

Compiler Correctness using Algebraic Operational Semantics

K Stephenson

Department of Computer Science,
University Wales Swansea,
Swansea, SA2 8PP. UK.

November 5, 1998

1 Introduction

Many attempts at establishing the correctness of compilation have used the method introduced in Morris [1973], subsequent to its extension and widespread publicisation by Thatcher *et al.* [1981]. The essence of the advice is that the correctness of a compiler should be expressed within an algebraic framework. The syntax and semantics of the source and target languages can be modelled by algebras with a common signature. Then compiler correctness can be stated as an equation

$$\mathit{encode}(\mathit{semantics}_{\mathit{source}}(P)) = \mathit{semantics}_{\mathit{target}}(\mathit{compile}(P)),$$

requiring the diagram shown in Figure 1 to commute.

$$\begin{array}{ccc} \mathit{Prog}_1 & \xrightarrow{\mathit{compile}} & \mathit{Prog}_2 \\ \mathit{semantics}_{\mathit{source}} \downarrow & & \downarrow \mathit{semantics}_{\mathit{target}} \\ \mathit{Semantics}_1 & \xrightarrow{\mathit{encode}} & \mathit{Semantics}_2 \end{array}$$

Figure 1: Traditional view of compiler correctness.

The problem with modelling lies in the details of the design of the algebras and homomorphisms for the two languages and the proof of the correctness equation. For example, the semantic algebras $\mathit{Semantics}_i$ are commonly based on an algebra of partial state transformations of the form $[\mathit{State}_i \rightsquigarrow \mathit{State}_i]$, for $i = 1, 2$. These are *partial* computable functions, mapping an initial state $\sigma_i \in \mathit{State}_i$ to the final state (if it exists) that results from executing some program $P_i \in \mathit{Prog}_i$ on the state σ_i . Hence we are dealing with a *partial* equation for correctness, which is of Π_2 complexity to validate in partial computable algebras.

We improve on this situation by designing a general model of semantics in which we have computable algebras involving *total* functions. This reduces the problem of determining validity of

compiler correctness equations to that of Π_1 logical complexity. We also give a general strategy for correctness proofs.

We give a method in Section 2 of defining an operational semantics $OS : Prog \times State \rightarrow State^\omega$ such that $OS(P, \sigma_0)$ gives a finite $\sigma_0, \sigma_1, \dots, \sigma_t$, or infinite $\sigma_0, \sigma_1, \dots, \sigma_t, \dots$, sequence of states produced by executing a program $P \in Prog$ on an initial state $\sigma_0 \in State$ using equations (Section 2.2). With our model of semantics we can model the complex correspondence between deterministic processes taking place in time. The new algebraic semantics has the form

$$Exec_i : (Prog_i \times State_i \times Time_i) \rightarrow (Prog_i \times State_i \times Time_i)$$

for $i = 1, 2$, such that $Exec_i(P_i, \sigma_i, t_i)$ gives the program, state and time that remain after executing the program $P_i \in Prog_i$ on the state $\sigma_i \in State_i$ for $t_i \in Time_i$ clock cycles.

We now have to decide in Section 3 how we relate two operational semantics OS_1 and OS_2 ; we have two sequences of states to relate. Let $\bar{c} : Prog_1 \rightarrow Prog_2$ be a compiler such that $\bar{c}(P_1)$ gives the target program that results from compiling a source program $P_1 \in Prog_1$. Then we define the correctness of the compiler \bar{c} by constructing an abstracting homomorphism

$$\Phi : (Prog_2 \times State_2 \times Time_2) \rightarrow (Prog_1 \times State_1 \times Time_1)$$

between the target and source models, so that the diagram shown in Figure 2 commutes on correctly initialised target systems.

$$\begin{array}{ccc}
 Prog_1 \times State_1 \times Time_1 & \xrightarrow{Exec_1} & Prog_1 \times State_1 \times Time_1 \\
 \uparrow \Phi & & \uparrow \Phi \\
 Prog_2 \times State_2 \times Time_2 & \xrightarrow{Exec_2} & Prog_2 \times State_2 \times Time_2
 \end{array}$$

Figure 2: Operational View of Compiler correctness.

The next step is to address the question of proving correctness. Using our model of semantics, we need only use *equational* reasoning coupled with induction over the natural numbers to show that the compiler is correct, if and only if, it is correct over one step of time for any correctly initialised system. (One Step Theorem 3.3.2). This defines a strategy for equational correctness proofs. We illustrate this strategy in Section 3.4 by considering a compiler from **while** programs to Unlimited Register Machine programs.

