A Constructive Approach to Compiler Correctness *

Peter Mosses

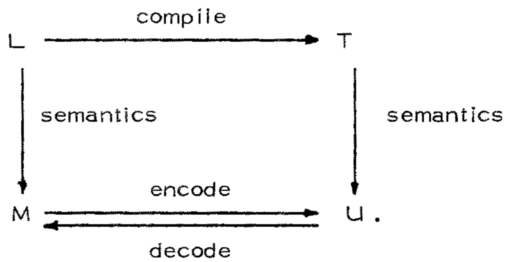
Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C, Denmark

Abstract

It is suggested that denotational semantic definitions of programming languages should be based on a small number of abstract data types, each embodying a fundamental concept of computation. Once these fundamental abstract data types are implemented in a particular target language (e.g. stack-machine code), it is a simple matter to construct a correct compiler for any source language from its denotational semantic definition. The approach is illustrated by constructing a compiler equivalent to the one which was proved correct by Thatcher, Wagner & Wright (1979).

1. Introduction

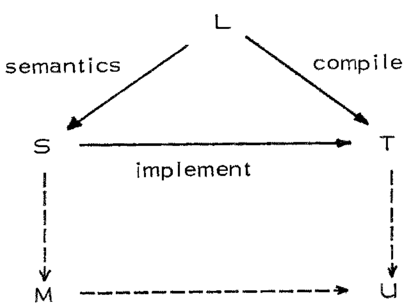
There have been several attacks on the compiler-correctness problem: by McCarthy & Painter (1967), Burstall & Landin (1969), F.L. Morris (1973) and, more recently, by Thatcher, Wagner & Wright, of the ADJ group (1979). The essence of the approach advocated in those papers can be summarised as follows: One is given a source language L, a target language T, and their respective semantics in the form of models M and U. Given also a compiler to be proved correct, one constructs an encoder: M → U (or a decoder: U → M) such that this diagram commutes:



* A revised version of this paper is to appear in the Proceedings of the 7th International Colloquium on Automata, Languages and Programming, 1980.

ADJ (1979) suggested that this is most easily done by making M, U and T into G-algebras, where G is the grammar (signature, abstract syntax) of L. The two semantic functions and the compiler are defined as homomorphisms, so the initiality of L gives the commutativity of the diagram, once encode is shown to be a homomorphism. ADJ illustrated their approach for a simple language L, including assignment, loops, expressions with side-effects and simple declarations. T was a language corresponding to flow charts with instructions for assignment and stacking. Their semantic definitions of L and T can be regarded as "standard" denotational semantics in the spirit (though not the notation!) of Scott and Strachey (1970). The simplicity of their definition of an encoder was, however, rather outweighed by the tediousness of their proof that it was a homomorphism.

We shall take a different approach in this paper. The semantics of the source language L will be given in terms of an abstract data type S, rather than a particular model. The target language T will also be taken as an abstract data type. Then the correct implementation of S by T will enable us to construct a correct compiler (from L to T) from the semantic definition of L. The compiler to be constructed is actually the composition of the semantics and the implementation, as shown by the following diagram:



A crucial point is that the implementation of S by T is proved correct independently of making S and T into the G-algebras implied by the diagram. This allows us to generate correct compilers for a whole family of source languages - languages which are similar to L, in that their denotational semantics can be given in terms of S - without repeating (or even modifying) the proof that the implementation of S by T is correct.

Note the use of the word "implements" above. We are considering the implementation of one abstract data type by another abstract data type, rather than by a particular algebra ("concrete" data type). Let us refer to the latter situation as modelling.

The main concern of this paper is with the compiler-correctness problem. However, it is hoped that the example presented below will also serve as an illustration of on-going work on making denotational semantics "less concrete" and "more modular". It is claimed that there are abstract data types corresponding to all our fundamental concepts of computation - and that any programming language can be analyzed in terms of a suitable combination of these. ("Bad" features of programming languages are shown up by the need for a complicated analysis - so long as the fundamental concepts are chosen appropriately.) Of course, only a few of the fundamental concepts are needed for semantics of the simple example language L (they include the sequential execution of actions, the computation and use of semantic values, and dynamic associations). An ordinary denotational semantics for L would make use of these concepts implicitly - the approach here is to be explicit.

The use of abstract data types in this approach encourages a greater modularity in semantic definitions, making them - hopefully - easier to read, write and modify. It seems that Burstall & Goguen's (1977) work on "putting theories together" forms a suitable formal basis for expressing the modularity. However, this aspect of the approach is not exploited here.

It should be mentioned that the early paper by McCarthy & Painter (1967) already made use of abstract data types: the relation between storing and accessing values in variables was specified axiomatically. ADJ (1979) used an abstract data type, but only for the operators on the integers and truth-values.

The approach presented here has been inspired by much of the early work on abstract data types, such as that of ADJ (1975, 1976), Guttag (1975), Wand (1977) and Zilles (1974). Also influential has been Wand's (1976) description of the application of abstract data types to language definition, although he was more concerned with definitional interpreters than with denotational semantics. Goguen's (1978) work on "distributed-fix" operators has contributed by liberating algebra from the bonds of prefix notation.

However, it is also the case that the proposed approach builds to a large extent on the work of the Scott-Strachey "school" of semantics, as described by Scott & Strachey (1971), Tennent (1976), Milne & Strachey (1976), Stoy (1977) and Gordon (1979). Also, the success of Milner (1979) in describing concurrency algebraically has provided some valuable guidelines.

The rest of this paper is organized as follows. After the explanation of some notational conventions, the abstract syntax of the ADJ (1979) source language L is given. A "standard" semantic abstract data type S is described, possible models discussed, and the standard semantics of L given. The next section presents a "stack" abstract data type T, which needs extending before the implementation of S can be expressed homomorphically. The proof of the correctness of the implementation is sketched, and a compiler - corresponding closely to ADJ's - is constructed. Finally, the application of the approach to more realistic examples is discussed.

It is assumed that the reader will be familiar with many-sorted algebras, equational specifications and - to a lesser extent - denotational semantics.

