E (\mathrm{CE}, v'')} \qquad \frac{\dfrac{PT}{(\mathrm{CE}, id) \vdash e' \xrightarrow{Comp} c}}{\mathrm{CE} \vdash \textbf{fn}\ id\ .\ e' \xrightarrow{Comp} \langle \textbf{cur}(c) \rangle}$$

and we have the following single CAM transition:

$$(v'' \cdot S', \langle \textbf{cur}(c) \rangle @ c') \xrightarrow{\mathrm{CAM}} ([v'', c] \cdot \mathrm{S}', c')$$

It follows from the definition of G_{funval} that $((E, id, e'), [v'', c]) \in \mathrm{G}_{val}$.

Rule 3.1(6) 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:

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

Therefore, in *Comp* we must have the following proof trees:

$$\frac{}{\mathrm{E} \longrightarrow_E (\mathrm{CE}, v'')} \qquad \frac{\dfrac{PT_1'}{\mathrm{CE} \vdash e_1 \xrightarrow{Comp} c_1} \qquad \dfrac{PT_2'}{\mathrm{CE} \vdash e_2 \xrightarrow{Comp} c_2}}{\mathrm{CE} \vdash e_1(e_2) \xrightarrow{Comp} \langle \textbf{push} \rangle @ c_1 @ \langle \textbf{swap} \rangle @ c_2 @ \langle \textbf{cons,app} \rangle}$$

and the following CAM transition sequence:

$$
\begin{aligned}
&(v'' \cdot S', \langle \textbf{push} \rangle @ c_1 @ \langle \textbf{swap} \rangle @ c_2 @ \langle \textbf{cons,app} \rangle @ c') \xrightarrow{\mathrm{CAM}} \\
&(v'' \cdot v'' \cdot S', c_1 @ \langle \textbf{swap} \rangle @ c_2 @ \langle \textbf{cons,app} \rangle @ c') \xrightarrow{\mathrm{CAM}^*} && (1)\\
&(v_1' \cdot v'' \cdot S', \langle \textbf{swap} \rangle @ c_2 @ \langle \textbf{cons,app} \rangle @ c') \xrightarrow{\mathrm{CAM}} \\
&(v'' \cdot v_1' \cdot S', c_2 @ \langle \textbf{cons,app} \rangle @ c') \xrightarrow{\mathrm{CAM}^*} && (2)\\
&(v_2' \cdot v_1' \cdot S', \langle \textbf{cons,app} @ c') \xrightarrow{\mathrm{CAM}} \\
&(([v''', c''], v_2') \cdot S', \langle \textbf{app} \rangle @ c') \xrightarrow{\mathrm{CAM}} \\
&((v''', v_2') \cdot S', c'' @ c') \xrightarrow{\mathrm{CAM}^*} && (3)\\
&(v' \cdot S', c')
\end{aligned}
$$

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 $(CE', id) \vdash e' \overset{Comp}{\longrightarrow} c''$. 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 6.2(1).

6.2(2) (Sketch): This proof is analogous to the proof of part 6.2(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 6.2. $\square$

Proof (of Proposition 6.3) We must prove that (G, H) obeys the requirements of Definitions 4.3(1) and 4.3(2). Since AE and AC give the same interpretation for each symbol in Σ_{OBS} the proof of Definition 4.3(1) is trivial. Therefore, we can assume that $\mathrm{G} : \mathrm{AE} \Longrightarrow\!\!\times_{\Sigma_{\mathrm{OBS}}} \mathrm{AC}$. It remains to prove Definition 4.3(2). This proof becomes clearer if we first use Rule 5.3(2 to rewrite Lemma 6.2 as follows: for all $(\mathrm{E}, \mathrm{E}) \in \mathrm{G}_{env}$, $(e, e) \in \mathrm{G}_{exp}$:

1. If there exists $v \in \mathrm{AE}_{val}$ such that $(E, e, v) \in \_ \vdash \_ \Rightarrow \_^{\mathrm{ME}}$
   then there exists $v' \in \mathrm{AC}_{val}$, such that $(E, e, v) \in \_ \vdash \_ \Rightarrow \_^{\mathrm{MC}}$
   and $(v, v') \in \mathrm{G}_{val}$
2. and conversely.

First, let us prove Definition 4.3(2.i) is satisfied. Any pair $((E, e, v), (E', e', v'))$ can only be in $\mathrm{H}_{\_\vdash\_\Rightarrow\_}$ because of either Definition 6.2(1) or Definition 6.2(2). Suppose it is because of Definition 6.2(1), then from Proposition 4.1 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 6.2(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 6.2 that $(v, v') \in \mathrm{G}_{val}$. We then conclude that Definition 4.3(2.i) is satisfied

Definition 4.3(2.ii) follows trivially from Definition 6.2(1). It remains to show that the relation $\_ \vdash \_ \Rightarrow \_$ is compatible with $\mathrm{H}_{\_\vdash\_\Rightarrow\_}$ in the sense of Definition 4.3(2). i.e., we must prove the following statement:

whenever $((E, e, v), (E, e, v')) \in \mathrm{H}_{\_\vdash\_\Rightarrow\_}$ then:
$(E, e, v) \in \_ \vdash \_ \Rightarrow \_^{\mathrm{ME}}$ if and only if $(E, e, v') \in \_ \vdash \_ \Rightarrow \_^{\mathrm{MC}}$

Let us prove the left to right implication first. Suppose that $(E, e, v) \in \_ \vdash \_ \Rightarrow \_^{\mathrm{ME}}$ and let us analyse the possible values for v.

