Proving Compiler Correctness with Evolving Algebra Specifications

Bernhard Beckert Reiner Hähnle

1 Introduction

The purpose of this note is to define a framework for proving compiler correctness with evolving algebra (EA) specifications [2]. Although our specific domain is the verification of a Prolog-to-WAM compiler [1, 3], we think that our considerations are fairly general and they should be useful in other areas as well.

The starting point for us was the observation that the notions of correctness and completeness as used in [1] become quite counterintuitive when seen from the point of view of compiler construction.

First we will define our general view of the semantics of a programming language, of how semantics can be specified using EAs, and of compiler correctness; then we describe how the correctness of a compiler may be proven; and finally we point out the differences to the approach of [1] and to the notion of correctness as commonly used in logic.

2 General View

2.1 Semantics of a Programming Language

A *language* $\mathcal{L}$ is a set of (well-formed) programs. Associated with each language is a *domain* $\mathcal{I}$ *of input values* and a *domain* $\mathcal{O}$ *of output values*.[1]

The *semantics* Φ of a language is a total function

$$\Phi \; : \; \mathcal{L} \times \mathcal{I} \; \rightarrow \; \mathcal{O} \cup \{\perp\}$$

that assigns to each program $p \in \mathcal{L}$ and input value $i \in \mathcal{I}$ an output value $\Phi(p, i) \in (\mathcal{O} \cup \{\perp\})$, where $\perp$ denotes non-termination.

2.2 Evolving Algebras

An *evolving algebra* $\mathcal{A} = \langle \mathcal{T}, \Sigma \rangle$ consists of a set $\mathcal{T}$ of transition rules and a signature Σ. By $\mathfrak{A}$ we denote the set

$$\mathfrak{A} = \{A \; : \; A \text{ is in the similarity class of } \Sigma\}$$

of all *static algebras* associated with $\mathcal{A}$.

A static algebra $A \in \mathfrak{A}$ is called *terminal* iff none of the transition rules in $\mathcal{T}$ is applicable to A.

[1] In the case of Prolog, $\mathcal{L}$ is the set of all (well-formed) Prolog programs, $\mathcal{I}$ is the set of all (well-formed) queries, and $\mathcal{O}$ is the set containing all answer substitutions and the special symbol *fail*.

2.3 Using an Evolving Algebra to Specify the Semantics of a Language

An EA $\mathcal{A}$ can be used to formally describe the semantics Φ of a language $\mathcal{L}$. The static algebras $A \in \mathfrak{A}$ represent a program p with input i and its associated current computation state. The transition rules $\mathcal{T}$ of $\mathcal{A}$ specify an interpreter for programs and computation states.

In addition, to define Φ, we associate with each program p and input i, an initial algebra $A^0 = \mathit{Init}(p, i) \in \mathfrak{A}$, where Init is a total function

$$\mathit{Init} \ : \mathcal{L} \times \mathcal{I} \ \rightarrow \ \mathfrak{A} \ .$$

Finally, an output function

$$\mathit{Out} \ : \mathfrak{A} \ \rightarrow \ \mathcal{O}$$

has to be given. *Out* may be partial; it only has to be defined on terminal algebras.

Since we defined the semantics Φ to be a function, i.e., the programs of the language to be deterministic, the EA $\mathcal{A}$ we use to describe the semantics has to be deterministic, too:[2] To each $A \in \mathfrak{A}$ either exactly one transition rule is applicable (if A is non-terminal) or no rule is applicable (if A is terminal). In general, this property of $\mathcal{A}$ has to be proven separately.[3]

From here on we assume $\mathcal{A}$ to be deterministic. Thus, for each initial algebra A^0 there is either

1. exactly one infinite sequence $A^0 \xrightarrow{\mathcal{T}} A^1 \xrightarrow{\mathcal{T}} A^2 \xrightarrow{\mathcal{T}} \cdots$
2. or exactly one finite sequence $A^0 \xrightarrow{\mathcal{T}} A^1 \xrightarrow{\mathcal{T}} \cdots \xrightarrow{\mathcal{T}} A^{\text{fin}}$ such that A^{fin} is terminal.

