ISSN 0105-8517

A CONSTRUCTIVE APPROACH TO COMPILER CORRECTNESS

by

Peter D. Mosses

DAIMI PB-118

April 1980

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 have been 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 similar to the one which was proved correct by Thatcher, Wagner & Wright (1979). Some familiarity with many-sorted algebras is presumed.

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 and shows that this diagram commutes:

$$\begin{tikzcd}
L \arrow[r, "\text{compile}"] \arrow[d, "\text{semantics}"'] & T \arrow[d, "\text{semantics}"] \\
M \arrow[r, "\text{encode}"] & U .
\end{tikzcd}$$

*) An earlier version of this paper is to appear in Proceedings of a Workshop on Semantics-Directed Compiler Generation, Aarhus, 1980 (Springer-Verlag).

It is assumed that the semantic and compiling functions are "syntax-directed". This amounts to insisting on denotational semantics in the style of Scott & Strachey (1971): "The values of expressions are determined in such a way that the value of a whole expression depends functionally on the values of its parts". ADJ (1979) reformulated this in the framework of initial algebra semantics, where the grammar, say G, of L is identified with "the" initial G-algebra. The advantage of this is that a semantic function: L → M can be seen to be a (by initiality, unique) homomorphism from L to a G-algebra based on the model M. Similarly, a compiling function: L → T is a homomorphism from L to a G-algebra derived from T, and then the semantics: T → U induces a G-algebra based on U.

So L, M, T and U can be considered as G-algebras, and the two semantics and the compiler are homomorphisms. A proof that encode: M → U is a homomorphism then gives the commutativity of the above diagram, by the initiality of L. (Actually, to interpret this as "compiler correctness", one should also show that encode is injective, or else work with decode: U → M.) ADJ (1979) illustrated the 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 & Strachey (1971). They succeeded in giving a (very!) full proof of the correctness of a simple compiler: L → T.

We shall take a somewhat 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:

$$\begin{tikzcd}
 & L \arrow[dl, "\text{semantics}"'] \arrow[dr, "\text{compile}"] & \\
S \arrow[rr, "\text{implement}"'] \arrow[d, dashed] & & T \arrow[d, dashed] \\
M \arrow[rr, dashed] & & U
\end{tikzcd}$$

The models M and U are not relevant to the proof of the correctness of the implementation: S → T, but may aid the comparison of this diagram with the preceding one.

As with the earlier attacks on the compiler correctness problem, we shall regard the semantics and the compiler as homomorphisms on G-algebras, where G is the grammar of L. However, a crucial point is that with the present approach, the implementation of S by T can be proved correct before making S and T into G-algebras (one need only make T into an algebra with the same signature as S). Thus the proof is completely independent of the productions of G, in contrast to that of ADJ (1979). 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.

The abstract data types S and T will be specified equationally, enabling the use of the work on initial algebras, such as that by ADJ (1976), in proving our implementation of S by T correct. It is important to establish the "correctness" of these equational specifications, in order to see that the semantics: L $\rightarrow$ S is the intended semantics. However, this problem will be considered only briefly here, as it is independent of the proof of correctness of our implementation.

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 the 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 advocated 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" could form 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) also 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). The success of Milner (1979) in describing concurrency algebraically has provided some valuable guidelines for choosing semantic primitives.

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 semantic abstract data type S is described, possible models are discussed, and the standard semantics of L is 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.

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. 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!). Perhaps more important is that we apply the operators of the abstract syntax only to syntactic values, whereas in the pure algebraic notation, used by ADJ, one applies the semantic versions of the syntactic operators to semantic values - thereby hindering a "naive" reading of a semantic description.

(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

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

can be distributed freely around and between operands (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.

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 $\in$ 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

$$S <= f(S_1,\ldots,S_n)$$

- 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" will here be used synonymously with "abstract data type". 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			indices	
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	∈ {+, –, x}
		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 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.