Although we have not yet done so, this method can be mechanised using term rewriting systems; if the data type over which computations are performed can be described with a complete term-rewriting system (TRS), then the whole system will be a complete TRS as everything is primitive recursively defined over underlying data type (RESULT OF ROB).

2 Algebraic Operational Semantics

An operational semantics for a programming language provides a mathematical model of the way programs in the language perform computations, based on ideas about how the instructions or constructs in a program operate, step by step. In many cases, the operational semantics defines how

a program P generates a finite or convergent sequence $\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_t$, or an infinite or divergent sequence $\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_t, \dots$, of computation states, from an initial state σ_0 .

We develop and generalise an idea from Tucker and Zucker [1988] for expressing the operational semantics of a language equationally (further details can be found in Stephenson [1996]). Algebraic Operational Semantics (AOS) combines the notion of time with Plotkin's Structural Operational Semantics (Plotkin [1981], Plotkin [1983]) in an axiomatic fashion. The basis of AOS is a function

$$Comp : Prog \times State \times Time \rightarrow State$$

such that $Comp(P, \sigma_0, t)$ gives the state σ_t that results from executing the program $P \in Prog$ on the initial state $\sigma_0 \in State$ for $t \in Time$ cycles of time.

Suppose we have some function $Act : AProg \times State \rightarrow State$ such that $Act(\alpha, \sigma)$ gives the behaviour of a basic or *atomic* program $\alpha \in AProg$ on a state $\sigma \in State$. AOS simulates the behaviour of a program by executing some finite $\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_t$, or infinite $\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_t, \dots$, sequence of atomic programs such that the t^{th} execution state is given by

$$\sigma_t = Act(\alpha_{t-1}, \sigma_{t-1}) = Comp(P, \sigma_0, t).$$

2.1 Components of AOS

First, we want to develop a completely algebraic description of the technique of operational semantics in an analogous fashion to that of algebraic semantics for denotational semantics (Goguen *et al.* [1977], Nivat and Reynolds [1985], Goguen and Malcolm [1996]) Our first consideration is how we can model the sets $Prog$ of programs, $State$ of states and $Time$ of clock cycles.

2.1.1 Programs There exist established techniques to translate any context-free grammar into a closed term algebra (Rus [1971], Goguen *et al.* [1977]). Less well developed are techniques to deal with non-context-free languages, but see Broy *et al.* [1981], Courcelle and Franchi-Zanettacci [1982]. In Stephenson [1996] we establish that we can model any recursively enumerable language as a computable algebra. Consequently, we know that we can algebraically specify a recursively enumerable language in a number of ways (for example, Bergstra and Tucker [1982], Bergstra and Tucker [1996]).

Furthermore, in Stephenson [1996], we describe a general method of how such equational specifications can be developed in practice. Using this technique, we formulate an algebraic specification of a language using hidden functions to act as a filter for any context-sensitive constraints of the language; we equationally define these operations over the terms that are induced by the context-free grammar.

Finally, we need to decide on the set $AProg$ of atomic programs that we shall use to express the behaviour of any program.

2.1.2 States At one level of abstraction, we simply require that we can model the set $State$ of states as an algebra in some way, and that we have some distinguished state $* \in State$ which we use to denote that a computation has terminated. When we come to consider the semantics of a particular language, we have to add further structure to our model of states. Typically, modelling the state set of a high-level language revolves around the set $[Var \rightarrow A]$ of all maps from the set Var of variables of the language to their values in the set A of data over which we compute.

2.1.3 Time We use a simple clock consisting of discrete intervals of time to enumerate the sequences of states produced by executing a program. We model the clock with the algebra $Time = (\{0, 1, 2, \dots\}; 0 : \rightarrow Time, Succ : Time \rightarrow Time)$.

2.2 Methodology

We define each transition σ_t, σ_{t+1} from one state to the next in a computation sequence in terms of the behaviour of some atomic program of the language. Our task is to determine which atomic program we are to execute at each moment of time.

