Structuring Compilers Using Relational Semantics

Fabio Q. B. da Silva
Departamento de Informática, Universidade Federal de Pernambuco,
Av. Prof. Luiz Freire, s/n, CP 7851, CEP 50732-970,
Recife, Pernambuco, Brasil. e-mail: fabio@di.ufpe.br.

Abstract

We study the problem of establishing the correctness of compilers in the framework of observational equivalence [ST87]. The starting point for correctness is a formal semantics of the programming language given in a form of Structural Operational Semantics [Plo81] which we call Relational Semantics [dS92, Chapter 2]. We start by introducing Relational Semantics. We then define an equivalence relation between Relational Semantics based on the notion of observational equivalence [ST87]. 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.

1 Introduction

We provide an overview of the main problems involved in establishing the correctness of compilers based on formal semantics. We discuss how to define a compiler using Relational Semantics and how to establish a criterion for its correctness based on the algebraic concept of an observational equivalence. We do not address the problem of providing proofs of compiler correctness, which is treated elsewhere [dS92, dS94]. We omit some formal definitions and proofs and concentrate on an intuitive presentation. The interested reader is referred to [dS92, dS94] for a detailed account of the ideas addressed in this paper.

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. The starting point to study this problem is a semantic formalism in which the programming language is defined. We use the formalism of Relational Semantics [dS92].

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.

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 $\pi$ arrow denotes the semantics of the programming language, while the $\mu$ arrow denotes the semantics of the machine language. In the Morris Diagram, the algebra $T(\Sigma)$ for an algebraic signature $\Sigma$ which defines the programming language. Therefore, $\gamma$ and $\pi$ are unique homomorphisms by initiality. The proof method is also based on initiality, for if $\mu$ and $\delta$ 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 $\delta$ 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 $\text{fn } x . x + 1$ could be represented as $(\varepsilon_E, x, x + 1)$, and the value of $\text{fn } y . y + 1$ could be represented as $(\varepsilon_E, y, y + 1)$. It is conceivable that in a machine implementation these two expressions evaluate to the same machine value that "represents" all $\alpha$-conversions of a function value like $(\varepsilon_E, x, x + 1)$. In this case there is no homomorphism $\delta$ 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 $\epsilon : M \rightarrow U$ (of encoding) to replace the homomorphism $\delta : U \rightarrow M$ in the Morris Diagram. A motivation for using $\epsilon$ 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 $\epsilon : M \rightarrow U$ is not a sufficient criterion for correctness. For instance, in the case where $T$ and $U$ are one-point algebras and $\gamma, \mu$, and $\epsilon$ are the unique homomorphism to these algebras, the diagram commutes trivially, as mentioned in [TWW81]. A second reason why $\epsilon$ does not give a sufficient correctness criterion is illustrated by a simple example. Suppose $\gamma$ compiles every program $l$ in $L$ into a (fixed) code sequence $t$ in $T$. Therefore, since $\mu$ is a function, every program has the (fixed) meaning $\mu(l)$ in $U$. Furthermore, for every $l$ in $L$ suppose that $\epsilon$ maps $\pi(l)$ into (the fixed) $\mu(l)$. 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 $T$ never to be one-point. However, the second problem discussed above suggests that errors in the compiler arrow $\gamma$ can be hidden by a suitable choice of the encoding arrow $\epsilon$. 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 produced 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 $M$. However, the existence of an encoding $\epsilon$ 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 $M$ that corresponds to the result in $U$ obtained from the execution of the program's code. We argue this is not sufficient from a pragmatic point of view.

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 $\epsilon$ 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 $M$ and $U$. Another solution would be to compare the algebras $M$ and $U$ under observational equivalence [ST87, 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. We propose to use observational equivalence as the criterion for compiler correctness.

The rest of this paper is organised as follows. Section 2 presents the formalism of Relational Semantics. In

![Figure 1: The Morris Diagram](image-url)
Section 3, we treat the specification of compilers in the framework of Relational Semantics. In Section 4, we study compiler correctness in the framework of Relational Semantics and observational equivalence. Finally, in Section 5 we present some conclusions and directions for future work.

