Towards a Provably Correct Compiler for OBJ3*

Lutz H. Hamel and Joseph A. Goguen

Programming Research Group
Oxford University Computing Laboratory
Oxford, OX1 3QD U.K.

Abstract. Abstract machines have proved very successful in the implementation of very high level logic and functional programming languages; in particular, we have the G-machine for functional programming languages and the WAM for Prolog. In this paper we develop an abstract machine appropriate for the implementation of algebraic specification languages. We then employ general algebra techniques and initiality properties to prove correctness of the translation from equational rewrite rules to the abstract machine code. The correctness proof itself has been automated using the OBJ3 system.

Keywords: abstract machines, algebraic specification, compilers, rewrite rules.

1 Introduction

We are interested in implementing algebraic specification languages such as OBJ [7] and ACT-ONE [5] on an abstract machine. In this approach the terms of the source language are translated into abstract machine code which is powerful enough to handle difficult computational aspects of the source language fairly easily, but is also low level enough that it can be executed very efficiently on conventional computer architectures. This was first proposed for and applied to the Prolog logic programming language by Warren [17] in the early 1980's, and in more recent years it has been applied successfully to functional languages, for example in the G-machine [11]. In this paper we take a similar approach and translate the equations of specifications into code on our TRIM abstract machine (**T**iny **R**ewrite **I**nstruction **M**achine).

* The research reported in this paper has been supported in part by the Science and Engineering Research Council, the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems COrrectness and REusability). and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), Fujitsu Laboratories Limited, and the Information Technology Promotion Agency, Japan, as part of the R & D of Basic Technology for Future Industries "New Models for Software Architecture" project sponsored by NEDO (New Energy and Industrial Technology Development Organization).

It is not enough, however, to define the translation from equations to abstract machine code; we also have to worry about the correctness of the translation. In order to show the correctness of the translation we construct a proof based on the algebraic methods first put forward by Morris in [14] and further developed in Thatcher *et al.* [16]. The proof hinges on the fact that a context-free grammar for a source language defines an initial Σ-algebra [8] and the essential proof obligation is to show that a particular diagram in the category of Σ-algebras commutes.

The proof presented here is the correctness proof of a compiler specification. Clearly, if this specification is written in an executable specification language we immediately obtain a compiler from the specification. However, in general another proof step is required, which is to show that the compiler correctly implements the specification [4].

In order to keep the proof manageable we limit ourselves to single sorted equations. For similar reasons we only consider specifications with a single equation. We keep category theoretic ideas to a minimum save a passing comment here and there in order to make the material accessible to the widest audience possible.

The rest of the paper is structured as follows. Section 2 provides an overview of the general proof strategy and introduces some fundamental notions needed. Section 3 defines the basic mathematical building blocks. In Section 4 we define our source language, single sorted equational rewrite rules, in a mathematically rigorous way. Section 5 introduces our abstract machine. The translation from equations to abstract machine code is presented in Section 6. In Section 7 we show that the translation is indeed correct. Section 8 ends the paper with a summary and with pointers to possible directions for future research and some related work.

2 The Structure of the Proof

The intuitive notion of correctness in our proof is the preservation of the semantics of the source language in the target semantics. To be more precise, given a translation of the source to the target language we have to show that there exists a homomorphism from the source semantics to the target semantics. If we can establish such a homomorphism, then initiality will guarantee the commutativity of the correctness diagram.

We adapt Morris' approach [14] and demonstrate the correctness of the compiler specification by making the following diagram commute:

$$\begin{tikzcd}
M_{\Sigma^\sharp} \arrow[r, "\epsilon"] & dU_{\Sigma'} & & & U_{\Sigma'} \\
 & & {} & {} \arrow[l, mapsto, "d"] & \\
T_{\Sigma^\sharp} \arrow[uu, "\theta"] \arrow[r, "\gamma"'] & dT_{\Sigma'} \arrow[uu, "d\psi"] & & & T_{\Sigma'} \arrow[uu, "\psi"']
\end{tikzcd}$$

The rest of this section will explain the notation used in this diagram. We first define a signature $\Sigma^\sharp$ which is an extension of some user defined signature $\Sigma(X)$, where Σ is a set of operation symbols and X is a set of variables disjoint from Σ. By writing $\Sigma(X)$ we wish to indicate that we treat the variables $x \in X$ as constants – as nullary operators in $\Sigma(X)$ – and that we will assign semantics to these operators as we will for all other operators in the signature. The term algebra of our source language forms the lower lefthand corner, and is the initial $\Sigma^\sharp$-algebra. We then define an appropriate $\Sigma^\sharp$-algebra, $M_{\Sigma^\sharp}$, which we view as the semantics for our source language. By initiality the homomorphism θ, which interprets terms of the source language in this semantics, exists and is unique. This is developed in more detail in Sect. 4.

Next, we denote the term algebra of the abstract machine language by $T_{\Sigma'}$. As with our source language, we define an appropriate Σ'-algebra, $U_{\Sigma'}$, as the semantics for this machine language. Again, by initiality in the category of Σ'-algebras the homomorphism ψ exists and is unique. For a more concrete discussion of this see Sect. 5.