2.2.1 Decomposing Syntax We need to be able to determine the sequence of atomic programs that we will use to mimic the behaviour of any program on any initial state. We begin by defining a function

$$First : Prog \times State \rightarrow AProg$$

such that $First(P, \sigma)$ gives the atomic program that we will execute to simulate the execution of the program $P \in Prog$ on the state $\sigma \in State$ in the first cycle of time. We couple this with a function

$$Rest : Prog \times State \rightarrow Prog$$

such that $Rest(P, \sigma)$ gives the rest of the program $P \in Prog$ that we need to consider after one cycle of execution from the initial state $\sigma \in State$. Note that we sometimes find it useful to extend this function to give a time-dependent operation $Rest^T : Prog \times State \times Time \rightarrow Prog$.

2.2.2 Semantics The function

$$Comp : Prog \times State \times Time \rightarrow State$$

gives the semantics of any program by repeatedly applying the syntax decomposition functions $First$ and $Rest$, together with the atomic program semantics function Act . We define $Comp$ for any $P \in Prog$, $\sigma \in State$ and $t \in Time$ by:

$$\begin{aligned} Comp(P, \sigma, 0) &= \sigma \\ Comp(P, \sigma, 1) &= Act(First(P, \sigma), \sigma) \\ \forall t \geq 1 : Comp(P, \sigma, t + 1) &= \begin{cases} * & \text{if } P \text{ is atomic;} \\ Comp(Rest(P, \sigma), Comp(P, \sigma, 1), t) & \text{if } P \text{ is not atomic.} \end{cases} \end{aligned}$$

For our application of compiler correctness, we extend the function $Comp$ to that of

$$Exec : (Prog \times State \times Time) \rightarrow (Prog \times State \times Time)$$

which we define for any $P \in Prog$, $\sigma \in State$ and $t \in Time$, by:

$$\begin{aligned}
Exec(P, \sigma, 0) &= (P, \sigma, 0) \\
Exec(P, \sigma, t + 1) &= (Rest(P, \sigma), Comp(P, \sigma, 1), t)
\end{aligned}$$

2.2.3 Example Consider a simple **while** programming language which contains identity statements (**skip**), assignments, sequencing, conditionals and iteration. We can express the behaviour of any **while** program in terms of the semantics of **skip** and assignment statements. We decompose any program into a sequence of such atomic programs by defining the functions *First* (which is state-independent for **while** programs) and *Rest*, for any variable x , expression e , **while** programs S_0 , S_1 and S_2 , and any state σ , by:

$$\begin{aligned}
First(\mathbf{skip}) &= \mathbf{skip} \\
Rest(\mathbf{skip}, \sigma) &= \mathbf{skip} \\
\\
First(x := e) &= x := e \\
Rest(x := e, \sigma) &= \mathbf{skip} \\
\\
First(S_1; S_2) &= First(S_1) \\
Rest(S_1; S_2, \sigma) &= \begin{cases} S_2 & \text{if } S_1 \text{ is atomic;} \\ Rest(S_1, \sigma); S_2 & \text{otherwise.} \end{cases} \\
\\
First(\mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \mathbf{ fi}) &= \mathbf{skip} \\
Rest(\mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \mathbf{ fi}, \sigma) &= \begin{cases} S_1 & \text{if } \llbracket b \rrbracket(\sigma) = tt; \\ S_2 & \text{if } \llbracket b \rrbracket(\sigma) = ff. \end{cases} \\
\\
First(\mathbf{while } b \mathbf{ do } S_0 \mathbf{ od}) &= \mathbf{skip} \\
Rest(\mathbf{while } b \mathbf{ do } S_0 \mathbf{ od}, \sigma) &= \begin{cases} S_0; \mathbf{while } b \mathbf{ do } S_0 \mathbf{ od} & \text{if } \llbracket b \rrbracket(\sigma) = tt; \\ \mathbf{skip} & \text{if } \llbracket b \rrbracket(\sigma) = ff. \end{cases}
\end{aligned}$$

These definitions, coupled with that of the function *Comp*, the semantics of the atomic programs (*Act*) and the manner in which we perform tests ($\llbracket \cdot \rrbracket$), provide a complete description of the AOS of **while** programs.

2.2.4 Example (Low-level) Although AOS is primarily designed for high-level languages, it can be easily adapted to deal with low-level languages. Consider the language of Unlimited Register Machines (URMs) of Shepherdson and Sturgis [1963]. We adapt this slightly to give the language consisting of sequences of labelled instructions that take one of the following forms:

Data manipulating For each function of the underlying data type over which we perform calculations, we have an instruction that applies this function to values stored in specified registers of the state, and stores this result in a specified register; we also have **copy** instructions that duplicate a value stored in one register into another.