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	( )	( )
$A' ; A''$	$\sigma A' \cdot \sigma A''$	$\tau A' \cdot \tau A''$
$V!$	( )	δV
$X.A'$	$\delta X \cdot \sigma A'$	$\tau A'$
$A' \underset{n}{\succ} A''$	$\sigma A' \cdot s''$	$t' \cdot \tau A''$
	where $\sigma A'' = d_1 \cdots d_n \cdot s''$,	and $\tau A' = d_1 \cdots d_n \cdot t'$
$tt?\,A' \,/\, ff?\,A''$	$T \cdot \sigma A'$	$\tau A'$
	where $\sigma A'' = \sigma A'$,	and $\tau A'' = \tau A'$
$fix\ Y.A'$	σY	τY
	where $\sigma A' = \sigma Y$	and $\tau A' = \tau Y$
Y	σY	τY
$contents_{id}$	( )	Z
$update_{id}$	Z	( )

action variables Y	source σA	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	

The following informal description of S may help the reader.

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.

Table 2 continued

equations

1.	$\text{skip} \; ; \; A = A$
2.	$A \; ; \; \text{skip} = A$
3.	$(A_1 \; ; \; A_2) \; ; \; A_3 = A_1 \; ; \; (A_2 \; ; \; A_3)$
4.	$V! \succ (X.\ A) = A \{X \leftarrow V\}$
5.	$(V! \; ; \; A_1) \succ_n A_2 = A_1 \succ_{n-1} (V! \succ_1 A_2)$
6.	$tt! \succ (tt?\ A_1 \ /\ ff?\ A_2) = A_1$
7.	$ff! \succ (tt?\ A_1 \ /\ ff?\ A_2) = A_2$
8.	$\text{fix}\ Y.\ A = A \{Y \leftarrow \text{fix}\ Y.\ A\}$
9.	$(V! \succ \text{update}_{id}) \; ; \; \text{contents}_{id} = (V! \succ \text{update}_{id}) \; ; \; V!$
10.	$(V! \succ \text{update}_{id}) \; ; \; \text{contents}_{id'} = \text{contents}_{id'} \; ; \; (V! \succ \text{update}_{id})$ 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 $\tau A_1 = (\delta X)$
18.	$\text{contents}_{id} \succ X.\ tt?\ A_1 \ /\ ff?\ A_2 = tt?\ \text{contents}_{id} \succ X.\ A_1 \ /\ ff?\ \text{contents}_{id} \succ X.\ A_2$

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. $A \Leftarrow$ skip is the null action, it is an identity for the sequencing operator $A \Leftarrow A' \; ; \; A''$. Note that sequencing is additive in the sources and targets. For example, if A' and A'' both consume one value, then $A' \; ; \; A''$ consumes two values.

The most basic action operator producing a value is $A \Leftarrow V!$. The consumption of a value is effected by $A \Leftarrow X.\ A'$, and X is bound to the consumed value in A'. To indicate that n values produced by one action are consumed by another, we have the operator $A \Leftarrow A' \succ_n A''$, and it is the first n values produced by A' which get consumed by A''. ($\succ_n$ will be written simply as $\succ$ when the value of n can be deduced from the context.) For example, consider $V! \succ X.\ A'$. The value V is produced (by $V!$), consumed (by $X.\ A'$) and then bound to X in A'. Free occurrences of X in A' just indicate where the consumed value is used.
In $(V_1! \; ; \; V_2!) \succ X_1.\ X_2.\ (X_2! \; ; \; X_1!)$, the values V_1, V_2 are produced, consumed and bound to X_1, X_2 and then produced again in reverse order – thus the net effect of the whole action is to produce two values.

$A \Leftarrow tt?\ A' \,/\, ff?\ A''$ 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.

$A \Leftarrow$ fix $Y.\ A'$ binds Y in A' and, together with $A \Leftarrow Y$, allows the expression of recursively-defined actions. Actually, it is used here (in describing L) only in a very limited form, corresponding to iteration: $A \Leftarrow$ fix a. $(A' \succ tt?\ A''; \ a / ff?\ \text{skip})$, 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. Note that update_{id} consumes a value, contents_{id} produces a value.

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 ($A \Leftarrow X.\ A'$ and $A \Leftarrow$ 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. (This is not immediately obvious, because one cannot define an operator in S to test whether a variable is free in an action. Fortunately, Berkling's (1976) idea of adding an "unbinding" operator to the λ-calculus leads to an equational specification of syntactic substitution in S.) Another way of treating binding operators will be discussed below.

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 canonical term algebra in effect specifies (possibly infinite) "normal forms" for arbitrary actions in S, and we are satisfied when these normal forms correspond to "fully evaluated" actions. (The potential infiniteness of actions suggests that the rational theories of ADJ (1976a) are the right setting for the meta-theory of abstract data types like S. The operator fix of S corresponds closely to the dagger of rational theories - when $\circ$ is substitution.)