In order to relate the two languages to each other and to obtain the righthand side of our diagram we use the concept of a *derivor* developed in [10]:

Definition 1. Let Σ be an S-sorted signature, and let Σ' be an S'-sorted signature. Then a derivor $d : \Sigma \to \Sigma'$ is a function $f : S \to S'$ and a family $d_{w,s} : \Sigma_{w,s} \to (T_{\Sigma'})_{f(w),f(s)}$, where $w = s_1 \dots s_n$ and $f(w) = f(s_1 \dots s_n) = f(s_1) \dots f(s_n)$ and where $(T_{\Sigma'})_{f(w),f(s)}$ denotes the set of all Σ'-terms of sort $f(s)$ using variables $\{y_1, \dots, y_n\}$ with y_i of sort $f(s_i)$. (Each operation symbol $\sigma \in \Sigma_{w,s}$ is expressed using a derived operation $d_{w,s}(\sigma)$ of the appropriate arity.)

Now let B be a Σ'-algebra. Then the *d-derived algebra dB* of B is the Σ-algebra with carriers $(dB)_s = B_{f(s)}$, for all $s \in S$, and with σ_{dB_s} (for $\sigma \in \Sigma_{w,s}$) defined to be $(d(\sigma))_{B_{f(s)}}$, the derived operator of the Σ'-term $d(\sigma)$.

The next step is to define an appropriate derivor from $\Sigma^\sharp$ to Σ'. Then we can fill in the righthand side of our diagram. The algebras $dT_{\Sigma'}$ and $dU_{\Sigma'}$ are immediate from the above definition. That the morphism $d\psi$ between them exists is not so obvious. However, the following proposition is helpful and we state it here without proof.

Proposition 2. *Let A and B be Σ'-algebras and let $h : A \to B$ be a Σ'-homomorphism between them. Then any derivor $d : \Sigma \to \Sigma'$ induces a Σ-homomorphism $dh : dA \to dB$, with $dh_s(x) = h_{f(s)}(x)$ for $x \in dA_s$ and $s \in S$.*

Since we know that ψ exists, we also know that the derivor d induces the homomorphism $d\psi$. By initiality we also have the homomorphism γ from $T_{\Sigma^\sharp}$ to $dT_{\Sigma'}$. It is this homomorphism which we view as the compiler specification. Notice that we now have defined the four corners and three sides of the diagram. To show that the diagram commutes it suffices to show that the homomorphism ϵ exists, since $\theta \, ; \, \epsilon = \gamma \, ; \, d\psi$ due to *uniqueness*. That is, the diagram *necessarily* commutes provided the homomorphisms exist at all. This material is developed in Sect. 6 and Sect. 7.

3 Tuples, Products, Projections and other Things

Throughout this paper we rely heavily on function composition, products and projection functions. In this section we define the basic notions we use during the development of the compiler correctness proof. This closely follows the development in [16].

Definition 3. Given two functions $f_i : A \rightarrow B$, we define the *source tuple*, $(f_1, f_2) : A \times [2] \rightarrow B$, by

$$(f_1, f_2)\langle a, i\rangle = f_i(a) \ ,$$

where [2] denotes the set $\{\mathbf{1}, \mathbf{2}\}$. Also, given two functions $f_i : A \rightarrow B_i$, we define the *target tuple*, $[f_1, f_2] : A \rightarrow B_1 \times B_2$, by

$$[f_1, f_2](a) = \langle f_1(a), f_2(a)\rangle \ ,$$

and given the two functions $f_i : A_i \rightarrow B_i$, we define the *product map*, $f_1 \times f_2 : A_1 \times A_2 \rightarrow B_1 \times B_2$, by

$$(f_1 \times f_2)\langle a_1, a_2\rangle = \langle f_1(a_1), f_2(a_2)\rangle \ .$$

The *projection* $\pi_i : A_1 \times \ldots \times A_n \rightarrow A_i$ takes the tuple $\langle a_1, \ldots, a_n\rangle$ to a_i.

The *composition* of two functions, $f : A \rightarrow B$ and $g : B \rightarrow C$, is denoted in the diagrammatic way by $f \ ; \ g : A \rightarrow C$. Finally, we write $f : A \rightarrow B$ even if the function is partial and we also assume that all functions are strict and note that strictness is preserved by the above operations. We also let $[A \rightarrow B]$ denote the set of possibly partial mappings from A to B.

4 The Source Language

Algebraic specification languages have a fairly efficient operational semantics by viewing the equations of the specification as directed rewrite rules [9, 12]. It is this operational view of the equations which allows the user to study the runtime behavior of a specification before committing to any kind of implementation.

Therefore, rather than equations, our source language consists of single sorted equational rewrite rules of the form $t \Rightarrow t'$ where $t, t' \in T_{\Sigma(X)}$ and where $T_{\Sigma(X)}$ denotes the set of terms over some user defined signature $\Sigma(X)$. We also assume that the rewrite rules are left linear and that any variable which appears on the righthand side of a rule also appears on the lefthand side of that rule.

4.1 The Source Syntax

To make the syntax more precise, rewrite rules in our source language have a form like the following:

$$(1 * v) + w \Rightarrow w + v$$

where $v, w \in X$ and $*, +, 1 \in \Sigma$. Here of course, $*$ and $+$ are binary operators, 1 is a constant operator and v and w are variables.