Flow of control We use **jump** instructions to conditionally transfer control to another instruction with a specified label, so interrupting the otherwise sequential order of instruction execution.

We use one register of the state as a program counter to record the label of the instruction that we are to execute next. If there is no such label in a program, this forces termination.

To remove semantical problems associated with **jump** instructions we can impose syntactical restrictions on the set of URM programs to ensure that we have a unique instruction to execute at any one moment in time.

It is a simple matter to define the function *First* to locate the appropriate instruction that we are to execute. To ensure that it is a total function though, we add a new instruction **halt** to our instruction set that forces termination; the individual URM instructions together with **halt** form the set of atomic programs of the language. We define the behaviour of these atomic instructions with the function *Act*. Finally we redefine the function *Comp* for low-level languages by:

$$\begin{aligned} \text{Comp}(P, \sigma, 0) &= \sigma \\ \text{Comp}(P, \sigma, t + 1) &= \text{Comp}(P, \text{Act}(\text{First}(P, \sigma), \sigma), t) \end{aligned}$$

Note that we have removed the function *Rest* from our definitions because we always need to have the whole program available for inspection as we are dealing with a low-level program that has unstructured flow-of-control constructs (**jump** instructions).

3 Compiler Correctness

Suppose we have descriptions of the syntax and AOS of the source and target languages of a compiler. We want a structured model of compilation that we can use to analyse the correctness of the compiler.

3.1 Structuring

We have taken the first steps in structuring the compiler in describing the syntax and semantics of the source and target languages as algebras. We need to impose further structure by describing how we can form the target programs produced by compilation. Similarly, we shall need to determine how we can organise the data stored in the target states in a manner that facilitates the description of the behaviour of the target programs. (This is in anticipation of Section 3.2.2, where we consider exactly how we wish to relate the source and target states.)

3.1.1 Programs From previous work on compiler correctness set within an algebraic framework (starting with Burstall and Landin [1969], and principally including that of Morris [1973] and Thatcher *et al.* [1981]), we know that we can describe a compiler $\bar{c} : \text{Prog}_1 \rightarrow \text{Prog}_2$ as a homomorphism which requires us to construct a new algebra Prog_2 of target programs which is structured in the same manner as the set Prog_1 of source programs.

3.1.2 States The AOS models for the source and target languages provide us with algebraic descriptions of the source and target state sets. Just as we have imposed additional structure on the set of compiled target programs, we can also organise the target states in a manner that reflects the

operation of the compiled target programs. In particular, we shall need to be able to extract from the target state those elements which are to correspond (in some manner) to the values we store in the source states. These particular elements will form a sub-algebra which will be homomorphic to the source state algebra.

We shall find it useful to split the remaining elements of the target states into two classes. We suppose that we can extract in some way (typically by projection) the *canonical* elements of the state which are those values that initially need to be set to some specific value in order for calculations to proceed in the target system as required. An example of a canonical element is a program counter that stores information regarding the next instruction to be executed. The remaining elements of the state we term *non-canonical*. These elements can initially have any value, as they do not affect the proper execution of a system.

3.1.3 Time The clocks of both systems of AOS are isomorphic, but the rate at which they will run relative to each other will differ (in general) though; in Section 3.2.3, we describe how we can relate them using retimings.

3.2 Relating Models

Suppose we have structured our target system in the manner described in Section 3.1. We are now in a position to define what we mean by saying a compiler is correct. Typically, compiler correctness has historically been portrayed as requiring that the diagram shown in Figure 1 commutes, namely that for all $P \in Prog_1$,

$$encode(semantic_{source}(P)) = semantic_{target}(compile(P)).$$

This is an intuitively pleasing notion, reflecting the idea that the target representation of the execution of the source program should be the same as that given by executing the target program. However, this definition over-specifies any non-canonical state elements; the correctness of the compilation should obviously not depend on the initial values of the non-canonical elements. To resolve this problem, we use maps that *abstract* from target to source systems (instead of maps that represent source by target systems).