We call this sequence the $\mathcal{A}$-sequence for A^0.

Now, the semantics Φ can be described (resp. defined) using $\mathcal{A}$, *Init*, and *Out* in the following way:[4]

$$\Phi(p,i) = \begin{cases} \mathit{Out}(A^{\text{fin}}) & \text{if the } \mathcal{A}\text{-sequence for } A^0 = \mathit{Init}(p,i) \text{ is finite and} \\ & A^{\text{fin}} \text{ its final element} \\ \perp & \text{if the } \mathcal{A}\text{-sequence for } A^0 = \mathit{Init}(p,i) \text{ is infinite} \end{cases}$$

One consequence of the use of EAs is that programs, inputs and computation states are not properly separated in the static algebras and in the transition rules. Neither is it desirable to separate them as this causes only syntactical overhead and has otherwise no advantages.

2.4 Compiler Correctness

If two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are given with input domains $\mathcal{I}_1, \mathcal{I}_2$ and a single output domain $\mathcal{O}$, then a total[5] function

$$C \ : \ (\mathcal{L}_1 \times \mathcal{I}_1) \ \rightarrow \ (\mathcal{L}_2 \times \mathcal{I}_2)$$

is called a *compiler* (that compiles the "source language" $\mathcal{L}_1$ into the "target language" $\mathcal{L}_2$). Note, that a compiler does not generate an initial algebra, but a program/input pair of the target language. In general, a compiler is neither an injective nor a surjective function.

Given semantics Φ_1 for the source language and Φ_2 for the target language, the compiler C is *correct* (w.r.t. Φ_1 and Φ_2) iff for each $p \in \mathcal{L}_1$ and $i \in \mathcal{I}_1$

$$\Phi_1(i,p) = \Phi_2(C(i,p)) \ .$$

[2] The semantics of a non-deterministic language is not a function but a relation between $\mathcal{L} \times \mathcal{I}$ and $\mathcal{O} \cup \perp$.

[3] In the case of the Prolog-to-WAM compiler, we focus on, the EAs are defined in such a way, that it follows immediately from the syntactical form of the transition rules that they are deterministic.

[4] In practice, *Init* and *Out* should be (and in the case of the Prolog-to-WAM compiler are) "simple" to compute functions, at least they should be computable. In theory, however, this is not necessary: neither *Init*, nor *Out*, nor the transition rules of $\mathcal{A}$ have to be computable to define Φ.

[5] We do not define completeness of compilers since it does not make much sense in the present setting. It boils down to the fact that C is total.

Figure 1 illustrates the logical dependencies between some of the notions introduced, where the semantics of the two languages are given by EAs $\mathcal{A}_1 = \langle \mathcal{T}_1, \Sigma_1 \rangle$ and $\mathcal{A}_2 = \langle \mathcal{T}_2, \Sigma_2 \rangle$, initialization functions $Init_1, Init_2$, and output functions Out_1, Out_2 as described above.[6] In that case, the correctness of C can be formally defined in the following way: C is correct iff for each $p \in \mathcal{L}_1$ and $i \in \mathcal{I}_1$:

1. If the $\mathcal{A}_1$-sequence for $Init_1(p,i)$ is finite with final element $A_1^{\mathrm{fin}_1}$, then the $\mathcal{A}_2$-sequence for $Init_2(C(p,i))$ is finite with final element $A_2^{\mathrm{fin}_2}$, and
$$Out_1(A_1^{\mathrm{fin}_1}) = Out_2(A_2^{\mathrm{fin}_2}) \ .$$
2. If the $\mathcal{A}_1$-sequence for $Init_1(p,i)$ is infinite, then the $\mathcal{A}_2$-sequence for $Init_2(C(p,i))$ is infinite.