2 Relational Semantics

We use Relational Semantics [dS92] to define the semantics of programming languages. This formalism is a variant of Plotkin’s Structural Operational Semantics [Plo81], and is closely related to Kahn’s Natural Semantics [Kah88]. The underlying mathematics of Relational Semantics is simple, which, in general, makes Relational Specifications easier to understand and reason about than, say, a denotational semantics description. In this section, we introduce Relational Semantics through an example and give essential definitions for later sections. We assume familiarity with the basic concepts of Universal Algebras as in [Wec92].

To illustrate Relational Semantics let us define a semantics of a simple functional language with local variables, which we call Fun. The first stage is to define the language’s syntax. The abstract syntax of this language will be defined by a term algebra using 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 T disjoint from S, and production rules $P \subseteq S \times (S + T)^*$. Then a suitable transformation on the right hand side of the productions of G defines an $S^* \times S$-sorted set F of function constants.

Let $\Sigma$ stand for $(S,F)$. The initial $\Sigma$-algebra $T(\Sigma)$ has a carrier for every sort $s \in S$ which is the set of parse trees derived from a non-terminal $s \in P$. It is important to notice that the CFG defines the abstract syntax of the language rather than its concrete syntax. It is in this sense that this approach to define a $\Sigma$-algebra factors out parsing problems so that the CFG may be ambiguous yet the algebra is well defined.

The abstract syntax of Fun is defined by the following grammar:

$$
\begin{align*}
\text{exp} & ::= \text{id}(\text{var}) \mid \text{num}(\text{nat}) \mid \text{plus}(\text{exp}, \text{exp}) \mid \text{let}\ \text{var} = \text{exp}\ \text{in}\ \text{exp} \\
\text{fun}\ \text{var},\ \text{exp} \mid \text{exp}(	ext{exp}) \\
\text{env} & ::= \varepsilon_E \mid (\text{var}, \text{val}) :: \text{env} \\
\text{nat} & ::= \text{nat} + \text{nat} \mid 0 \mid 1 \mid \ldots \\
\text{var} & ::= x \mid y \mid \ldots \\
\text{funval} & ::= (\text{env}, \text{var}, \text{exp}) \\
\text{val} & ::= \text{N}(\text{nat}) \mid F(\text{funval})
\end{align*}
$$

The non terminals of the above grammar define the set of sorts $S = \{\text{exp}, \text{env}, \text{nat}, \text{var}, \text{funval}, \text{val}\}$, and the right hand side of the productions define the operation constants $0 :\rightarrow \text{nat}, 1 :\rightarrow \text{nat}, \text{num} : \text{nat} \rightarrow \text{exp}, \text{plus} : \text{exp} \times \text{exp} \rightarrow \text{exp}$, and so forth. Therefore, the above grammar defines an algebraic signature $(S,F)$, where $F$ is the $S^* \times S$-sorted family of sets of operation constants defined by the grammar productions. An algebraic signature is denoted by $\Sigma$, and we use $\Sigma^{\text{Fun}}$ to refer to the algebraic signature of the language Fun whenever it is necessary.

Every algebraic signature $\Sigma$ defines a set of well formed terms that can be constructed from free meta-variables and the function names in F (called the $\Sigma$-terms). Let X be an S-sorted family of meta-variables, $T_X(\Sigma)$, denotes the terms of sort $s$ with meta-variables in X, defined in the usual way [Wec92]. Typical $\Sigma$-terms for the algebraic signature defined above are $\text{num}(10)$, let $x = \text{num}(2)$ in $\text{plus}(x,x)$.