Our notion of correctness requires that we have a homomorphism

$$\Phi : (Prog_2 \times State_2 \times Time_2) \rightarrow (Prog_1 \times State_1 \times Time_1),$$

such that $\Phi(P_2, \sigma_2, t_2)$ gives a configuration of the source system that is the abstraction of the target configuration $(P_2, \sigma_2, t_2) \in Prog_2 \times State_2 \times Time_2$. We shall find it useful to consider

$$\Phi = (c, r, \lambda)$$

in terms of its constituent functions

$$\begin{aligned} c &: (Prog_2 \times State_2 \times Time_2) \rightarrow Prog_1 \\ r &: (Prog_2 \times State_2 \times Time_2) \rightarrow State_1 \\ \lambda &: (Prog_2 \times State_2 \times Time_2) \rightarrow Time_1 \end{aligned}$$

such that $c(P_2, \sigma_2, t_2)$, $r(P_2, \sigma_2, t_2)$ and $\lambda(P_2, \sigma_2, t_2)$ give, respectively, the source program, state and time that correspond to the execution in the target model of the compiled program $P_2 \in Prog_2$ on the initial state $\sigma_2 \in State_2$ for t_2 cycles of time.

We also need to be able to define the set

$$Init_2 \subseteq Prog_2 \times State_2 \times Time_2$$

of target configurations which are correctly initialised. Then, for any $(P_2, \sigma_2, t_2) \in Init_2$, our definition of correctness requires that $\Phi(Exec_2(P_2, \sigma_2, t_2)) = Exec_1(\Phi(P_2, \sigma_2, t_2))$.

Our definition of correctness requires that the semantics of the source and target programs are equivalent under the homomorphism Φ . We can now use information about the structure of AOS to help structure our proof of correctness. In executing a program on an initial state, we generate a sequence of states from the sequence of atomic programs that are determined by the functions $First_i$ and $Rest_i$ for $i = 1, 2$. As we have structured the set $Prog_2$ of target programs to correspond to that of the set $Prog_1$ of source programs, we should be able to apply the functions $First_1$ and $Rest_1$ to $Prog_2$. Furthermore, they should behave in the same manner on the structured target programs as they do on the source programs. Then, providing the behaviour of the source and target atomic programs is equivalent, the behaviour of any source and target programs should also be equivalent. We examine this method of proving correctness in more detail in Section 3.3.

3.2.1 Programs To express the correctness of a compiler $\bar{c} : Prog_1 \rightarrow Prog_2$, we use a function $c : Prog_2 \times State_2 \times Time_2 \rightarrow Prog_1$ to relate programs. Although we can model the compiler \bar{c} with a homomorphism, the abstraction mapping c is more complex, as it may be state- and time-dependent.

3.2.2 States With our model of AOS, we express the behaviour of both the source and target systems as a sequence of states. We shall say that these two sequences are equivalent if:

- (i) we have some method of determining which target state we should compare with which source state at each tick of the source clock; and
- (ii) we can determine that each such pair of source and target states are equivalent.

The problem identified in case (ii) is dependent on how we model the sets of source and target states. We can extract a general principle in the case of (i) though; we picture a typical situation in Figure 3. We can use a retiming map to determine which source and target states we are meant to compare

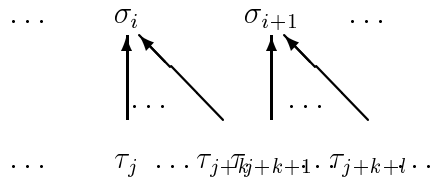


Figure 3: Relating a stream of target states to a stream of source states

at any moment in time:

3.2.3 Retimings We need to determine which target cycles (and hence states) we want to compare with each source cycle tick. We perform this task with a program- and state-dependent retiming function

$$\lambda : Prog_2 \times State_2 \times Time_2 \rightarrow Time_1$$

such that $\lambda(P_2, \sigma_2, t_2)$ gives the time in the source model that corresponds with the time $t_2 \in Time_2$ in the target model that we allow for the execution of the program $P_2 \in Prog_2$ on the state $\sigma_2 \in State_2$.

We can define the function λ in terms of a surjective monotonic map $\hat{\lambda} : Time_2 \rightarrow Time_1$ that relates the faster target clock $Time_2$ to the slower, more abstract source clock $Time_1$.

3.2.4 Initialisation We only want to compare source systems with target systems that have been correctly initialised. This is not a problem in models of correctness where we map down from source to target systems, as such a map automatically determines a correctly initialised target system. However, this only gives us *one* correctly configured target system; with our more general map up from target to source systems, we can choose any target system, providing it has been correctly initialised. We require that the values of the canonical elements of the state have been set to some correct values, and that we have been allocated the precise number of cycles that correspond to the source clock.