2. Standard Semantics

The notation used in this paper differs significantly from that recommended by ADJ (1979), by remaining close to the notation of the Scott-Strachey school. This is not just a matter of following tradition (although the familiarity of the notation might be a help to some readers of this paper). There are two main points of contention:

(i) The use of the semantic function explicitly in semantic equations. Although technically unnecessary, from an algebraic point of view, this allows us to regard the semantic function as just another equationally-defined operator in an abstract data type, and to forget about the machinery of homomorphisms and initial algebras (albeit temporarily!).
(ii) The use of mixfix(*) notation for the operators of the abstract syntax. Mixfix notation is a generalization of prefix, infix and postfix notation: operator symbols can be distributed freely around and between operands,

(*) called "distributed-fix" by Goguen (1978).

e.g. if-then-else. ADJ used infix and mixfix notation ($f \circ g$, $[f,g,h]$) freely in their semantic notation, but stuck to postfix notation ((x)f) for the syntactic algebra. This made the correspondence between the abstract syntax and the "usual" concrete syntax for their language rather strained. Whilst not disastrous for such a simple and well-known language as their example, the extra burden on the reader would be excessive for more realistic languages. Also, their claim of better readability does not seem to be justified.

Notational Conventions

The names of sorts are written starting with a capital, thus: A, Cmd. Algebraic variables over a particular sort are represented by the sort name, usually decorated with subscripts or primes: A, A_1, A'. Operator symbols are written with lower-case letters and non-alphabetic characters: tt, even( ), +, if then else. Families of operators are indicated by letting a part of the operator vary over a set, e.g. id := (id ∈ Id) is a family of prefix operators indexed by elements of Id. It is also convenient to allow families of sorts indexed by (sequences of) domain names from a set Δ - lower case Greek letters (δ, σ, τ) are used for the indices.

The arity and co-arity of an operator in a signature are indicated by the notation

$$S \Leftarrow f(S_1, \cdots, S_n)$$

- here, the arity of f is $S_1 \cdots S_n$, the co-arity is S. Mixfix notation can be used here for the operator symbol, giving a pleasing similarity to BNF, e.g.

Cmd <= if BExp then Cmd else Cmd.

The term "theory" is used synonymously with "abstract data type", i.e. it is basically a signature together with some laws. So much for notation.

Abstract Syntax (L)

The abstract syntax of the source language L is given in Table 1. It may be compared directly with that of ADJ (1979), although, as explained above, we shall not restrict ourselves to postfix notation for syntactic operators here. Id is taken to be a set, rather than a sort, following ADJ - in effect, this gives a parameterised abstract data type, and we need not be concerned about the details of Id.

Table 1. Abstract Syntax of L

sorts	Cmd	commands
	AExp	arithmetic expressions
	BExp	Boolean expressions
	Id	unspecified set of identifiers

operators

Cmd <=	continue		
	id := AExp	id	∈ Id
	if BExp then Cmd else Cmd		
	Cmd; Cmd		
	while BExp do Cmd		
AExp <=	aconst	aconst	∈ {0, 1}
	id	id	∈ Id
	aop1 AExp	aop1	∈ {−, pr, su}
	AExp aop2 AExp	aop2	∈ {+, −, ×}
	if BExp then AExp else AExp		
	Cmd result AExp		
	let id be AExp in AExp	id	∈ Id
BExp <=	bconst	bconst	∈ {tt, ff}
	prop AExp	prop	∈ {even}
	AExp rel AExp	rel	∈ {≤, ≥, eq}
	bop1 BExp	bop1	∈ {¬}
	BExp bop2 BExp	bop2	∈ {∧, ∨}

Standard Semantic Theory (S)

The standard semantic theory presented in Table 2 may seem a bit daunting at first. Actually, the operators themselves (left-hand column) are quite simple, but the "book-keeping" concerned with the indices (δ, σ, τ) of the sorts is somewhat cumbersome.

Table 2 could be regarded as a theory schema, or as an instantiation of a parameterised theory, where Δ is a formal parameter (as is Id). Whichever way one looks at it, the use of Δ gives a hint of modularity, as well as avoiding undue repetition in the specification.

The following informal description of S may help the reader.

Table 2. Semantic Theory S

sorts (indices: $\delta \in \Delta$; $\sigma, \tau \in \Delta^*$, where $\Delta = \{T, Z\}$)

A – actions, with source σA and target τA
Y – variables over actions, with source σY and target τY
V – values, with domain δV
X – variables over values, with domain δX

operators (indices: $id \in Id$; $n \in \{0, 1, \ldots\}$)

actions A		source σA	target τA
A <=	skip	( )	( )
	$A' \; ; \; A''$	$\sigma A' \cdot \sigma A''$	$\tau A' \circ \tau A''$
	$V!$	( )	δV
	$X.A'$	$\delta X \cdot \sigma A'$	$\tau A'$
	$A' >_n A''$	$\sigma A' \cdot s''$ where $\sigma A'' = d_1 \cdots d_n \cdot s''$,	$t' \cdot \tau A''$ and $\tau A' = d_1 \cdots d_n \cdot t'$
	tt ? A' / ff ? A''	$T \cdot \sigma A'$ where $\sigma A'' = \sigma A'$,	$\tau A'$ and $\tau A'' = \tau A'$
	fix $Y.A'$	σY where $\sigma A' = \sigma Y$	τY and $\tau A' = \tau Y$
	Y	σY	τY
	update_{id}	Z	( )
	contents_{id}	( )	Z

action variables Y		source σY	target τY
Y <=	a	( )	( )
	a_n	( )	( )

values V		domain δV	conditions
V <=	X	δX	
	aconst	Z	
	aop1 V'	Z	$\delta V' = Z$
	V' aop2 V''	Z	$\delta V' = \delta V'' = Z$
	bconst	T	
	prop V'	T	$\delta V' = Z$
	V' rel V''	T	$\delta V' = \delta V'' = Z$

value variables X		domain δX	
X <=	z	Z	
	z_n	Z	

Table 2 contd.