We summarize the general shape of our source language with the following grammar:

$$\begin{array}{lll}
eq & ::= & l\_term \;\Rightarrow\; r\_term \\
 & & \\
l\_term & ::= & \sigma(l\_term_1, \ldots, l\_term_n) \\
 & \mid & \kappa \\
 & \mid & x \\
 & & \\
r\_term & ::= & \sigma(r\_term_1, \ldots, r\_term_n) \\
 & \mid & \kappa \\
 & \mid & x
\end{array}$$

where $x \in X$ and $\kappa, \sigma \in \Sigma$. Thus, in the grammar x represents variables such as v or w. The symbol κ stands for constant operators such as 1 and σ represents operators of arbitrary arities such as the the binary operators $*$ and $+$. Without loss of generality we will assume for the remainder of the paper that σ is a binary operator.

Making use of the fact that a context-free grammar defines a signature [8], we denote the signature defined by this context-free grammar with $\Sigma^\sharp$. We may then view the context free syntax of the rewrite rules as a term algebra $T_{\Sigma^\sharp}$ with the carriers $T_{\Sigma^\sharp l}$, $T_{\Sigma^\sharp r}$ and $T_{\Sigma^\sharp eq}$ representing terms on the lefthand side, righthand side, and equations, respectively.

Definition 4. The inductive definition of the $\Sigma^\sharp$-term algebra $T_{\Sigma^\sharp}$ is as follows:

$$\begin{array}{r}
\kappa_l \in T_{\Sigma^\sharp} \\
x_l \in T_{\Sigma^\sharp} \\
\sigma_l(t_{l1}, t_{l2}) \in T_{\Sigma^\sharp} \\
\kappa_r \in T_{\Sigma^\sharp} \\
x_r \in T_{\Sigma^\sharp} \\
\sigma_r(t_{r1}, t_{r2}) \in T_{\Sigma^\sharp} \\
\Rightarrow_{eq} (t_l, t_r) \in T_{\Sigma^\sharp}
\end{array}$$

where $t_l, t_{l1}, t_{l2} \in T_{\Sigma^\sharp l}$ and $t_r, t_{r1}, t_{r2} \in T_{\Sigma^\sharp r}$.

The sort subscripts l, r and eq are carried through as a reminder of where exactly these terms appear. Even though this inductive definition of the $T_{\Sigma^\sharp}$ algebra seems somewhat redundant, we give it here explicitly to show the $\Sigma^\sharp$ structure of this algebra, which will constantly reappear throughout the proof.

4.2 The Source Semantics

Our semantics for rewrite rules is a $\Sigma^\sharp$-algebra representing an abstract interpreter for the rewrite rules. A state in this interpreter is a tuple $\langle t, s, e\rangle$, where $t \in T_\Sigma$ is the term to be reduced, $s \in Stk_{Addr}$ is a stack of term addresses pointing into the Σ-term, and $e \in Env_M$ is a mapping of user-variables to Σ-terms. Env_M is defined to be the set of all possible mappings from variables to Σ-terms, $[X \rightarrow T_\Sigma]$.

Viewing equations as rewrite rules, as we do, allows us to design a semantics in which the lefthand side of the rules attempt to match a piece of the input term, thus effectively establishing an assignment to the variables ocurring in the lefthand side. Once a lefthand side is matched, the righthand side of that rule and the assignments to the variables are used to construct a new term which replaces the input term.

We construct the algebra from a number of primitive operations using function composition, tupling, and projections to build the operations in $\Sigma^\sharp$. In particular we use the following functions: $match_\kappa : T_\Sigma \times Stk_{Addr} \rightarrow T_\Sigma \times Stk_{Addr}$, $match_x : T_\Sigma \times Stk_{Addr} \rightarrow T_\Sigma \times Stk_{Addr} \times T_\Sigma$, $match_\sigma : T_\Sigma \times Stk_{Addr} \rightarrow T_\Sigma \times Stk_{Addr}$, $assign_x : Env_M \times T_\Sigma \rightarrow Env_M$, $fetch_x : Env_M \rightarrow Env_M \times T_\Sigma$, $build_\sigma : T_\Sigma \times T_\Sigma \rightarrow T_\Sigma$, and $apply : T_\Sigma \times Stk_{Addr} \times T_\Sigma \rightarrow T_\Sigma \times Stk_{Addr}$.

In order to aid the exposition, we now give the set theoretic definitions of these functions. However, these are not used in the actual proof, which instead uses a direct equational axiomatization of the target language.

$$
\begin{aligned}
match_\kappa\langle t, a \bullet \rho\rangle &= \begin{cases} \langle t, \rho\rangle \text{ if } t|_a = \kappa \\ \bot \quad \text{otherwise} \end{cases} \\
match_x\langle t, a \bullet \rho\rangle &= \langle t, \rho, t|_a\rangle \\
match_\sigma\langle t, a \bullet \rho\rangle &= \begin{cases} \langle t, a_{t1} \bullet a_{t2} \bullet \rho\rangle \text{ if } t|_a = \sigma(t1, t2) \\ \bot \qquad\qquad\qquad \text{otherwise} \end{cases} \\
assign_x\langle e, t\rangle z &= \begin{cases} t & \text{if } z = x \\ e(z) & \text{if } z \neq x \end{cases} \\
fetch_x(e) &= \langle e, e(x)\rangle \\
build_\sigma\langle t_2, t_1\rangle &= \sigma(t_1, t_2) \\
apply\langle t_1, \rho, t_2\rangle &= \langle t_2, a_{t2} \bullet []\rangle
\end{aligned}
$$

where $t, t_1, t_2 \in T_\Sigma$, and $t|_a$ denotes the term t restricted to the address a, in other words a subterm of t with its root at address a. a_{t1} and a_{t2} denote the root addresses of the terms t_1 and t_2, respectively. We also have $\rho \in Stk_{Addr}$ and we let $[]$ denote the empty stack and $\bullet$ denote the *push* onto the stack. Finally, $e \in Env_M$.