3.3 Proving Correctness

To prove that a compiler is correct, we wish to establish that

$$\Phi(Exec_2(P_2, \sigma_2, t_2)) = Exec_1(\Phi(P_2, \sigma_2, t_2))$$

for any correctly initialised system $(P_2, \sigma_2, t_2) \in Init_2 \subseteq Prog_2 \times State_2 \times Time_2$. Our proof method is based around showing that the compiler is correct over a period of one cycle of source time. There are four key steps involved in proving a compiler correct:

3.3.1 Compiler Correctness Conditions Let $k_2 \in Time_2$ be the least amount of cycles in the target clock such that $\lambda(P_2, \sigma_2, k_2) = 1$.

- (i) We need to establish that the program that remains after executing a source program and its compiled version for one step of source time on equivalent states, are equivalent:

$$c(Rest_2^T(P_2, \sigma_2, k_2), Comp_2(P_2, \sigma_2, k_2), t_2) = Rest_1^T(c(P_2, \sigma_2, t_2 + k_2), r(P_2, \sigma_2, t_2 + k_2), 1)$$

- (ii) Similarly, we need to establish that the program that remains after executing a source program and its compiled version for one step of time on equivalent states, produce equivalent states when executed:

$$r(Rest_2^T(P_2, \sigma_2, k_2), Comp_2(P_2, \sigma_2, k_2), t_2) = Comp_1(c(P_2, \sigma_2, t_2 + k_2), r(P_2, \sigma_2, t_2 + k_2), 1)$$

- (iii) We need to show that we can split periods of target time into a sequence of unit source cycles, according to the structure of the programs we are executing:

$$\lambda(P_2, \sigma_2, t_2 + k_2) = \lambda(Rest_2^T(P_2, \sigma_2, k_2), Comp_2(P_2, \sigma_2, k_2), t_2) + 1.$$

- (iv) Finally, we need to determine that executing the target program for one cycle of time will yield a system that is correctly configured for the remaining execution:

$$\Phi(Rest_2^T(P_2, \sigma_2, k_2), Comp_2(P_2, \sigma_2, k_2), t_2) \in Init_2.$$

These four conditions are necessary and sufficient to prove a compiler correct:

3.3.2 Theorem (One Step) *If the Compiler Correctness Conditions 3.3.1 are all satisfied, the following are equivalent*

- (i) for any $(P_2, \sigma_2, k_2) \in \text{Init}_2$, where $k_2 \in \text{Time}_2$ is the least value such that $\lambda(P_2, \sigma_2, k_2) = 1$, $\Phi(\text{Exec}_2(P_2, \sigma_2, k_2)) = \text{Exec}_1(\Phi(P_2, \sigma_2, k_2))$; and
- (ii) for any $(P_2, \sigma_2, t_2) \in \text{Init}_2$, $\Phi(\text{Exec}_2(P_2, \sigma_2, t_2)) = \text{Exec}_1(\Phi(P_2, \sigma_2, t_2))$.

Proof. (i) \Rightarrow (ii) Follows by induction on time, using the Compiler Correctness Conditions 3.3.1.

(ii) \Rightarrow (i) Trivial. □

3.4 Example

We sketch the proof of the correctness of a compiler $\bar{c} : \text{Prog}_1 \rightarrow \text{Prog}_2$ from the set Prog_1 of **while** programs to the set Prog_2 of constructed target URM programs; full details can be found in Stephenson [1996].

3.4.1 Structuring URM programs As explained in Section 3.1.1, we first need to develop program-constructing operations on the basic URM programs of Section 2.2.4 that will enable us to define the set Prog_2 of target programs. We design operations that enable us to either combine two individual URM programs into one, or else to add an individual URM instruction to an existing program. We can then construct the set of target programs that consist of:

atomic programs that mimic identity and assignment statements; and

program-forming operations that mimic sequencing, conditional branching and iteration.

3.4.2 Structuring URM States We divide the set of URM registers into:

variable-storing registers which store values that correspond to those in **while** states;

canonical values consisting of the program counter register; and

non-canonical values consisting of all the remaining registers.