equations

1. $\mathrm{skip}\ ;\ A = A$
2. $A\ ;\ \mathrm{skip} = A$
3. $(A_1\ ;\ A_2)\ ;\ A_3 = A_1\ ;\ (A_2\ ;\ A_3)$
4. $V!\ \succ (X.\ A) = A\ \{X \leftarrow V\}$
5. $(V!\ ;\ A_1) \succ_n (X.\ A_2) = A_1 \succ_{n-1} (A_2\ \{X \leftarrow V\})$
6. $tt!\ \succ (tt?\ A_1\ /\ ff?\ A_2) = A_1$
7. $ff!\ \succ (tt?\ A_1\ /\ ff?\ A_2) = A_2$
8. $\mathrm{fix}\ Y.\ A = A\ \{Y \leftarrow \mathrm{fix}\ Y.\ A\}$
9. $(V!\ \succ \mathrm{update}_{id})\ ;\ \mathrm{contents}_{id} = (V!\ \succ \mathrm{update}_{id})\ ;.\ V!$
10. $(V!\ \succ \mathrm{update}_{id})\ ;\ \mathrm{contents}_{id'} = \mathrm{contents}_{id'}\ ;\ (V!\ \succ \mathrm{update}_{id})$ for $id \neq id'$
11. $A\ ;\ V! = V!\ ;\ A$ for $\tau A = (\ )$
12. $X.\ A = X'.\ A\ \{X \leftarrow X'\}$ for X' not free in A
13. $(tt?\ A_1\ /\ ff?\ A_2)\ ;\ A_3 = tt?\ (A_1\ ;\ A_3)\ /\ ff?\ (A_2\ ;\ A_3)$
14. $A_1\ ;\ (tt?\ A_2\ /\ ff?\ A_3) = tt?\ (A_1\ ;\ A_2)\ /\ ff?\ (A_1\ ;\ A_3)$
15. $(X.\ A_1)\ ;\ A_2 = X.\ (A_1\ ;\ A_2)$ for X not free in A_2
16. $A_1\ ;\ (X.\ A_2) = X.\ (A_1\ ;\ A_2)$ for X not free in A_1 and $\sigma A_1 = (\ )$
17. $V!\ ;\ (A_1 \succ X.\ A_2) = A_1 \succ X.\ (V!\ ;\ A_2)$ for X not free in V and $\tau(A_1) = (d)$

The basic concept is that of actions (A). Actions not only have an "effect", but may also consume and/or produce sequences of values (V). These values can be thought of as belonging to the "semantic domains" in Δ. i.e. T and Z. The book-keeping referred to above mainly consists of keeping track of the number and sorts of values consumed (σ, for source) and produced (τ, for target). Note that a raised dot ($\cdot$) stands for concatenation of sequences in Δ^* and ( ) is the empty sequence.

Variables (X) are used to name computed values, and to indicate dependency on these values (by actions and other computed values). Variables over actions (Y) allow the easy expression of recursion and iteration.

We consider the value operators first. They are taken straight from the "underlying" data type of ADJ (1979). It is assumed that bconst, prop, etc. vary over the same sets as in Table 1, thus giving families of operators. The Boolean operators ($\neg$, $\wedge$, $\vee$) are not needed in giving the semantics of L, and have been omitted from S (as have variables over truth values).

There is a domain name $\delta \in \Delta$ associated with each value of V; also, the domain name Z is associated with the variables used to name values in the sort Z. (This would be of more importance if we were to include variables naming T-values as well - the idea is just to make sure that a sort-preserving substitution can be defined.)

The action operators are perhaps less familiar. A <= skip is the null action, it is an identity for the sequencing operator A <= A'; A''. Note that sequencing is additive in the sources and targets.

The most basic action operator producing a value is A <= V! . The consumption of a value is effected by A <= X.A', and X is bound to the value in A'. To indicate that n values produced by one action are consumed by another, we have the operator $A \Leftarrow A' >\!\!-_n A''$, and it is the first n values produced by A' which get consumed by A''. ($A \Leftarrow A' >\!\!-_0 A''$ is equivalent to A <= A'; A''. $>\!\!-_n$ may be written simply as >- when the value of n can be deduced from the context.)

A <= tt? A' / ff? A'' is a choice operator: it consumes a truth value (tt or ff) and reduces to A' or A''. The sources and targets of A' and A'' must be identical.

A <= fix Y. A' binds Y in A' and, together with A <= Y, allows the expression of recursively-defined actions. Acutally, it is used here (in describing L) only in a very limited form, corresponding to iteration: A <= fix a.A' >- tt? A''; a / ff? skip, where A' produces a truth-value, and the action variable, a, does not occur free in A' or A''.

Finally, there are two families of operators for storing and accessing computed values: $A \Leftarrow \text{update}_{id}$ and $A \Leftarrow \text{contents}_{id}$, for id $\in$ Id. Only integer values may be stored.

Now for the equations of Table 2, specifying the laws which the operators of S are to satisfy. ADJ (1979) gave equations for the value operators - they are much as one might expect, and are not repeated here. The novelty of S lies in its action operators.

To avoid getting bogged down in irrelevant details, the equations for the binding operators of S (A <= X.A' and A <= fix Y.A') are given with the help of notation for syntactic substitution: for any action term A of S, $A\{X \leftarrow V\}$ is the term with all free occurrences of X replaced by the value term V (and with uniform changes of bound variables in A to avoid "capturing" free variables in V). Similarly for $A\{Y \leftarrow A'\}$. This syntactic substitution could have been added as an operator to S, and specified equationally.

The equations should now be self-explanatory. What might not be obvious is that they are the "right" equations, and are neither inconsistent nor incomplete. It would delay us too much to go into all the details here, but the idea is to use a Scott-model for S to show consistency, and a so-called canonical term algebra to prove completeness.

The "obvious" Scott-model for S (corresponding to the M of ADJ (1979)) has as carrier for sort A, with $\sigma A = d_1 \cdots d_m$ and $\tau A = d'_1 \cdots d'_n$ (d_i, $d'_i \in \{T, Z\}$), the domain of continuous functions

$$[Env \times d_1 \times \cdots \times d_m \rightarrow Env \times d'_1 \times \cdots \times d'_n],$$

where Env = Id → Z. (Of course, one could also take a continuations-based model, or one with static environments, if one preferred.)

The reader may have noticed that S has binding operators, and that terms can have "free" (semantic) variables. This raises the question of whether a modelling function from S to a Scott-model could be expressed as a homomorphism, or whether one must allow the function to take an environment (giving the values of the semantic variables, not of the program variables). Robin Milner has suggested that one can regard a binding operator as a notational means for representing a family, indexed by the values which may be substituted for the bound variables. E.g. X.A represents the family $\langle A\{X \leftarrow v\}\rangle_{v \in \delta X}$, and in $V! \succ (X.A)$, the second operand of $\succ$ is a family. This enables the modelling function to be given as a homomorphism. One might wonder whether the introduction of operators acting on (in general) infinite families undermines the whole algebraic framework, but Reynolds (1977) shows that this is not the case. Anyway, modelling is not our main concern in this paper, so let us leave the topic there.