Some intuitive explanation of these operations is in order. Concrete interpreters are usually built recursively, using the function activation records on the stack to keep track of which part of the input term needs to be evaluated next. In our abstract interpreter the stack of addresses can be viewed as a model for the runtime stack of a concrete interpreter. Thus, the top of the stack always points to the part of the term which needs to be evaluated next.

For example, $match_\kappa$ checks whether the top of the stack does indeed point to a constant term equal to κ. If so, it will pop the address off the stack and return. If not, it will just fail. In our case failure is signaled by returning $\perp$. Similarly for $match_x$, but instead of matching against a constant term, $match_x$ returns the input term restricted to the address on the top of the stack.

A somewhat more complicated operation is matching against an operation with subterms such as $match_\sigma$. Here we check if the operation pointed to by the top of the stack is equal to σ. If so, we pop the top of the stack and push the addresses of the two subterms onto the stack. If not, we return failure.

The operations $assign_x$ and $fetch_x$ are self-explanatory and are simple manipulations of the environment. What is noteworthy about the operation $build_\sigma$ is that it accepts the subterms of the σ operator in reverse order. The operation $apply$ replaces the input term t_1 with the newly constructed term t_2 and initializes the stack to the address of the root of term t_2.

Definition 5. The source semantics is given by the $\Sigma^\sharp$-algebra $M_{\Sigma^\sharp}$ with the carriers:

$$\begin{aligned} M_l &= [T_\Sigma \times Stk_{Addr} \times Env_M \to T_\Sigma \times Stk_{Addr} \times Env_M] \\ M_r &= [T_\Sigma \times Stk_{Addr} \times Env_M \to T_\Sigma \times Stk_{Addr} \times Env_M \times T_\Sigma] \\ M_{eq} &= [T_\Sigma \times Stk_{Addr} \times Env_M \to T_\Sigma \times Stk_{Addr} \times Env_M] \end{aligned}$$

and operations:

$$\begin{aligned} \kappa_l^M &= match_\kappa \times 1_{Env_M} \\ x_l^M &= (match_x \times 1_{Env_M}) \ ; \ [\pi_1, \pi_2, \pi_4, \pi_3] \ ; \ (1_{T_\Sigma \times Stk_{Addr}} \times assign_x) \\ \sigma_l^M(\alpha_1, \alpha_2) &= (match_\sigma \times 1_{Env_M}) \ ; \ \alpha_1 \ ; \ \alpha_2 \\ \kappa_r^M &= 1_{T_\Sigma \times Stk_{Addr} \times Env_M} \times \kappa_{T_\Sigma} \\ x_r^M &= 1_{T_\Sigma \times Stk_{Addr}} \times fetch_x \\ \sigma_r^M(\beta_1, \beta_2) &= \beta_1 \ ; \ (\beta_2 \times 1_{T_\Sigma}) \ ; \ (1_{T_\Sigma \times Stk_{Addr} \times Env_M} \times build_\sigma) \\ \Rightarrow_{eq}^M (\alpha, \beta) &= \alpha \ ; \ \beta \ ; \ [\pi_1, \pi_2, \pi_4, \pi_3] \ ; \ (apply \times 1_{Env_M}) \end{aligned}$$

where $\alpha, \alpha_1, \alpha_2 \in M_l$ and $\beta, \beta_1, \beta_2 \in M_r$.

In this algebra each operator is constructed from our primitive functions and the underlying operations defined in Sect. 3. For example, the operation κ_l^M attempts to match the constant κ in the input term but leaves the environment unchanged.

5 The Abstract Machine: TRIM

Our TRIM machine (**T**iny **R**ewrite **I**nstruction **M**achine) is a stack based abstract machine. Each instruction acts on a state which consists of a stack and an environment for variable bindings.

5.1 The Syntax

The syntax of our abstract machine is given by the following context-free grammar:

$$\begin{array}{lll} \mathit{program} & ::= & \mathit{program} \fatsemi \mathit{instruction} \\ & \mid & \mathit{instruction} \\ & & \\ \mathit{instruction} & ::= & \mathit{MATCH}\ a \\ & \mid & \mathit{ENTER} \\ & \mid & \mathit{LEAVE} \\ & \mid & \mathit{SAVE}\ x \\ & \mid & \mathit{GET}\ x \\ & \mid & \mathit{PUSH}\ a \\ & \mid & \mathit{POP} \\ & \mid & \mathit{FLIP}\ \sigma \end{array}$$

where $x \in X$ and $a, \sigma \in \Sigma$.

As in most cases, the instruction set of the abstract machine reflects the computational characteristics of the source language. For example, the $\mathit{MATCH} a$ instruction attempts to match the operator symbol a in the input term. Clearly, this instruction can be used to implement operator symbols appearing on the lefthand side of our equational rewrite rules.

However, abstract machines tend to act on much more concrete data structures than their respective source languages, so that they can be efficiently implemented on conventional computer architectures. In our case, these data structures are a stack and an environment for variable bindings, probably implemented as some sort of hashed symbol table.