In the following examples, we use $e, e_1, e_2$ in $\text{Xexp}, E$ and $E'$ in $\text{Xenv}, n, n_1, n_2$ in $\text{Xnat}, v, v_1, v_2$ in $\text{Xval}$, and $id, id'$ in $\text{Xvar}$. It is convenient to emphasise that $\text{plus}$ is the language constructor and “+” is the mathematical sum operation. The constructors $\text{num}$ and $\text{id}$ are used to coerce $\text{nat}$ and $\text{var}$ to $\text{exp}$ respectively. Similarly, N and F are used to coerce $\text{nat}$ and $\text{funval}$ to $\text{val}$. It is often tedious to write unary operators that are only used for sort conversion, e.g., $\text{num}$, $\text{id}$, 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.

Expressions in Fun may contain variables; thus the value of $\text{plus}(x, 1)$ clearly depends on the context in which the expression is evaluated. Therefore, expressions are evaluated with respect to a context, called an environment. We represent environments as a list of pairs $(id, v)$, where $id$ is a variable and $v$ is the value of the variable in the environment. The operatorion “::” represents the “cons” function.

Once we have defined the syntactical aspects of the language, the next stage is to define a Relational Semantics of Fun using environments. A Relational Semantics is given by the set of inference rules, which we call Relational
rules. In Figure 2, we have a set of rules that defines a “call by value” semantics of Fun. Rules (1) to (6) define the evaluation of Fun expression in an arbitrary environment. Rules (7) and (8) define how to search an environment for the value of variables. In those Relational rules, \( E \vdash e \Rightarrow n \) is a (first order) formula which asserts that in an environment \( E \) the expression \( e \) has value \( n \). In this case, \(-\vdash-)\) is a relation name of sort \( env \times exp \times val \), and the formulae used in the rules are (freely) generated using this relation name and terms in \( T_X(\Sigma^{Fun}) \). Similarly, \( (E, id) \rightarrow_L v \) asserts that \( id \) has value \( v \) in \( E \).

In general, a (many sorted) first order signature (or simply a signature) is a triple \( (S, F, \Pi) \), where \( (S, F) \) is an algebraic signature and \( \Pi \) is a \( S^+ \)-sorted family of sets (of relation names); we denote a first order signature by \( \Omega \). The class of all first order signatures is denoted by \( \text{Sig} \). If \( \Omega, \Omega' \in \text{Sig} \) then \( \Omega \subseteq \Omega' \) states that \( \Omega \) is a sub-signature of \( \Omega' \). Every first order signature \( \Omega = (S, F, \Pi) \) defines a set of well formed atomic formulae that can be constructed from free meta-variables and \( \Sigma \)-terms. For each relation name \( \pi \in \Pi_{n_1 \ldots n_s} \), the set \( F_X(\Omega)_\pi \) of well formed atomic formulae with relation name \( \pi \), or \( \pi \)-formulae, is defined by: \( F_X(\Omega)_\pi = \{ (t_1, \ldots, t_n) : t_i \in T_X(\Sigma_{s_i}) \} \). We will use \( \pi(t_1, \ldots, t_n) \) as an alternative notation for a \( \pi \)-formula to emphasise that \( (t_1, \ldots, t_n) \) in an element in \( F_X(\Omega)_\pi \), for some \( \pi \in \Pi \); in which case we write \( \pi(t_1, \ldots, t_n) \in F_X(\Omega) \) omitting the index \( \pi \) from \( F_X(\Omega)_\pi \).

A Relational Specification is formally defined as a triple \( (\Omega, \phi, A) \) where \( \Omega = (S, F, \Pi) \) is a first order signature, \( \phi \) is a set of Relational rules with formulae over \( \Omega \), and \( A \) is a partial \( \Sigma \)-algebra for \( \Sigma = (S, F) \). The \( \Sigma \)-algebra gives an interpretation for each function symbol in \( F \). For each operation \( \sigma \in F_{w,a} \), the interpretation of \( \sigma \) in \( A \), denoted by \( \sigma^A \), is a partial function from \( A^w \) to \( A_a \). In our example, \( A \) gives the usual mathematical interpretation for the operation “+” and the standard term algebra interpretation for all other symbols in \( F \). Hereafter, we use \( Env \) to refer to the Relational Specification of Fun formed by the Relational rules in Figure 2 and a first order signature \( (S, F, \Pi) \), where \( \Sigma_{Fun} = (S, F) \) and \( \Pi \) is smallest family of sets of relations where \( \Pi_{env \times exp \times val} = \{ _- \vdash _- \Rightarrow _- \} \) and \( \Pi_{env \times exp \times val} = \{ _- \rightarrow_L _- \} \).