Standard Semantics

The "standard" denotational semantics of L in terms of the abstract data type S is given in Table 3. The use of the "semantic equations" notation, with the explicit definition of the semantic function, is defended at the beginning of this section. To allow the omission of parentheses, it is assumed that the operator '.' binds as far to the right as possible (as in λ-notation).

Note that $\mathrm{sem}[\![\ ]\!]$ can be considered either as an operator in an extension of the theories L and S, or else as a homomorphism from L to a derived theory of S. Under the latter view, the composition of sem with the modelling function (from S to the Scott-model mentioned above) yields the semantics which ADJ (1979) gave for L.

Table 3. Standard Semantics for L using S

<u>operators</u>

$A \Leftarrow \mathrm{sem}[\![\mathrm{Cmd}]\!] \qquad \sigma A = (\ ), \quad \tau A = (\ )$

$A \Leftarrow \mathrm{sem}[\![\mathrm{AExp}]\!] \qquad \sigma A = (\ ), \quad \tau A = Z$

$A \Leftarrow \mathrm{sem}[\![\mathrm{BExp}]\!] \qquad \sigma A = (\ ), \quad \tau A = T$

<u>$\mathrm{sem}[\![\mathrm{Cmd}]\!]$ equations</u> $\qquad (\mathrm{id} \in \mathrm{Id})$

$\mathrm{sem}[\![\mathrm{continue}]\!] = \mathrm{skip}$

$\mathrm{sem}[\![\mathrm{id} := \mathrm{AExp}]\!] = \mathrm{sem}[\![\mathrm{AExp}]\!] \succ \mathrm{update}_{\mathrm{id}}$

$\mathrm{sem}[\![\mathrm{if\ BExp\ then\ Cmd}_1\ \mathrm{else\ Cmd}_2]\!] =$

$\qquad \mathrm{sem}[\![\mathrm{BExp}]\!] \succ \mathrm{tt?}\ \mathrm{sem}[\![\mathrm{Cmd}_1]\!]\ /\ \mathrm{ff?}\ \mathrm{sem}[\![\mathrm{Cmd}_2]\!]$

$\mathrm{sem}[\![\mathrm{Cmd}_1\ ;\ \mathrm{Cmd}_2]\!] = \mathrm{sem}[\![\mathrm{Cmd}_1]\!]\ ;\ \mathrm{sem}[\![\mathrm{Cmd}_2]\!]$

$\mathrm{sem}[\![\mathrm{while\ BExp\ do\ Cmd}]\!] =$

$\qquad \mathrm{fix}\ a.\ \mathrm{sem}[\![\mathrm{BExp}]\!] \succ \mathrm{tt?}\ \mathrm{sem}[\![\mathrm{Cmd}\ ]\!]\ ;\ a\ /\ \mathrm{ff?}\ \mathrm{skip}$

<u>$\mathrm{sem}[\![\mathrm{AExp}]\!]$ equations</u>

$\mathrm{sem}[\![\mathrm{aconst}]\!] = \mathrm{aconst}\ !$

$\mathrm{sem}[\![\mathrm{id}]\!] = \mathrm{contents}_{\mathrm{id}}$

$\mathrm{sem}[\![\mathrm{aop1\ AExp}]\!] = \mathrm{sem}[\![\mathrm{AExp}]\!] \succ z.\ (\mathrm{aop1}\ z)\ !$

$\mathrm{sem}[\![\mathrm{AExp}_1\ \mathrm{aop2}\ \mathrm{AExp}_2]\!] =$

$\qquad \mathrm{sem}[\![\mathrm{AExp}_1]\!] \succ z_1.\ \mathrm{sem}[\![\mathrm{AExp}_2]\!] \succ z_2.\ (z_1\ \mathrm{aop2}\ z_2)\ !$

$\mathrm{sem}[\![\mathrm{if\ BExp\ then\ AExp}_1\ \mathrm{else\ AExp}_2]\!] =$

$\qquad \mathrm{sem}[\![\mathrm{BExp}]\!] \succ \mathrm{tt?}\ \mathrm{sem}[\![\mathrm{AExp}_1]\!]\ /\ \mathrm{ff?}\ \mathrm{sem}[\![\mathrm{AExp}_2]\!]$

$\mathrm{sem}[\![\mathrm{Cmd\ result\ AExp}]\!] = \mathrm{sem}[\![\mathrm{Cmd}]\!]\ ;\ \mathrm{sem}[\![\mathrm{AExp}]\!]$

$\mathrm{sem}[\![\mathrm{let\ id\ be\ AExp}_1\ \mathrm{in\ AExp}_2]\!] =$

$\qquad \mathrm{contents}_{\mathrm{id}} \succ z_1.\ (\mathrm{sem}[\![\mathrm{AExp}_1]\!] \succ \mathrm{update}_{\mathrm{id}});$

$\qquad \mathrm{sem}[\![\mathrm{AExp}_2]\!] \succ z_2.\ (z_1\ ! \succ \mathrm{update}_{\mathrm{id}});\ z_2\ !$

<u>$\mathrm{sem}[\![\mathrm{BExp}]\!]$ equations</u>

$\mathrm{sem}[\![\mathrm{bconst}]\!] = \mathrm{bconst}\ !$

$\mathrm{sem}[\![\mathrm{prop\ AExp}]\!] = \mathrm{sem}[\![\mathrm{AExp}]\!] \succ z.\ (\mathrm{prop}\ z)\ !$

$\mathrm{sem}[\![\mathrm{AExp}_1\ \mathrm{rel}\ \mathrm{AExp}_2]\!] =$

$\qquad \mathrm{sem}[\![\mathrm{AExp}_1]\!] \succ z_1.\ \mathrm{sem}[\![\mathrm{AExp}_2]\!] \succ z_2.\ (z_1\ \mathrm{rel}\ z_2)\ !$

$\mathrm{sem}[\![\neg\ \mathrm{BExp}]\!] = \mathrm{sem}[\![\mathrm{BExp}]\!] \succ \mathrm{tt?}\ \mathrm{ff!}\ /\ \mathrm{ff?}\ \mathrm{tt!}$