In our proof we denote the term algebra generated by the above grammar by $T_{\Sigma'}$ and note that its carriers are $T_{\Sigma' \mathit{program}}$ and $T_{\Sigma' \mathit{instruction}}$. This algebra is of course initial in the category of Σ'-algebras.

5.2 The Machine Semantics

Each instruction of the abstract machine acts on a state which is a pair $\langle s', e' \rangle$, where $s' \in \mathit{Stk}$ is assumed to be a stack of well formed Σ-terms, and $e' \in \mathit{Env}$ is a mapping from variables to Val-items, $X \longrightarrow \mathit{Val}$. Val is the stack representation of a Σ-term which can be pushed and popped in an atomic operation.

Similar to the semantics of our source language, we construct an algebra from the various primitive functions and operations defined in Sect. 3. In particular we make use of the following functions, where we assume that $x \in X$ and $a, \sigma \in \Sigma$: $\mathit{peek}_a : \mathit{Stk} \longrightarrow \mathit{Stk} \times [2]$, $\mathit{pop} : \mathit{Stk} \longrightarrow \mathit{Stk}$, $\mathit{push}_a : \mathit{Stk} \longrightarrow \mathit{Stk}$, $\mathit{popval} : \mathit{Stk} \longrightarrow \mathit{Stk} \times \mathit{Val}$, $\mathit{pushval} : \mathit{Stk} \times \mathit{Val} \longrightarrow \mathit{Stk}$, $\mathit{flip}_\sigma : \mathit{Stk} \longrightarrow \mathit{Stk}$, $\mathit{bind}_x : \mathit{Env} \times \mathit{Val} \longrightarrow \mathit{Env}$, and $\mathit{retrieve}_x : \mathit{Env} \longrightarrow \mathit{Env} \times \mathit{Val}$.

As with the source semantics we give the set theoretic definitions of the functions only to make the exposition easier to understand. The actual proof uses axioms which these functions satisfy. The set theoretic definitions of these functions are:

$$\begin{aligned}
peek_a(z \bullet \tau) &= \begin{cases} \langle z \bullet \tau, \mathbf{1}\rangle & \text{if } z = a \\ \langle z \bullet \tau, \mathbf{2}\rangle & \text{otherwise} \end{cases} \\
pop(z \bullet \tau) &= \tau \\
push_a(\tau) &= a \bullet \tau \\
popval(v \bullet \tau) &= \langle \tau, v\rangle \\
pushval\langle \tau, v\rangle &= v \bullet \tau \\
flip_\sigma(t_1 \bullet t_2 \bullet \tau) &= t_2 \bullet t_1 \bullet \tau \\
bind_x\langle e, v\rangle y &= \begin{cases} v & \text{if } y = x \\ e(y) & \text{if } y \neq x \end{cases} \\
retrieve_x(e) &= \langle e, e(x)\rangle
\end{aligned}$$

where $\tau \in Stk$, $v, t_1, t_2 \in Val$, $z \in \Sigma$, $y \in X$, and $e \in Env$.

We should mention that the $\bullet$-operation in this case is overloaded. It actually stands for two operations; first, pushing a single operator symbol from Σ onto the stack; second, pushing a complete Σ-term represented by a Val item onto the stack. This lets us construct whole Σ-terms from single operator symbols in Σ.

Definition 6. Viewing each operator in Σ' as an operation $Stk \times Env \longrightarrow Stk \times Env$, we define the machine semantics as the Σ'-algebra $U_{\Sigma'}$ with the carriers:

$$\begin{aligned}
U_{\Sigma'instruction} &: [Stk \times Env \longrightarrow Stk \times Env] \\
U_{\Sigma'program} &: [Stk \times Env \longrightarrow Stk \times Env]
\end{aligned}$$

and operations:

$$\begin{aligned}
\mu \ ⨾^U \ \nu &= \mu \ ; \ \nu \\
MATCH^U \ a &= (peek_a \times 1_{Env}); ((pop, \bot) \times 1_{Env}) \\
ENTER^U &= 1_{Stk \times Env} \\
LEAVE^U &= 1_{Stk \times Env} \\
SAVE^U \ x &= (popval \times 1_{Env}); [\pi_1, \pi_3, \pi_2]; (1_{Stk} \times bind_x) \\
GET^U \ x &= (1_{Stk} \times retrieve_x); [\pi_1, \pi_3, \pi_2]; (pushval \times 1_{Env}) \\
PUSH^U \ a &= push_a \times 1_{Env} \\
POP^U &= pop \times 1_{Env} \\
FLIP^U \ \sigma &= flip_\sigma \times 1_{Env}
\end{aligned}$$

where $\mu \in U_{\Sigma'program}$ and $\nu \in U_{\Sigma'instruction}$.

Some explanatory remarks on the semantics: the operation $MATCH a$ looks at the top of the stack to see if the top of the stack is equal to the operator symbol a. If so, the stack is popped, otherwise a failure is returned. This operation leaves the environment unchanged. Concatenating an instruction at the end of a program is just the functional composition of the program and the instruction viewed as operations. We hope that the remainder of the semantics is fairly self-explanatory.