In the formalism of Relational Semantics, inference rules may be given an operational and a declarative interpretation. Under the operational interpretation, the rules describe how expressions evaluate to some natural number. For instance, rule (1) states that in an environment \( E \) an expression \( \text{num}(n) \) evaluates to the \( n \) number \( n \); rule (3) states that if (in \( E \)) the expression \( e_1 \) evaluates to a number \( n_1 \) and (in \( E \)) the expression \( e_2 \) evaluates to a number \( n_2 \) then (in \( E \)) the sum expression \( \text{plus}(e_1, e_2) \) evaluates to the result of the function application \( n_1 + n_2 \). This operational interpretation is formalised in [dS92, Chapter 5].

Under the declarative interpretation, the inference rules are seen as inductive rules in the sense of [Acz77]; thus the rules in Figure 2 inductively define two ternary relations: \( \vdash \Rightarrow \subseteq \text{A}_{env} \times \text{A}_{exp} \times \text{A}_{val} \) and \( _- \rightarrow_L \subseteq A_{env} \times A_{var} \times A_{val} \). These relation provides a (first order) \( \Omega \)-model, denoted \( M \), for the first order signature \( \Omega \) of a

\[ \begin{align*}
E \vdash n & \Rightarrow n \\
(E, id) \rightarrow_L v \\
E \vdash e_1 & \Rightarrow n_1 \quad E \vdash e_2 \Rightarrow n_2 \\
E \vdash \text{plus}(e_1, e_2) & \Rightarrow v_1 + v_2 \\
E \vdash e_1 & \Rightarrow v_1 \quad id \Rightarrow v_2 \quad E \vdash e_2 \Rightarrow v_2 \\
E \vdash \text{let id} = e_1 \text{ in } e_2 & \Rightarrow v_2 \\
E \vdash \text{fin id} \cdot e & \Rightarrow (E, id, e) \\
E \vdash e_1 & \Rightarrow (E', id, e') \quad E \vdash e_2 \Rightarrow v_2 \quad id \Rightarrow v_2 \quad E' \vdash e' \Rightarrow v' \\
E \vdash e_1 \Rightarrow v_2 \\
\text{id} \neq id' & \vdash (E, id) \rightarrow_L v \\
\text{id'} \Rightarrow v' & \vdash (E, id) \rightarrow_L v 
\end{align*} \]

Figure 2: An Environment-model Semantics of Fun

\[ \text{1} \text{Exactly how a first order model is derived from a set of Relational rules is not relevant for the purposes of this paper; this can be} \]

1296
Relational Specification. In general, a first order model consists of a $\Sigma$-algebra $A$ and a relation $\pi^M \subseteq A^w$ for each $\pi \in \Pi_w$; $\pi^M$ is the interpretation of $\pi$ in $M$. $\text{Mod}(\Omega)$ is the class of all $\Omega$-models. Let $\Omega \subseteq \Omega'$, and $M \in \text{Mod}(\Omega')$, the reduct of $M$ by $\Omega$, written $M/\Omega$, is an $\Omega$-model that assigns the same interpretation as $M$ to the symbols of $\Omega$. Let $\Sigma = (S,F)$, $\Sigma' = (S',F')$, and $A \in \text{Alg}(\Sigma')$: $\Sigma \subseteq \Sigma'$ denotes the algebraic signature inclusion, and $A/\Sigma$ denotes the reduct of $A$ by $\Sigma$. Furthermore, if $N \in \text{Mod}(\Omega)$ then $N/\Sigma$ denotes the $\Sigma$-algebra part of the model $N$.