$\mathrm{sem}[\![\mathrm{BExp}_1 \wedge \mathrm{BExp}_2]\!] = \mathrm{sem}[\![\mathrm{BExp}_1]\!] \succ \mathrm{tt?}\ \mathrm{sem}[\![\mathrm{BExp}_2]\!]\ /\ \mathrm{ff?}\ \mathrm{ff!}$

$\mathrm{sem}[\![\mathrm{BExp}_1 \vee \mathrm{BExp}_2]\!] = \mathrm{sem}[\![\mathrm{BExp}_1]\!] \succ \mathrm{tt?}\ \mathrm{tt!}\ /\ \mathrm{ff?}\ \mathrm{sem}[\![\mathrm{BExp}_2]\!]$

3. Stack Implementation

We now take a look at the target language T for our compiler. Like the target language taken by ADJ (1979), T represents flow-charts over stack-machine instructions. The abstract syntax of T is given in Table 4.

A comparison of Tables 2 and 4 shows that T is rather similar to S. However, this should not be too surprising: the same fundamental concepts of computation are being used, e.g. sequencing of actions, storing of values. Note that $A <= A' \underset{n}{\rightarrow} A''$ in T corresponds to $A <= A' \underset{n}{\succ} A''$ in S, but it is the last n values produced by A' which get consumed (in reversed order), by A'' in T. Also, the value terms V in T are restricted to be constants, and A <= V! represents pushing V onto the stack. The value operators (prop, rel, aop1, aop2) of S have become actions operating on the stack in T. A <= switch interchanges the top two values on the stack. Finally, there are no value variables X in T - and hence no A <= X.A' either.

However, T is to be more than just a language: it is to be an abstract data type! There are equations, similar to those for S, which the operators of T must satisfy. (The equations are not listed here, although a couple of them are given (indirectly) by Table 5.)

So the problem is now to implement one abstract data type (S) by another (T), and show that the implementation is correct. If imp: S → T, then imp is said to be a correct implementation of S by T if it is an injective homomorphism. In other words, imp respects the equations of S: for any s, s' in S, imp⟦s⟧ = imp ⟦s'⟧ iff s = s'. Having found such an imp, the composite imp ∘ sem: L → T is a correct compiler from L to T.

Unfortunately, it is actually impossible to implement S by the T of Table 4! To see why, consider a term of S with free (value-) variables, such as z! . What could imp give in T as the implementation of this term? If one tries to answer this question, one discovers that free variables in S correspond to values at an unknown depth on the stack in T - and that there is no way of representing such values. (Considering binding operators as a means for representing families of terms without free variables doesn't help, as there is no means of representing a family in T.)

This is annoying, because one can easily implement the closed terms of S by T: one knows the positions of all the values on the stack. Moreover, only closed terms were used in giving the semantics of L. One could argue that we could make do with an implementation of only the closed terms of S, and proceed with our compiler construction. However, to show that the implementation (and hence the compiler) is correct, we need it to be a homomorphism – and that means considering all the terms of S, including those with free variables.

Table 4. Stack Theory T

sorts (indices: $\delta \in \Delta$; $\sigma, \tau \in \Delta^*$, where $\Delta = \{T, Z\}$)

A – actions, with source σA and target τA
Y – variables over actions, with source σY and target τY
V – values, with domain δV

operators (indices: $id \in Id$; $n \in \{0, 1, \ldots\}$)

actions A	source σA	target τA
A <= skip	( )	( )
$A' ; A''$	$\sigma A' \cdot \sigma A''$	$\tau A' \cdot \tau A''$
V!	( )	δV
$A' \rightarrow_n A''$	$\sigma A' \cdot s''$ where $\sigma A'' = d_1 \cdots d_n \cdot s''$,	$t' \cdot \tau A''$ and $\tau A' = t' \cdot d_n \cdots d_1$
tt ? A' / ff ? A''	where $\sigma A' = \sigma A''$	and $\tau A' = \tau A''$
fix Y.A'	σY where $\sigma A' = \sigma Y$	τY and $\tau A' = \tau Y$
Y	σY	τY
$update_{id}$	Z	( )
$contents_{id}$	( )	Z
switch	$Z \cdot Z$	$Z \cdot Z$
prop	Z	T
rel	$Z \cdot Z$	T
aop1	Z	Z
aop2	$Z \cdot Z$	Z

action variables Y	source σY	target τY
Y <= a	( )	( )
a_n	( )	( )

values V	domain δV	
V <= aconst	Z	
bconst	T	

Thus we are forced to extend T, before we can use it to give a homomorphic implementation of S. The most natural extension to take seems to be Tx, given in Table 5. The action A <= X.A' can be thought of as removing the top item from the stack and binding it to X in A'.

Now we are able to give a homomorphic implementation of S by Tx, and prove it correct. But how does that help us in constructing a compiler from L to T (rather than to Tx)? Recall that only closed terms of S are used in the semantics of L – and they are implemented by closed terms in Tx. It just so happens that any closed term of Tx is equivalent to a term of T, i.e. one without any value variables at all! This ensures that our compiler from L to Tx can be converted to one from L to T.

Actually, that is not quite true. We need to add a few derived operators to Tx: generalizations of A <= switch, for permuting the top values on the stack. (This is analogous to adding the combinators (S, K, etc.) to the λ-calculus, in using them to eliminate λ-abstractions.) The extra operators, extending Tx to Tx', are given in Table 6. It turns out that they do not occur in the compiler we construct for L, because of the lack of exploitation of the generality of S in giving the semantics of L.

Table 5. Extension of T to Tx

sorts X – variables over values, with domain δV

operators

actions A	source σA	target τA
A <= X.A'	δX • σA'	τA'
values V	**domain δV**	
V <= X	δX	
value variables X	**domain δX**	
X <= t	T	
t_n	T	
z	Z	
z_n	Z	

equations similar to those of S, except for:

1. V! → (X.A) = A{X ← V}
2. $(A_1 ; V!) \underset{n}{\rightarrow} (X.A_2) = A_1 \underset{n-1}{\longrightarrow} (A_2\{X \leftarrow V\})$
3. switch = $z_1. z_2. (z_2! ; z_1!)$

Table 6 also gives the (derived) equations which are used in converting closed terms in Tx' to ones without value variables. Note that these equations simplify considerably when the sources or targets of actions are empty: $up^{d}_{()}$ and $down^{d}_{()}$ have no effect, and may be removed.