6 The Target Language

Up to now we have defined the source language and the abstract machine language with their corresponding syntax and semantics. However, we have not mentioned anything about how one language is related to the other. In fact, from the algebraic point of view the term algebras of each of these languages lives in its own separate category and has no relation to the other language at all. Therefore, the next step is to define the translation from the source language to the abstract machine language. As mentioned before, we do that with the help of a derivor. Notice that defining a derivor turns out to be a compiler specification.

A compiler cannot generate arbitrary sequences of instructions in the abstract machine language but only certain combinations of instructions. We call the restricted machine language which the compiler produces the *target language*.

It turns out that this restricted machine language is equivalent to the d-derived term algebra of the original machine language, and that this d-derived algebra is also an algebra in the category in which the source language is the initial algebra. Thus, it now makes sense to talk about homomorphisms from the source language to the target language. In fact, we will investigate the homomorphism γ which effects the translation.

6.1 The Compiler Specification

A derivor has two constituents: a function which maps the sorts from one signature into the sorts of the other signature, and a family which maps the operator symbols of one signature into derived operators of the other signature.

Definition 7. We define the derivor $d : \Sigma^\sharp \rightarrow \Sigma'$ as follows: Let $S^\sharp = \{l, r, eq\}$ and $S' = \{program, instruction\}$, then the function $f : S^\sharp \rightarrow S'$ maps sorts $l, r, eq \in S^\sharp$ into the sort $program \in S'$. The family $d_{w,s} : \Sigma^\sharp_{w,s} \rightarrow (T_{\Sigma'})_{f(w),f(s)}$ is defined by

$$\begin{aligned} d(\kappa_l) &= MATCH\ \kappa \\ d(x_l) &= SAVE\ x \\ d(\sigma_l) &= MATCH\ \sigma \ ⨾\ y_1 \ ⨾\ y_2 \\ d(\kappa_r) &= PUSH\ \kappa \\ d(x_r) &= GET\ x \\ d(\sigma_r) &= y_1 \ ⨾\ y_2 \ ⨾\ FLIP\ \sigma \ ⨾\ PUSH\ \sigma \\ d(\Rightarrow_{eq}) &= ENTER \ ⨾\ y_1 \ ⨾\ y_2 \ ⨾\ LEAVE \end{aligned}$$

where y_1 and y_2 are variables of sort *program*.

6.2 The Target Syntax

Our target language is the restricted abstract machine language the compiler can produce. We are interested in the d-derived term algebra of the algebra $T_{\Sigma'}$. This algebra is a $\Sigma^\sharp$-algebra, that is, the only operation symbols available in this algebra are due to the operators defined by the derivor. From the definition of the derivor d the following is immediate.

Definition 8. The restricted abstract machine language viewed as the $\Sigma^\sharp$-algebra $dT_{\Sigma'}$ with carriers $dT_{\Sigma'_{eq}} = dT_{\Sigma'_l} = dT_{\Sigma'_r} = T_{\Sigma'_{program}}$:

$$\begin{aligned}
\kappa_l^{dT} &= MATCH\ \kappa \\
x_l^{dT} &= SAVE\ x \\
\sigma_l^{dT}(t_1, t_2) &= MATCH\ \sigma \fatsemi t_1 \fatsemi t_2 \\
\kappa_r^{dT} &= PUSH\ \kappa \\
x_r^{dT} &= GET\ x \\
\sigma_r^{dT}(t_1, t_2) &= t_1 \fatsemi t_2 \fatsemi FLIP\ \sigma \fatsemi PUSH\ \sigma \\
\Rightarrow_{eq}^{dT} (t_1, t_2) &= ENTER \fatsemi t_1 \fatsemi t_2 \fatsemi LEAVE
\end{aligned}$$

where $t_1, t_2 \in T_{\Sigma'_{program}}$.

6.3 The Target Semantics

Since we are interested in showing that the diagram in Sect. 2 commutes, we need a $\Sigma^\sharp$-algebra which appropriately represents the semantics of our target language. The best suited algebra is of course the d-derived algebra $dU_{\Sigma'}$ of the algebra $U_{\Sigma'}$, the original semantics of the abstract machine language.

Definition 9. The semantics of our target language is then given by the d-derived algebra $dU_{\Sigma'}$ with carriers $dU_{\Sigma'_{eq}} = dU_{\Sigma'_l} = dU_{\Sigma'_r} = U_{\Sigma'_{program}}$:

$$\begin{aligned}
\kappa_l^{dU} &= (peek_\kappa \times 1_{Env})\ ;\ ((pop, \perp) \times 1_{Env}) \\
x_l^{dU} &= (popval \times 1_{Env})\ ;\ [\pi_1, \pi_3, \pi_2]\ ;\ (1_{Stk} \times bind_x) \\
\sigma_l^{dU}(t_1, t_2) &= (peek_\sigma \times 1_{Env})\ ;\ ((pop, \perp) \times 1_{Env})\ ;\ t_1\ ;\ t_2 \\
\kappa_r^{dU} &= push_\kappa \times 1_{Env} \\
x_r^{dU} &= (1_{Stk} \times retrieve_x)\ ;\ [\pi_1, \pi_3, \pi_2]\ ;\ (pushval \times 1_{Env}) \\
\sigma_r^{dU}(t_1, t_2) &= t_1\ ;\ t_2\ ;\ (flip_\sigma \times 1_{Env})\ ;\ (push_\sigma \times 1_{Env}) \\
\Rightarrow_{eq}^{dU} (t_1, t_2) &= 1_{Stk \times Env}\ ;\ t_1\ ;\ t_2\ ;\ 1_{Stk \times Env}
\end{aligned}$$

where $t_1, t_2 \in U_{\Sigma'_{program}}$.