The model of $\text{Env}$ is defined using a $\Sigma$-algebra in which the function "+" has the usual mathematical interpretation and all other symbols in $\Sigma$ are given the standard term algebra interpretation. A formalisation of this declarative interpretation is given in [dS92, Chapter 2].

3 A Compiler in Relational Semantics

In this section, we discuss how to specify compilers in the framework of Relational Semantics. In fact, we shall define the semantics of a source program in three stages: the program is compiled into machine code which is loaded and executed on the machine; if this execution is successful then the result of the evaluation is unloaded from the machine and given as the result of the program. The composition of these stages defines a semantics of the programming language which we call a Compilation.

Let us now use an example to illustrate how to define Compilations using Relational Specifications. In agreement with previous approaches, we advocate that the starting point in the design of a Compilation is a formal semantics of the programming language. Therefore, our starting point is a Relational Specification $\text{Env}$ developed in Section 2. Once we have the programming language semantics, the next stage is to define the three stages involved in the Compilation process: the compiler, the machine language semantics, and the unloading of results. In practice, this starts with the definition of the machine language semantics. We will use the Categorical Abstract Machine (CAM) [CCM84] defined using Relational Semantics.

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.

Our definition of the CAM is the first difference between our approach and Despeyroux's approach in [Des86]. Despeyroux defines the CAM by a relation that describes the entire evaluation of a sequence of machine instructions.
into a final result. We define a transition relation that describes the evaluation of a single machine instruction and then use the transitive-reflexive closure of this relation to evaluate sequences of instructions. We use a single step semantics of CAM to illustrate that Relational Semantics can also be used to define transition relations in the style of Structural Operational Semantics [Plo81]. The following grammar rules define the machine language:

```
state ::= (stack, code)
stack ::= ε | val · stack
code ::= ε | inst · code
val ::= nat | funval | (val, val) | ()
nat ::= nat + nat | 0 | 1 | ...
funval ::= [val, code]
inst ::= quote(nat) | push | car | cdr | cons
        | swap | cur(code) | app | add
```

where ε denotes the empty stack, ε 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 use the meta-variables st in Xstate, S in Xstack, c in Xcode, and op in Xinst.

Figure 3 presents the rules that define the relation CAM**: state × state. This relation defines the evaluation of a single CAM instruction. The transitive-reflexive closure of CAM**, written CAM**, defines the evaluation of arbitrary sequences of instructions. The definition of CAM** by a set of Relational Rules is straightforward and is omitted from this presentation.

Once the machine language semantics is defined, the next step in the definition of a Compilation process is to define the compiler. We first define a compiler for Fun into CAM code by presenting a Relational Specification. The rules in Figure 4 define the three components of this compiler. Rule 24 defines the actual compiler a relation E: env × env × state used in rule 24 translates a source level environment into a compilation environment and an initial machine stack. The relation L: env × var × code used in rule 19 defines lookup code generation. The definition of these relations is irrelevant for our purposes and therefore omitted.

This 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 \((S, e)\) where \(e\) 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. In the rest of this section, we will write a sequence of CAM code \(op_1 \ldots op_n \cdot e\) using the usual sequence notation \((op_1, \ldots, op_n)\), to improve readability.

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. The rules in Figure 5 define the unloading relation \(\rightarrow u\) \(u\): state \(\times\) val and the evaluation of \(Fun\) expressions given by the relation \(\vdash : env \times exp \times val\). In rule (25), \(e\) indicates we only unload results from successful states. Rule (26) 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 CAM, and unloading the result from the final CAM state.

Therefore, we can define a Relational Specification using the rules of Figures 3, 4, and 5, which defines a compilation semantics for \(Fun\). Let us denote this specification by \(Comp\), such that its \(\Sigma\)-algebra \(A_{Comp}\) agrees with the \(\Sigma\)-algebra \(A_{Env}\) of \(Env\) in all function names. Hereafter, let \(M_{Comp}\) denote the first order model of \(Comp\).