Because $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic, this is equivalent to:

1′. If the $\mathcal{A}_1$-sequence for $Init_1(p,i)$ is finite with final element $A_1^{\mathrm{fin}_1}$ and the $\mathcal{A}_2$-sequence for $Init_2(C(p,i))$ is finite with final element $A_2^{\mathrm{fin}_2}$, then
$$Out_1(A_1^{\mathrm{fin}_1}) = Out_2(A_2^{\mathrm{fin}_2}) \ .$$

2′. The $\mathcal{A}_1$-sequence for $Init_1(p,i)$ is finite if and only if the $\mathcal{A}_2$-sequence for $Init_2(C(p,i))$ is finite.

Note, however, that the compiler C is not surjective (in general). Therefore, its correctness is not related to a property of all $\mathcal{A}_2$-sequences, nor of all $\mathcal{A}_2$-sequences starting with an initial algebra $Init_2(p', i')$ (where $p' \in \mathcal{L}_2$ and $i' \in \mathcal{I}_2$),[7] but only of all $\mathcal{A}_2$-sequences starting with an initial algebra $Init_2(C(p,i))$ (where $p \in \mathcal{L}_1$ and $i \in \mathcal{I}_1$)

$$\begin{tikzcd}
\langle p_1, i_1 \rangle \arrow[r, "Init_1"] \arrow[d, "C"'] & A_1^0 \arrow[r, "\mathcal{T}_1"] & A_1^1 \arrow[r, "\mathcal{T}_1"] & \cdots \arrow[r, "\mathcal{T}_1"] \arrow[d, "R"'] & A_1^{\mathrm{fin}_1} \arrow[r, "Out_1"] & o \arrow[d, equal] \\
\langle p_2, i_2 \rangle \arrow[r, "Init_2"] & A_2^0 \arrow[r, "\mathcal{T}_2"] & A_2^1 \arrow[r, "\mathcal{T}_2"] & \cdots \arrow[r, "\mathcal{T}_2"] & A_2^{\mathrm{fin}_2} \arrow[r, "Out_2"] & o
\end{tikzcd}$$

Figure 1: Schematic diagram of compiler correctness with EAs.

Example 1 *As an example for the general setup we consider the correctness of the first two levels of the Prolog-to-WAM compiler as given in [3].*

Here, the languages $\mathcal{L}_1 = \mathcal{L}_2$ consist of all Prolog programs, the input domains $\mathcal{I}_1 = \mathcal{I}_2$ contain all Prolog queries, and the output domain $\mathcal{O}$ contains all answer substitutions and the special symbol fail.

The particular compiler is the identity function and therefore trivial. Note that this does not mean that the correctness of this particular compilation step is trivial to prove. The transition rules of the two evolving algebras are different, and thus the two semantics Φ_1 and Φ_2 are defined differently. In that case, proving the correctness of the compiler amounts to proving that $\Phi_1(p,i) = \Phi_2(p,i)$ for all $p \in \mathcal{L}$ and $i \in \mathcal{I}$, i.e., $\Phi_1 = \Phi_2$.

[6]The Relation R shown in the figure is described in Section 3.

[7]Between the first two levels the Prolog-to-WAM compiler is the identity function (see Example 1) and thus surjective. This, however, should not be considered to be a typical example.

3 Proving Compiler Correctness

In order to carry out an actual correctness proof, we need to make a link between static algebras A_1^i and A_2^j. This is done with a suitable auxiliary relation R whose validity, of course, must be proved as well. The relation R is not part of the statement of the correctness theorem, but is introduced only during its proof. Note that R is not necessarily defined explicitly in the proof.