At last we can implement S, by Tx'. The implementation function, imp: S → Tx', is defined in Table 7, using the same notation as was used for defining the semantics of L. S-operators now occur inside ⟦ ⟧ (in contrast to Table 2). As one can see, the implementation

Table 6. Extension of Tx to Tx'

operators (indices: $d, d_i \in \Delta$)

actions A		source σA	target τA
A <=	pop	d	$()$
	copy	d	$d \cdot d$
	$up^{d}_{d_1 \cdots d_n}$	$d_n \cdots d_1 \cdot d$	$d_1 \cdots d_n \cdot d$
	$down^{d}_{d_1 \cdots d_n}$	$d \cdot d_n \cdots d_1$	$d \cdot d_1 \cdots d_n$
	$flip^{n}_{d_1 \cdots d_m}$	$d_m \cdots d_1$	$d_{n+1} \cdots d_m \cdot d_n \cdots d_1$

equations where $x_{(i)} = t_{(i)}$, if $d_{(i)} = T$
$z_{(i)}$, if $d_{(i)} = Z$

1. $pop_d = x.\ skip$
2. $copy_d = x.\ (x!\ ;\ x!)$
3. $up^{d}_{d_1 \cdots d_n} = x_n \cdots x_1.\ x.\ (x_1!\ ;\ \ldots\ ;\ x_n!\ ;\ x!)$
4. $down^{d}_{d_1 \cdots d_n} = x.\ x_n \cdots x_1.\ (x!\ ;\ x_1!\ ;\ \ldots\ ;\ x_n!)$
5. $flip^{n}_{d_1 \cdots d_m} = x_m \cdots x_1.\ (x_{n+1}!\ ;\ \ldots\ ;\ x_m!\ ;\ x_n!\ ;\ \ldots\ ;\ x_1!)$

---

6. $X.\ (X! \to A) = A$ when X not free in A
7. $X!\ ;\ A = X! \to down^{\delta X}_{\delta A} \to A$
8. $A\ ;\ X! = X! \to down^{\delta X}_{\delta A} \to A \to up^{\delta X}_{\tau A}$
9. $X! \to (X! \to A) = X! \to copy_{\delta X} \to A$
10. $A_1 \to (X! \to A_2) = X! \to down^{\delta X}_{\sigma A_1} \to A_1 \to up^{\delta X}_{\tau A_1} \to A_2$
11. $tt\ ?\ (X! \to A_1)\ /\ ff\ ?\ (X! \to A_2) = X! \to down^{\delta X}_{T} \to (tt\ ?\ A_1\ /\ ff\ ?\ A_2)$
12. $fix\ Y.\ (X! \to A) = X! \to (fix\ Y.\ copy_{\delta X} \to A) \to up^{\delta X}_{\tau A} \to pop_{\delta X}$
13. $A = X! \to (pop_{\delta X}\ ;\ A)$

itself is really quite trivial: most of the operators go straight over from S to Tx'. The exceptions are value transfers $A <= A' \succ_n A''$, which cause some "shuffling" on the stack; and the production of compound values $A <= V!$, which get sequentialized.

The rest of this section sketches the proof of the correctness of imp, and justifies the claim that value variables can be eliminated from closed terms of Tx'. The next section goes on to construct a correct compiler from L to T.

Table 7. Implementation of S by Tx'

operators

$$A <= \mathrm{imp}[\![A']\!] \quad \sigma A = \sigma A', \quad \tau A = \tau A'$$
$$Y <= \mathrm{imp}[\![Y']\!] \quad \sigma Y = \sigma Y', \quad \tau Y = \tau Y'$$
$$A <= \mathrm{imp}[\![V]\!] \quad \sigma A = (\,), \quad \tau A = \delta V$$
$$X <= \mathrm{imp}[\![X']\!] \quad \delta X = \delta X'$$

imp⟦A⟧ equations

$$\mathrm{imp}[\![\mathrm{skip}]\!] = \mathrm{skip}$$
$$\mathrm{imp}[\![A_1 \,;\, A_2]\!] = \mathrm{imp}[\![A_1]\!] \,;\, \mathrm{imp}[\![A_2]\!]$$
$$\mathrm{imp}[\![V!]\!] = \mathrm{imp}[\![V]\!]$$
$$\mathrm{imp}[\![X.A]\!] = \mathrm{imp}[\![X]\!].\ \mathrm{imp}[\![A]\!]$$
$$\mathrm{imp}[\![A_1 \succ_n A_2]\!] = \mathrm{imp}[\![A_1]\!] \rightarrow \mathrm{flip}^{n}_{\tau A_1} \underset{n}{\rightarrow} \mathrm{imp}[\![A_2]\!]$$
$$\mathrm{imp}[\![tt?\ A_1 \,/\, ff?\ A_2]\!] = tt?\ \mathrm{imp}[\![A_1]\!] \,/\, ff?\ \mathrm{imp}[\![A_2]\!]$$
$$\mathrm{imp}[\![\mathrm{fix}\ Y.A]\!] = \mathrm{fix}\ \mathrm{imp}[\![Y]\!].\ \mathrm{imp}[\![A]\!]$$
$$\mathrm{imp}[\![Y]\!] = \mathrm{imp}[\![Y]\!] \qquad \text{(the Y on the left is an action)}$$
$$\mathrm{imp}[\![\mathrm{update}_{id}]\!] = \mathrm{update}_{id}$$
$$\mathrm{imp}[\![\mathrm{contents}_{id}]\!] = \mathrm{contents}_{id}$$

imp⟦V⟧ equations

$$\mathrm{imp}[\![X]\!] = X!$$
$$\mathrm{imp}[\![\mathrm{aconst}]\!] = \mathrm{aconst}!$$
$$\mathrm{imp}[\![\mathrm{aop1}\ V]\!] = \mathrm{imp}[\![V]\!] \underset{1}{\rightarrow} \mathrm{aop1}$$
$$\mathrm{imp}[\![V_1\ \mathrm{aop2}\ V_2]\!] = (\mathrm{imp}[\![V_1]\!] \,;\, \mathrm{imp}[\![V_2]\!]) \underset{2}{\rightarrow} \mathrm{aop2}$$
$$\mathrm{imp}[\![\mathrm{bconst}]\!] = \mathrm{bconst}!$$
$$\mathrm{imp}[\![\mathrm{prop}\ V]\!] = \mathrm{imp}[\![V]\!] \underset{1}{\rightarrow} \mathrm{prop}$$
$$\mathrm{imp}[\![V_1\ \mathrm{rel}\ V_2]\!] = (\mathrm{imp}[\![V_1]\!] \,;\, \mathrm{imp}[\![V_2]\!]) \underset{2}{\rightarrow} \mathrm{rel}$$