We illustrated above 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.

### 4 Observational Equivalence and Compiler Correctness

We now study the problem of compiler correctness in the framework of observational equivalence [ST87]. Our objective is to define a suitable criterion for correctness that is more general than previous approach, and yet intuitively sufficient. We proceed by first defining a general equivalence relation between Relational Semantics and then specialise this relation to definition of compiler correctness.

Towards the definition of an equivalence relation between Relational Specifications, we start 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. We then use observational equivalence to define an equivalence relation between Relational Specifications.

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 this work, we are interested in observing the semantics of programs according to some Relational Specification. For an arbitrary Relational Specification \((\Omega, \phi, A)\) the set of ground formulae \(F(\Omega)\) 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 \(\Omega\), called an observation signature, which we will denote by \(\Omega_{OBS}\). Therefore, the set \(F(\Omega_{OBS})\) defines the observations we can make for a given observation signature.

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

The following definitions and theorems use the concept of evaluation of ground terms and ground formulae in a partial \(\Sigma\)-algebra. The evaluation of a ground \(\Sigma\)-term is an \(S\)-sorted partial function \(\psi = \{\psi_s\}_{s \in S}\) such that
ψ_s : T_A(Σ)_s \rightarrow A_s is defined as follows:

1. if \( t = a \), and \( a \in A_s \), for some \( s \in S \), then \( \psi_s(t) = a \).

2. if \( t = σ(t_1, \ldots, t_n), σ : s_1 \times \ldots \times s_n \rightarrow s, t_i \in \text{dom} \psi_{s_i} \), for \( i \in \{1, \ldots, n\} \) and \( n \geq 0 \), and \( (ψ_{s_1}(t_1), \ldots, ψ_{s_n}(t_n)) \in \text{dom} σ^A \), then \( ψ_s(t) = σ^A(ψ_{s_1}(t_1), \ldots, ψ_{s_n}(t_n)) \).

The evaluation of \( Ω \) formulae in \( A \) is a natural extension of \( ψ \) to a \( Ω \)-sorted function \( ψ = \{ψ_π\}_\pi \in Ω \), such that \( \psi_π : F_A(Ω)_π \rightarrow A_w \), for \( π \in Ω_w, w = s_1 \ldots s_n \), and \( n > 0 \), is obtained by applying \( ψ \) to every term in a formula.

We can now present the definition of Observational Equivalence between first order models.

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

1. For all \( s \in S_{OBS} \) and \( t \in T(Σ_{OBS})_s, t \in \text{dom} \psi^A_s \) if and only if \( t \in \text{dom} \psi^B_s \).

2. For all \( π \in Ω_{OBS} \) and \( f \in F(Ω_{OBS})_π, f \in \text{dom} \psi^A_π \) if and only if \( f \in \text{dom} \psi^B_π \) and if both sides of the above equivalence are true then \( \psi^A_π(f) \in π^M \) if and only if \( \psi^B_π(f) \in π^N \).

Definition 1(1) says that the evaluation of a ground observable term is defined in \( A \) if and only if it is defined in \( B \). Definition 1(2) says that the evaluation of an observable formula is defined in \( A \) if and only if it is defined in \( B \), and whenever its evaluation is defined in both algebras then this evaluation is valid in \( M \) if and only if it is valid in \( N \).

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

**Fact 1** For any signature \( Ω \) and \( Ω_{OBS} \subseteq Ω, \equiv_{Ω_{OBS}} \in Mod(Ω) \times Mod(Ω) \) is an equivalence relation on \( Mod(Ω) \), i.e., \( \equiv_{Ω_{OBS}} \) is transitive, reflexive, and symmetric.

**Fact 2** For any signature \( Ω \), and observation signatures \( Ω_{OBS}, Ω'_{OBS} \subseteq Ω \), if \( Ω_{OBS} \subseteq Ω'_{OBS} \) then \( \equiv_{Ω'_{OBS}} \subseteq \equiv_{Ω_{OBS}} \).