7 The Diagram Commutes!

As outlined in Sect. 2, we have defined all four corners of the diagram. The morphisms θ, γ and $d\psi$ exist due to initiality and the properties of the derivor d, respectively. What remains is to show that the morphism ϵ exists. We need to show that there exists a mapping from the source semantics to the target semantics which make the diagram commute, i.e., which preserves meaning.

Since the notion of state is different in each of these semantics, we need a way to convert from one representation to the other. In order to accomplish this we postulate the existence of a number of functions:

$$\begin{aligned}
encode_{T_\Sigma} &: T_\Sigma \times Stk_{Addr} \longrightarrow Stk \\
decode_{T_\Sigma} &: Stk \longrightarrow T_\Sigma \times Stk_{Addr} \\
encode_{Env_M} &: Env_M \longrightarrow Env \\
decode_{Env_M} &: Env \longrightarrow Env_M \\
convert &: T_\Sigma \longrightarrow Val \\
unconvert &: Val \longrightarrow T_\Sigma
\end{aligned}$$

Furthermore we require that these functions satisfy the following equations:

$$\begin{aligned}
encode_{T_\Sigma} \; ; \; decode_{T_\Sigma} &= 1_{T_\Sigma \times Stk_{Addr}} \\
decode_{T_\Sigma} \; ; \; encode_{T_\Sigma} &= 1_{Stk} \\
encode_{Env_M} \; ; \; decode_{Env_M} &= 1_{Env_M} \\
decode_{Env_M} \; ; \; encode_{Env_M} &= 1_{Env} \\
convert \; ; \; unconvert &= 1_{T_\Sigma} \\
unconvert \; ; \; convert &= 1_{Val}
\end{aligned}$$

The second equation might be somewhat surprising, but it is a valid requirement, since we assume the stack to only hold well formed Σ-terms which of course makes $decode_{T_\Sigma}$ a total function. We axiomatize the move from one representation with equations like the following:

$$\begin{aligned}
(decode_{Env_M} \times unconvert) \; ; \; assign_x &= bind_x \; ; \; decode_{Env_M} \\
decode_{T_\Sigma} \; ; \; match_\kappa &= peek_\kappa \; ; \; (pop, \perp) \; ; \; decode_{T_\Sigma} \\
(decode_{T_\Sigma} \times unconvert) \; ; \; apply &= pushval \; ; \; decode_{T_\Sigma}
\end{aligned}$$

The actual proof uses about 50 such equations, which all have a broadly similar flavor. We are now in a position to define $\epsilon : M_{\Sigma^\sharp} \longrightarrow dU_{\Sigma'}$ to be a function which takes an operation in the source semantics to an appropriate operation in the target semantics. Since the algebra $M_{\Sigma^\sharp}$ has three carriers, ϵ is comprised of the family $\epsilon_l : M_l \longrightarrow dU_{\Sigma' l}$, $\epsilon_r : M_r \longrightarrow dU_{\Sigma' r}$, and $\epsilon_{eq} : M_{eq} \longrightarrow dU_{\Sigma' eq}$.

Definition 10. We define ϵ as follows:

$$\begin{aligned}
\epsilon_l(\alpha) &= (decode_{T_\Sigma} \times decode_{Env_M}) \ ; \ \alpha \ ; \ (encode_{T_\Sigma} \times encode_{Env_M}) \\
\epsilon_r(\beta) &= (decode_{T_\Sigma} \times decode_{Env_M}) \ ; \ \beta \ ; \ (encode_{T_\Sigma} \times encode_{Env_M} \times convert) \ ; \\
&\quad [\pi_1, \pi_3, \pi_2] \ ; \ (pushval \times 1_{Env}) \\
\epsilon_{eq}(\eta) &= (decode_{T_\Sigma} \times decode_{Env_M}) \ ; \ \eta \ ; \ (encode_{T_\Sigma} \times encode_{Env_M})
\end{aligned}$$

where $\alpha \in M_l$, $\beta \in M_r$ and $\eta \in M_{eq}$.

However, mapping the operations from the source semantics into operations in the target semantics is not enough, we also have to to show that this mapping is a homomorphism; we have to show that the *homomorphism condition* holds for every operation symbol in $\Sigma^\sharp$. Therefore, to complete the correctness proof of the compiler we have to show that the following proposition holds:

Proposition 11. *The mapping ϵ is a homomorphism from the source language semantics to the target language semantics. Explicitly, the seven equations below hold:*

$$\begin{aligned}
\epsilon_l(\kappa_l^M) &= \kappa_l^{dU} \\
\epsilon_l(x_l^M) &= x_l^{dU} \\
\epsilon_l(\sigma_l^M(\alpha_1, \alpha_2)) &= \sigma_l^{dU}(\epsilon_l(\alpha_1), \epsilon_l(\alpha_2)) \\
\epsilon_r(\kappa_r^M) &= \kappa_r^{dU} \\
\epsilon_r(x_r^M) &= x_r^{dU} \\
\epsilon_r(\sigma_r^M(\beta_1, \beta_2)) &= \sigma_r^{dU}(\epsilon_r(\beta_1), \epsilon_r(\beta_2)) \\
\epsilon_{eq}(\Rightarrow_{eq}^M (\alpha, \beta)) &= \Rightarrow_{eq}^{dU} (\epsilon_l(\alpha), \epsilon_r(\beta))
\end{aligned}$$

where $\alpha, \alpha_1, \alpha_2 \in M_l$ and $\beta, \beta_1, \beta_2 \in M_r$.