3.4.3 Relating Components We now need to define the component functions c , r and λ of the abstraction homomorphism Φ that describe how we are to relate the source and target systems. For this example,

- (i) we use a state-dependent function $c : \text{Prog}_2 \times \text{State}_2 \rightarrow \text{Prog}_1$ because we require the label of the first instruction to tally with the value held in the program counter of the state;
- (ii) we use a function $r : \text{State}_2 \rightarrow \text{State}_1$ which is derived from homomorphisms r_{val} and r_{var} that relate the underlying data types and the locations that we use to store data, respectively;
- (iii) we use a program- and state-dependent retiming map $\lambda : \text{Prog}_2 \times \text{State}_2 \times \text{Time}_2 \rightarrow \text{Time}_1$ because we have conditional tests in **while** programs which can be state-dependent.

Finally, we need to consider the initialisation criteria. We need to have a URM program and a state where the value of the program counter has been correctly set. We also need to check that we have been allocated the correct number of cycles in the target clock.

3.4.4 Proving Correctness We prove the compiler correct by applying the methodology outlined in Section 3. We prove each case (CC1)–(CC4) by using structural induction on programs; each sub-case is proved by applying equational reasoning, coupled with induction on the natural numbers.

4 Conclusions

We have given an equational method of defining an operational semantics which we structure algebraically. We then considered how we could use this technique of AOS to describe firstly how any compiler between two languages could be structured; and secondly how the correctness of any such compiler could be proved correct. We noted that our approach differs from, and has advantages over, other approaches advocated in the literature. Finally, we illustrated our strategy with an example of a compiler from **while** programs to URM programs.

Acknowledgments

I wish to thank Professor J V Tucker who supervised my PhD studies; the material in this paper is based on Stephenson [1996], where further details and examples can be found.

References

Bergstra and Tucker [1982]

J A Bergstra and J V Tucker. The Completeness of the Algebraic Specification Methods for Computable Data Types. *Information and Control*, 54:186–200, 1982.

Bergstra and Tucker [1996]

J A Bergstra and J V Tucker. Equational Specifications, Complete Term Rewriting Systems, and Computable and Semicomputable Algebras. *Journal of the Association for Computing Machinery*, 42(6):1194–1230, 1996.

Broy *et al.* [1981]

M Broy, W Dosch, B Möller, and M Wirsing. GOTOS - A study in the Algebraic Specification of Programming Languages (Extended Abstract). In W Brauer, editor, *Proceedings of the 11 Jahrestagung der Gesellschaft für Informatik*, number 50 in Informatik Fachberichte, pages 109–121, Berlin, 1981. Springer-Verlag.

Burstall and Landin [1969]

R M Burstall and P J Landin. Programs and Their Proofs: An Algebraic Approach. In B Meltzer and D Michie, editors, *Machine Intelligence*, volume 4, pages 17–43. Edinburgh University Press, Edinburgh, 1969.

Courcelle and Franchi-Zanettacci [1982]

B Courcelle and P Franchi-Zanettacci. Attribute Grammars and Recursive Program Schemes. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

Goguen and Malcolm [1996]

J A Goguen and G Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

Goguen *et al.* [1977]

J A Goguen, J W Thatcher, E G Wagner, and J B Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24:68–95, 1977.

Morris [1973]

F L Morris. Advice on Structuring Compilers and Proving them Correct. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 144–152. ACM Press, New York, 1973.

Nivat and Reynolds [1985]

M Nivat and J C Reynolds. *Algebraic Methods in Semantics*. Cambridge University Press, Cambridge, 1985.

Plotkin [1981]

G D Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Århus University, Århus, Denmark, September 1981.

Plotkin [1983]

G D Plotkin. An Operational Semantics for CSP. In D Bjørner, editor, *Formal Descriptions of Programming Concepts II*, pages 199–255. North-Holland, Amsterdam, 1983.

Rus [1971]

T Rus. ΣS -Algebra of a Formal Language. *Société des Sciences Mathématiques de la République Socialiste de Roumaine Bulletin Mathématique*, 15(2):227–235, 1971.

Shepherdson and Sturgis [1963]

J C Shepherdson and H E Sturgis. Computability of Recursive Functions. *Journal of the ACM*, 10:217–255, 1963.

Stephenson [1996]

K Stephenson. *An Algebraic Approach to Syntax, Semantics and Compilation*. PhD thesis, Department of Computer Science, University of Wales Swansea, 1996.

Thatcher *et al.* [1981]

J W Thatcher, E G Wagner, and J B Wright. More Advice on Structuring Compilers and Proving them Correct. *Theoretical Computer Science*, 15:223–249, 1981.

Tucker and Zucker [1988]

J V Tucker and J I Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*. North Holland, 1988.