Transitivity is necessary to allow the composition of various compilation phases. For instance, the front-end and back-end parts of a compiler can be specified and proved correct in isolation and then joined together consistently due to the transitivity property of the correctness criterion. Among the other properties of \( \equiv_{Ω_{OBS}} \), symmetry requires further comments. In the field of algebraic specification of programs there are various notions of representation between algebras that do not have this property. These include the relations of behavioural inclusion [Sch87, page 216] and simulation [Nip86]. Those relations express the intuition that a program may be a partial implementation of its specification. For instance, a program may be undefined in more arguments than its specification, and yet be considered a suitable implementation.

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

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

On the other hand, there is another kind of non-deterministic behaviour in the concrete definitions of some programming languages that we call non-essential. For instance, in the semantics of Standard ML [HMT90] 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.

We now define an equivalence relation between Relational Specifications based on Observational Equivalence. We first extend a Relational Specification with an observation signature such that whenever $(\Omega, \phi, A)$ is a Relational Specification and $\Omega_{\text{OBS}} \subseteq \Omega$ is an observation signature of $\Omega$, we write the Relational Specification as $(\Omega, \phi, A, \Omega_{\text{OBS}})$. The class of Relational Specifications with observation signature $\Omega_{\text{OBS}}$ is denoted by $\text{Spec}(\Omega_{\text{OBS}})$. Notice that in general the specifications in $\text{Spec}(\Omega_{\text{OBS}})$ have different signatures. The notion of Observational Equivalence between Relational Specifications will account for this.

For instance, let $\Omega_{\text{Env}}^{\text{Env}}$ be the observation signature for the Relational Specification $\text{Env}$ defined as follows. The set $\Pi_{\text{Obs}}^{\text{En}}$ has only the relation $\cdot \rightarrow \cdot$, and the sets $S_{\text{Obs}}^{\text{En}}$ and $F_{\text{Obs}}^{\text{En}}$ are defined by the following grammar rules:

\[
\begin{align*}
\text{exp} & ::= \text{var} \mid \text{nat} \mid \text{exp} + \text{exp} \mid \text{let } \text{var} = \text{exp} \text{ in } \text{exp} \\
& \quad \mid \text{fn } \text{var} . \text{exp} \mid \text{exp}(\text{exp}) \\
\text{env} & ::= \epsilon_E \\
\text{var} & ::= x \mid y \mid \ldots \\
\text{nat} & ::= 0 \mid 1 \mid \ldots \\
\text{funval} & ::= \\
\text{val} & ::= \text{nat}
\end{align*}
\]

The observable terms according to these grammar rules are the expressions of sort $\text{exp}$, the empty environment, variables, and values of sort $\text{nat}$ of the form 0, 1, and so forth. The sort $\text{funval}$ is observable, but there is no constructor for building observable terms of this sort. That means we cannot neither observe nor alter the concrete representation of a function value. We now extend $\text{Env}$ with $\Omega_{\text{Obs}}^{\text{En}}$ such that the new Relational Specification is written $(\Omega_{\text{En}}, \phi_{\text{En}}, A_{\text{En}}, \Omega_{\text{Obs}}^{\text{En}})$. This definition of $\text{Env}$ will be used in the rest of this presentation.

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

Definition 2 (Equivalence of Relational Specifications) Two Relational Specifications $S$ and $R$ in $\text{Spec}(\Omega_{\text{Obs}})$, with Declarative Semantics $M^S$ and $M^R$ respectively, are observationally equivalent, written $S \equiv R$, if and only if $M^S/\Omega_{\text{Obs}} \equiv M^R/\Omega_{\text{Obs}}$.

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

We now study compiler correctness in the context of the theory of Observational Equivalence developed above. The generality of the framework of Observational Equivalence 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. We study these extra requirements in the sequel.