The correctness proof[8] has the general form of an induction on the length of the $\mathcal{A}_1$-sequence of algebras. The induction base involves showing the correctness of the mapping *Init* from programs to initial algebras. For the induction step both the $\mathcal{A}_1$- and the $\mathcal{A}_2$-sequence must be partitioned into appropriate segments that correspond to each other. In most cases $\mathcal{A}_2$ is a proper refinement of $\mathcal{A}_1$ which means that one transition of $\mathcal{A}_1$ corresponds to one or more transitions of $\mathcal{A}_2$. This gives roughly the following picture:

$$\begin{tikzcd}
A_1^i \arrow[rr, "\mathcal{T}_1"] \arrow[d, "R"'] & & A_1^{i+1} \arrow[d, "R"'] \\
A_2^j \arrow[r, "\mathcal{T}_2"] & \cdots \arrow[r, "\mathcal{T}_2"] & A_2^{j+k}
\end{tikzcd}$$

It has to be part of the invariant of the induction that for each A_1^i there is a A_2^j such that $R(A_1^i, A_2^j)$, where A_2^j is terminal whenever A_1^i is terminal. To guarantees Part 1 of correctness, R has to be chosen such that if $R(A_1^{\mathrm{fin}_1}, A_2^{\mathrm{fin}_2})$ and $A_1^{\mathrm{fin}_1}, A_2^{\mathrm{fin}_2}$ are terminal, then $Out_1(A_1^{\mathrm{fin}_1}) = Out_2(A_2^{\mathrm{fin}_2})$.[9]

If $k \geq 1$ in all cases, then each $\mathcal{A}_2$-sequence is at least as long as the corresponding $\mathcal{A}_1$-sequence. This property guarantees Part 2 of correctness. However, if $\mathcal{A}_2$ is not a proper refinement of $\mathcal{A}_1$ this might not be the case. More generally, we might have a m-to-k correspondence instead of an 1-to-k correspondence between transitions in the induction step. This can occur when compilation involves code optimization that leads to a removal of redundant instructions (tail recursion is one example).

4 Relationship to Other Notions

In the paper of Börger & Rosenzweig [1] correctness means Part 1 of our definition. They work also with the notion “completeness” which in their framework simply means Part 1 of correctness with $\mathcal{A}_1$ and $\mathcal{A}_2$ exchanged (note that Part 2 of our correctness follows from completeness in the sense of Börger & Rosenzweig). Clearly their notions are derived from “correctness” and “completeness” as they are used in logic between, say, a calculus and model semantics. In logic, however, there is no refinement relation in either direction between calculi and semantics, because the respective algebras are not evolving algebras, but simply abstract algebras in the usual sense.

The very notion of a compiler from a source language into a target language, however, introduces a preferred direction between the respective EAs, even if the target machine cannot be considered as a proper refinement of the source machine. This means that, if we fix the natural direction as the correctness direction, the notion of completeness is not needed, because it would correspond to “correctness of decompilation” which is pretty uninteresting and very difficult to prove in general. We only need a very weak form of completeness hidden as the non-termination property of our correctness notion; it is ensured for proper refinements anyway.

As a consequence, it is not really necessary to formally prove both correctness and completeness in the WAM case study, rather, it is sufficient to prove correctness for each compilation level as it is defined above.

[8] That is, the constructive proof we are thinking of; there might be other ways to prove compiler correctness.
[9] In the case of Prolog, this means that on both levels identical variable bindings are maintained.

Another difference between the present approach and the one in [1] is that in the latter both the function C and the relation R appear in the *statement* of correctness, where it is called $\mathcal{F}$. We find it clearer to distinguish between compilation of programs (done by C) and "compilation" of the static algebras (done by R), that represent computation states. In logic we have this distinction as well: there, compilation is usually trivial and involves, for instance, mapping of objects of the semantics defining abstract algebra to clauses etc.; R, for instance, might correspond to the semantical validity of the inference schema of resolution etc. The latter kind of relations are typically introduced during a correctness (or completeness) proof, but they are not part of the theorem.

References

[1] Egon Börger and Dean Rosenzweig. The WAM—definition and compiler correctness. In Ch. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*. North-Holland, Amsterdam, 1995.

[2] Yuri Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[3] Peter H. Schmitt. Proving WAM compiler correctness. Interner Bericht 33/94, Universität Karlsruhe, Fakultät für Informatik, 1994.