The proof of this proposition has been automated with the OBJ3 system. A complete treatment of this proof is given in the full version of this paper, which is available as the Programming Research Group technical report PRG-TR-1-94 from the Oxford University Computing Laboratory.

8 Conclusions and Related Work

Taking hints from the functional and logic programming communities, we designed an abstract machine to compile algebraic specification languages rather than to interpret the specifications at run time. We successfully employed an initial algebra semantics approach to prove correctness of a compiler specification which translates the equational rewrite rules of the source language into abstract machine code. We succeeded in fully automating the proof using the OBJ3 system.

One of the key insights we gained is that efficient implementations are usually context sensitive, whereas the unique homomorphisms necessarily represent context free substitutions. Thus, the necessary existence of the homomorphism between the semantics is overly restrictive and prevents many intuitively correct implementations of being proven correct via the Morris Square technique. However, we feel that the separation between syntax and semantics of this algebraic approach, dating back to Burstall and Landin [2], captures an important intuition, and so we want to keep this separation but avoid the homomorphic mapping. One approach is to work with *theories* of the algebras instead of with the algebras themselves. A correct implementation is then a *theory morphism*. This also avoids the rather *ad hoc* axiomatization of the set theoretic functions that feature in the Morris Square proof. An additional advance would be to use *hidden sorted algebra* [6], which avoids the context free substitution problem.

One might worry that commutativity of the Morris Square does not really prove compiler correctness if some of the homomorphisms are trivial, e.g., if they identify all commands of the source language. However, that does not happen in our case, because the map induced by the derivor is injective, and the state conversion functions are bijective.

The design of abstract machines is based on a mix of intuition and deep insight into a particular domain. It would be interesting to find suitable guidelines for the design of efficient abstract machines. One such guideline is simply that the more concrete are the underlying data structures, the more efficient is the abstract machine.

Of course we aim to remove the restriction to a single unsorted rewrite rule, and consider sets of order sorted rewrite rules.

Work closely related to ours is Klaeren and Indermark's paper [13], which presents the translation of an algebraic specification language into code for an abstract stack machine. However, their correctness proof does not use general algebra and is not machine executable. There have also been a number of definitions of abstract machines for implementing algebraic specification languages, such as [15], but usually neither a translation scheme nor a correctness proof is given. Other work related to ours includes the Categorical Abstract Machine [3] and a correctness proof for the WAM [1].

Acknowledgements. We would like to thank Dr. Grant Malcolm whose detailed comments on earlier drafts of this work contributed immensely to the presentation. Lutz Hamel would also like to thank his wife Natalie for inspiration and support. We gratefully acknowledge financial support in part from Wolfson College, Oxford, the Oxford University Computing Laboratory, and the Oxford University Committee for Graduate Studies.

References

1. E. Böerger and D. Rosenzweig. From Prolog Algebras towards WAM - A Mathematical Study of Implementation. In *Proceedings CSL'90*, Lecture Notes in Computer Science. Springer-Verlag, 1991.

2. R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4, 1969. Edinburgh University Press, eds. B. Meltzer, D. Michie.
3. G. Cousineau. The categorical abstract machine. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 25–45. Addison Wesley, 1990.
4. P. Curzon. Of what use is a verified compiler specification? Technical Report 274, University of Cambridge, Computer Laboratory, 1992.
5. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.
6. J. Goguen and G. Malcolm. Proof of correctness of object representation. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 119–142. Prentice-Hall, 1994.
7. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. Draft, Oxford University Computing Laboratory, 1993.
8. J. A. Goguen. Semantics of computation. In E. G. Manes, editor, *Proceedings, First International Symposium on Category Theory Applied to Computation and Control*, pages 234–249. University of Massachusetts at Amherst, 1974. Also in Lecture Notes in Computer Science, Volume 25, Springer, 1975, pages 151–163.
9. J. A. Goguen, J. Jouannaud, and J. Meseguer. Operational semantics of order-sorted algebra. *Lecture Notes in Computer Science*, 194, 1985.
10. J. A. Goguen, J. W. Thatcher, and E. G. Wagner. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*, pages 80–149. Prentice-Hall, 1978. Current Trends in Programming Methodology, Data Structuring, edited by R. Yeh.
11. S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* International Series. Prentice-Hall, London, 1987.
12. C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ-3. *Lecture Notes in Computer Science*, 317, 1988.
13. H. Klaeren and K. Indermark. Efficient implementation of an algebraic specification language. *Lecture Notes in Computer Science*, 394:69–89, 1989.
14. F. L. Morris. Advice on structuring compilers and proving them correct. In *ACM Symposium on Principles of Programming Languages*, pages 144–152. Association for Computing Machinery, 1973.
15. K. Richta and S. Nesvera. The abstract rewriting machine. Research Report DC-91-04, Dept. of Computers Czech Technical University, Prague, September 1991.
16. J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Lecture Notes in Computer Science*, 71:596–615, 1979.
17. D. H. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Centre, SRI International, 1983.

This article was processed using the LaTeX macro package with LLNCS style