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) 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. Since arrows are relations, our approach deals directly with non-deterministic languages, in contrast with the approaches in [Mor73, TWW81]. We illustrate our interpretation of compiler correctness using the diagrams in Figure 6.

In Figure 6, the nodes $L$, $M$, $T$, $U$, and $M'$ are term algebras, the double arrows are relations and the single arrow is a partial function. Both diagrams describe first order models of some signature $\Omega$. The left diagram describes an

\[\text{Notice that we omitted from the grammar rules the coercion operators num, id and so on.}\]
Figure 6: Evaluation by Compilation

\[ \begin{array}{c}
\text{programming language} \\
\downarrow \pi \\
\equiv_{\text{OBS}} \\
\downarrow \\
\text{machine language} \\
\downarrow \mu \\
\text{unloading} \\
\downarrow \\
\text{results} \\
\end{array} \]

The right diagram formalises the notion of Compilation discussed in Section 3. A Compilation is an \( \Omega \)-model \( LTUM' \) in which the semantics of a program is defined by the composition of the compiler \( \gamma : L \to T \), the machine language semantics \( \mu : T \to U \), and the unloading of results \( \delta : U \to M' \).

The conditions for compiler correctness are also expressed by the diagrams in Figure 6 and can be formalised as follows.

**Definition 3 (Compiler Correctness)** Let \( \Omega_{\text{OBS}} \subseteq \Omega \) be an observation signature. The Compilation \( LTUM' \) is correct with respect to \( LM \) and \( \Omega_{\text{OBS}} \) if and only if \( \gamma \) is a (partial) function and \( LM \equiv_{\Omega_{\text{OBS}}} LTUM' \). Whenever the \( \Omega \)-models \( LM \) and \( LTUM' \) are defined by Relational Specifications in \( \text{Spec}(\Omega_{\text{OBS}}) \), say \( S \) and \( C \) respectively, the second requirement becomes \( S \cong C \).

For instance, to establish the correctness of the compiler defined in Section 3, we must show that the relation \( - \vdash - \longrightarrow_{\text{Comp}} \) is a partial function and that \( \text{Env} \cong \text{Comp} \) holds. 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 \( \gamma \) to be a partial function instead of an arbitrary relation. 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, \( \pi \) will be a relation in which any program in \( L \) may be related to more than one result in \( M \). 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 \( \pi \).

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 \( \gamma \) to be a (partial) function.

At this stage, we achieved the main goals of this article: we showed how to define compilers in Relational Semantics and define a criterion for compiler correctness based on observational equivalence. Let us now briefly...
discuss the problem of giving concrete proofs of Observational Equivalence between two models. These proofs are essential in establishing the equivalence of Relational Specifications and therefore are also essential in proofs of compiler correctness. 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.

Some authors have proposed proof methods to overcome this problem. In [Hen90] Context Induction is proposed as a proof method for behavioural abstractions. 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. This proof method naturally generalises to Observational Equivalence as defined above. In [dS92, dS94], we extended the correspondence relations to a proof method for Observational Equivalence between first order models, which we call Model Correspondence, and used this method to give proofs of compiler correctness. We also proved that Model Correspondence is sound and complete with respect to Observational Equivalence and can indeed be used as a proof method.

5 Concluding Remarks

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

It is generally agreed that a major contribution of the ideas in, for instance [Mor73, TWW81], 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, 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.

The Relational Semantics formalism may be extended by using order-sorted algebras and order-sorted first order models. An order-sorted algebra extends the expressiveness of many-sorted algebras by adding a partial ordering relation on the sorts of the algebra. This would involve changing the definition of Observational Equivalence and related results. We believe this extension would simplify the use of Relational Semantics in practical examples. However, we have not studied the effects of this extension in proofs of compiler correctness.

A natural problem for future research is to provide semi-automatic support for proofs of compiler correctness using Model Correspondences. This would help applying the methods discussed here to examples of larger size and complexity. The use of semi-automatic proof assistants can help in this task, as already has been demonstrated in some approaches in the literature, e.g., [Sim90, HP92].

References


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.