(imp⟦X⟧, imp⟦Y⟧ are identities– equations omitted)

The proof of the correctness of imp: S → Tx' is quite routine, but unfortunately no shorter than that of ADJ (1979). Recall that we are to prove that for terms s, s' in S, imp⟦s⟧ = imp⟦s'⟧ if and only if s = s'. The "if" part is the simpler: it is sufficient to show that for all equations s = s' in the specification of S, imp⟦s⟧ = imp⟦s'⟧ can be obtained from the equations of Tx'.

The "only if" part says that imp is injective. The easiest way to prove this seems to be to define an inverse for imp, abs: Tx' → S. This is just as simple as defining imp, and only the few non-trivial cases of the definition are given in Table 8. Using the equations of S, one can show that abs ∘ imp⟦s⟧ = s for all terms s in S. Furthermore, it can be shown that for all terms t, t' in Tx', abs⟦t⟧ = abs⟦t'⟧ if t = t' – this is just like the "if" part already proved for imp. But then, taking t = imp⟦s⟧ and t' = imp⟦s'⟧, it follows that s = s' if imp⟦s⟧ = imp⟦s'⟧, which is the desired result.

As for the elimination of value variables from closed terms of Tx', there is an algorithm, resembling the standard one for converting λ-calculus expressions to combinators. The algorithm proceeds as follows.

Table 8. Abstraction from Tx' to S

operators

$A \Leftarrow \text{abs}\,[\![A']\!] \quad \sigma A = \sigma A', \quad \tau A = \tau A'$

$Y \Leftarrow \text{abs}\,[\![Y']\!] \quad \sigma Y = \sigma Y', \quad \tau Y = \tau Y'$

$V \Leftarrow \text{abs}\,[\![V']\!] \quad \delta V = \delta V'$

$X \Leftarrow \text{abs}\,[\![X']\!] \quad \delta X = \delta X'$

abs ⟦A⟧ equations (examples)

. . .

$\text{abs}[\![A_1 \underset{n}{\rightarrow} A_2]\!] = \text{abs}[\![A_1]\!] \succ \text{flop}^n_{\tau A_1} \underset{n}{\succ} \text{abs}[\![A_2]\!]$

where $\text{flop}^n_{d_1 \ldots d_m} = x_m \ldots x_1.\ (x_1! \ ; \ \ldots \ ; \ x_n! \ ; \ x_m! \ ; \ \ldots \ ; \ x_{n+1}!)$

$\text{abs}[\![V!]\!] = \text{abs}[\![V]\!]\ !$

$\text{abs}[\![\text{aop1}]\!] = z.\ (\text{aop1}\ z)\ !$

$\text{abs}[\![\text{aop2}]\!] = z_1.\ z_2.\ (z_1\ \text{aop}\ z_2)\ !$

$\text{abs}[\![\text{prop}]\!] = z.\ (\text{prop}\ z)\ !$

$\text{abs}[\![\text{rel}]\!] = z_1.\ z_2.\ (z_1\ \text{rel}\ z_2)\ !$

Let A be a closed action term of Tx'. If A does not contain any occurrences of X.A', then it cannot contain any occurrences of X (by closedness) and we are done. Otherwise, consider an innermost occurrence of X.A' in A. If X does not occur free in A', then X.A' can be replaced by $\text{pop}_{\delta X}$; A', by the equations in Table 6, and so this occurrence of X.A' has been eliminated. On the other hand, if X does occur free in A', it must be as an action: X! . The equations of Table 6, interpreted as left-to-right replacement rules, allow A' to be transformed to the form X! → A'', where X does not occur in A''. But then X.A' can be replaced by A'', and again the occurrence of X.A' has been eliminated. As no extra occurrences have been introduced in the process (thanks to the use of the "combinators" pop, copy, up and down) the iteration of this process removes all occurrences of X.A' from A.

4. Compiler Construction

We are now able to construct a correct compiler from L to T - or for any other source language whose semantics is given in terms of S. All we need to do is to take comp: L → Tx' as imp ∘ sem, and, using the fact that imp: S → Tx' is a homomorphism, combine the definitions of imp and sem to a definition of comp. The correctness of comp comes from the correctness of imp. This correctness is preserved under transforming the terms in Tx' in the definition, to terms of T, using the algorithm of the previous section. The finished product is shown in Table 9.

The process of transformation is not as painful as the equations of Table 6 (used as replacement rules) might suggest. This is because the only action sorts used in giving the semantics of L have an empty source, and an empty or singleton target. Moreover, A' $>_n$ A'' is only used for n=1. It can be shown from the equations of Tx' that flip_d^1 can be omitted from the definition of imp, and that $\text{down}_{()}^d$ and $\text{up}_{()}^d$ are unnecessary in the equations in Table 6. In addition, up_z^z is equivalent to switch. These simplifications make the transformation from Tx' to T quite straightforward, and the only extra step necessary to obtain Table 9 is the removal of a couple of occurrences of switch; switch.