1. If $v = n$ since $(v, v') \in \mathrm{G}_{val}$ then $v = v'$. In this case it follows from Lemma 6.2 that there exists v'' such that $(E, e, v'') \in \_ \vdash \_ \Rightarrow \_^{\mathrm{MC}}$ and $(v, v'') \in \mathrm{G}_{val}$. Since $v = n$ then $v = v''$, therefore $v' = v''$.

2. If $v = (E', x, e')$, then $((E, e, v), (E, e, v'))$ is in $\mathrm{H}_{\_\vdash\_\Rightarrow\_}$ because of Definition 6.2(2) since there is no visible term of sort *funval*. In this case, $((E, e, v), (E, e, v')) \in \mathrm{H}_{\_\vdash\_\Rightarrow\_}$ only if $(E, e, v) \in \_\vdash\_\Rightarrow\_^{\mathrm{ME}}$ and $(E, e, v') \in \_\vdash\_\Rightarrow\_^{\mathrm{MC}}$, and the statement holds.

The proof of the right to left implication follows similarly. Therefore, we have proved that (G, H) satisfies Definitions 4.3(1) and 4.3(2), concluding that $(\mathrm{G}, \mathrm{H}) : \mathrm{ME} \longleftrightarrow_{\Omega^{SFun}_{\mathrm{OBS}}} \mathrm{MC}$. From Theorem 4.1 it follows that $\mathrm{ME} \equiv_{\Omega_{\mathrm{OBS}}} \mathrm{MC}$ and from Definition 4.4 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 $\mathrm{A}^{Comp}_{funval}$ to $\mathrm{A}^{Env}_{funval}$ 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].

7 Concluding Remarks

In this work we introduce a formalisation of Relational Semantics using concepts from Universal Algebra and Inductive Definitions. We then defined an equivalence relation between Relational Specifications using observational equivalence [ST87], and a proof method for proofs of equivalence based on correspondence [Sch87, Sch90]. The main theoretical result of this work proves that the proof method of Model Correspondence is sound and complete with respect to Observational Equivalence. We then applied these results to the problem of compiler correctness.

We started by 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. We then defined a criterion for compiler correctness based on Observational Equivalence, as defined in Section 4. This criterion is more general than the previous approaches to the compiler correctness problem. In this sense, we argue that the results of this paper 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 2). 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 1.

It is now necessary to try the results presented here to more and larger examples. As the size and complexity of examples increase, machine assistance will be required to help with clerical work on proofs of compiler correctness and also to avoid the possibility of human error in such proofs. A typical machine assistance tool would consist of a parser and type-checker for Relational Specifications, as in the TYPOL/CENTAUR system [Des88, BCD$^+$87], integrated with a semi-automatic theorem prover in which compiler correctness proofs can be carried out. We believe that the formalisation of Relational Semantics presented in Section 3 and the definitions of Observational Equivalence and Model Correspondence are important steps towards the definition of machine assisted tools for compiler correctness.

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.

Acknowledgements

I would like to thank Kevin Mitchell for helpful suggestions on the research that led to the results presented in this paper. Don Sannella introduced me to the work of Oliver Schoett on correspondence, and also contributed greatly with suggestions on the material developed in Sections 4 and 5. Thanks also to Ed Kazmierczak, Kees Goossens, and Matthew Morley for proof-reading previous versions of this text. During the four years of my Ph.D. research at Edinburgh, I was supported by a Brazilian government scholarship, CNPq process number 20.0459/88.0/CC.

References

[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.

[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.

[BCD+87] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. Technical Report 777, INRIA, Rocquencourt, France, December 1987.

[Ber91] 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.

[BF90] Clement A. Baker-Finch. The (algebraic) congruence of two programming language semantics. *Science of Computer Programming*, 14(1):81–96, June 1990.

[BL69] R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.

[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.

[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.

[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.

[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. Research Report 514, INRIA, Sophia-Antipolis, France, April 1986.

[Des88] T. Despeyroux. TYPOL: a formalism to implement Natural Semantics. Technical Report 94, INRIA, Sophia-Antipolis, France, March 1988.

[DM85] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2:119–155, 1985.

[dS92] Fabio Q. B. da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics.* Thesis submitted for the degree of Ph.D., LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, 1992.

[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.

[Grä79] G. Grätzer. *Universal Algebra.* Springer-Verlag, second edition, 1979.

[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.

[Hen90] 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.

[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.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[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.

[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.

[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.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Mor73] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, October 1973.

[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.

[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.

[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.

[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.

[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.

[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.

[Sto81] Joseph E. Stoy. The congruence of two programming language semantics. *Theoretical Computer Science*, 13:151–174, 1981.

[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.

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.