Table 9. Compiler from L to T

operators

$$A \Leftarrow comp[\![Cmd]\!] \qquad \sigma A = (\,), \quad \tau A = (\,)$$
$$A \Leftarrow comp[\![AExp]\!] \qquad \sigma A = (\,), \quad \tau A = Z$$
$$A \Leftarrow comp[\![BExp]\!] \qquad \sigma A = (\,), \quad \tau A = T$$

comp⟦Cmd⟧ equations

$$comp[\![continue]\!] = skip$$
$$comp[\![id := AExp]\!] = comp[\![AExp]\!] \rightarrow update_{id}$$
$$comp[\![if\ BExp\ then\ Cmd_1\ else\ Cmd_2]\!] =$$
$$comp[\![BExp]\!] \rightarrow tt?\ comp[\![Cmd_1]\!]\ /\ ff?\ comp[\![Cmd_2]\!]$$
$$comp[\![Cmd_1\ ;\ Cmd_2]\!] = comp[\![Cmd_1]\!]\ ;\ comp[\![Cmd_2]\!]$$
$$comp[\![while\ BExp\ do\ Cmd]\!] =$$
$$fix\ a.\ comp[\![BExp]\!] \rightarrow tt?\ comp[\![Cmd]\!]\ ;\ a\ /\ ff?\ skip$$

comp⟦AExp⟧ equations

$$comp[\![aconst]\!] = aconst!$$
$$comp[\![id]\!] = contents_{id}$$
$$comp[\![aop1\ AExp]\!] = comp[\![AExp]\!] \rightarrow aop1$$
$$comp[\![AExp_1\ aop2\ AExp_2]\!] = comp[\![Aexp_1]\!] \rightarrow comp[\![AExp_2]\!] \underset{2}{\rightarrow} aop2$$
$$comp[\![if\ BExp\ then\ AExp_1\ else\ AExp_2]\!] =$$
$$comp[\![BExp]\!] \rightarrow tt?\ comp[\![AExp_1]\!]\ /\ ff?\ comp[\![AExp_2]\!]$$
$$comp[\![Cmd\ result\ AExp]\!] = comp[\![Cmd]\!]\ ;\ comp[\![AExp]\!]$$
$$comp[\![let\ id\ be\ AExp_1\ in\ AExp_2]\!] =$$
$$contents_{id} \rightarrow comp[\![AExp_1]\!] \rightarrow update_{id};$$
$$comp[\![AExp_2]\!] \underset{2}{\rightarrow} switch \rightarrow update_{id}$$

comp⟦BExp⟧ equations

$$comp[\![bconst]\!] = bconst!$$
$$comp[\![prop\ AExp]\!] = comp[\![AExp]\!] \rightarrow prop$$
$$comp[\![AExp_1\ rel\ ARxp_2]\!] = comp[\![AExp_1]\!] \rightarrow comp[\![AExp_2]\!] \underset{2}{\rightarrow} rel$$
$$comp[\![\neg\ BExp]\!] = comp[\![BExp]\!] \rightarrow tt?\ ff!\ /\ ff?\ tt!$$
$$comp[\![BExp_1 \wedge BExp_2]\!] = comp[\![BExp_1]\!] \rightarrow tt?\ comp[\![BExp_2]\!]\ /\ ff?\ ff!$$
$$comp[\![BExp_1 \vee BExp_2]\!] = comp[\![BExp_1]\!] \rightarrow tt?\ tt!\ /\ ff?\ comp[\![BExp_2]\!]$$

Conclusion

By using a form of denotational semantics based on abstract data types, we have seen how to construct correct compilers for a whole family of source languages directly from their semantic definitions.

For realistic source languages (such as Pascal, Clu, Ada), the feasibility of the approach presented here depends on the extent to which their denotational semantics can be given in terms of a small number of fundamental abstract data types. On the other hand, going to more realistic target languages should not present any major problems – except that it might prove rather difficult to exploit the "richness" of some machine codes!

Finally, whydid our constructed compiler turn out to be so similar to the one proved correct by ADJ (1979)? One might suspect that our construction was "rigged" to deal with just this example – but that is not the case. Another possibility is that ADJ themselves constructed their compiler systematically – albeit informally – from their semantic definition. It may also be that there is essentially only one correct compiler from L to T! In any case, for realistic source languages, it seems safe to conjecture that compilers proved correct using the approach of ADJ (1979) will reflect the structure of the semantic definition of the source language, and in general be constructible by the method outlined here!

References

ADJ (⊆ { J.A. Goguen, J.W. Thatcher, E.A. Wagner, J.B. Wright })

(1975) "Initial algebra semantics and continuous algebras", IBM Res. Rep. RC-5701, 1975. JACM 24 (1977) 68-95.

(1976) "An initial algebra approach to the specification, correctness, and implementation of abstract data types", IBM Res. Rep. RC-6487, 1976. Current Trends in Programming Methodology IV (R. Yeh, ed.), Prentice Hall, 1979.

(1979) "More on advice on structuring compilers and proving them correct", IBM Res. Rep. 7588, 1979. Proc. Sixth Int. Coll. on Automata, Languages and Programming, Graz, 1979.

Burstall. R.M. & Goguen, J.A.

(1977) "Putting theories together to make specifications", Proc. Fifth. Int. Joint Conf. on Artificial Intelligence, Boston, 1977.

Burstall, R.M. & Landin, P. J.

(1969) "Programs and their proofs: an algebraic approach", Machine Intelligence 4, 1969.

Goguen, J.A.

(1978) "Order sorted algebras: exceptions and error sorts, coercions and overloaded operators", Semantics and Theory of Comp. Rep. 14, UCLA, 1978.

Gordon, M. J.C.

(1979) The Denotational Description of Programming Languages, Springer-Verlag, 1979.

Guttag, J.V.

(1975) "The specification and application to programming of abstract data types", Tech. Rep. CSRG-59, Toronto University, 1975.

McCarthy, J. & Painter, J.

(1967) "Correctness of a compiler for arithmetic expressions", Proc. Symp. in Applied Math. 19 (1967) 33-41.

Milne, R.E. & Strachey, C.

(1976) A Theory of Programming Language Semantics, Chapman & Hall (UK), John Wiley (USA), 1976.

Milner, R.

(1979) Algebraic Concurrency, unpublished lecture notes.

Reynolds, J.C.

(1977) "Semantics of the domain of flow diagrams", JACM 24 (1977) 484-503.

Scott, D.S. & Strachey, C.

(1971) "Toward a mathematical semantics for computer languages", Tech. Mono. PRG-6, Oxford University, 1971.

Stoy, J.E.

(1977) Denotational Semantics, MIT Press, 1977.

Tennent, R.D.

(1976) "The denotational semantics of programming languages", CACM 19 (1976) 437-453.

Wand, M.

(1976) "First order identities as a defining language", Tech. Rep. 29, Indiana University, 1976 (revised: 1977).

(1977) "Final algebra semantics and data type extensions", Tech. Rep. 65, Indiana University, 1977. JCSS 19 (1979) 27-44.

Zilles, S.N.

(1974) "Algebraic specification of data types", Computation Structures Group Memo 119, MIT, 1974.