The "obvious" Scott-model for S (corresponding to the M of ADJ (1979)) has a 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

$$[\mathrm{Env} \times d_1 \times \cdots \times d_m \rightarrow \mathrm{Env} \times d_1' \times \cdots \times d_n'],$$

where $\mathrm{Env} = \mathrm{Id} \rightarrow Z$. (Of course, one could also take a continuations-based model, or one with both static and dynamic environments, if preferred.)

However, S has binding operators, and terms can have "free" (semantic) variables. This raises the question of whether a modelling function from S to the Scott-model above 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 (1979) 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.

Semantics (sem)

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, was 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). As in the specification of S, it is assumed that bconst, prop, etc. vary over the same sets as in Table 1, thus giving families of equations.

Note that sem$\llbracket\ \rrbracket$ 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. The differences in appearance between ADJ's semantics and ours are due largely to ADJ's reliance on the operators of algebraic theories (tupling, projections, composition, iteration), whereas our S provides us with binding operators.

Table 3. Standard Semantics for L using S

operators	A <= sem$\llbracket$Cmd$\rrbracket$	$\sigma A = (\ )$,	$\tau A = (\ )$
	A <= sem$\llbracket$AExp$\rrbracket$	$\sigma A = (\ )$,	$\tau A = Z$
	A <= sem$\llbracket$BExp$\rrbracket$	$\sigma A = (\ )$,	$\tau A = T$

sem$\llbracket$Cmd$\rrbracket$ equations (id $\in$ Id)

sem$\llbracket$continue$\rrbracket$ = skip

sem$\llbracket$id := AExp$\rrbracket$ = sem$\llbracket$AExp$\rrbracket$ $\succ$ update$_{id}$

sem$\llbracket$if BExp then Cmd$_1$ else Cmd$_2\rrbracket$ = sem$\llbracket$BExp$\rrbracket$ $\succ$ tt? sem$\llbracket$Cmd$_1\rrbracket$ / ff? sem$\llbracket$Cmd$_2\rrbracket$

sem$\llbracket$while BExp do Cmd$\rrbracket$ = fix a. sem$\llbracket$BExp$\rrbracket$ $\succ$ tt? sem$\llbracket$Cmd$\rrbracket$; a / ff? skip

sem$\llbracket$AExp$\rrbracket$ equations

sem$\llbracket$aconst$\rrbracket$ = aconst !

sem$\llbracket$id$\rrbracket$ = contents$_{id}$

sem$\llbracket$aop1 AExp$\rrbracket$ = sem$\llbracket$AExp$\rrbracket$ $\succ$ z. (aop1 z) !

sem$\llbracket$AExp$_1$ aop2 AExp$_2\rrbracket$ = sem$\llbracket$AExp$_1\rrbracket$ $\succ$ z_1. sem$\llbracket$AExp$_2\rrbracket$ $\succ$ z_2. (z_1 aop2 z_2) !

sem$\llbracket$if BExp then AExp$_1$ else AExp$_2\rrbracket$ = sem$\llbracket$BExp$\rrbracket$ $\succ$ tt? sem$\llbracket$AExp$_1\rrbracket$ / ff? sem$\llbracket$AExp$_2\rrbracket$

sem$\llbracket$Cmd result AExp$\rrbracket$ = sem$\llbracket$Cmd$\rrbracket$; sem$\llbracket$AExp$\rrbracket$

sem$\llbracket$let id be AExp$_1$ in AExp$_2\rrbracket$ = contents$_{id}$ $\succ$ z_1. (sem$\llbracket$AExp$_1\rrbracket$ $\succ$ update$_{id}$);
sem$\llbracket$AExp$_2\rrbracket$ $\succ$ z_2. (z_1 ! $\succ$ update$_{id}$); z_2 !

sem$\llbracket$BExp$\rrbracket$ equations

sem$\llbracket$bconst$\rrbracket$ bconst !

sem$\llbracket$prop AExp$\rrbracket$ = sem$\llbracket$AExp$\rrbracket$ $\succ$ z. (prop z) !

sem$\llbracket$AExp$_1$ rel AExp$_2\rrbracket$ = sem$\llbracket$AExp$_1\rrbracket$ $\succ$ z_1. sem$\llbracket$AExp$_2\rrbracket$ $\succ$ z_2. (z_1 rel z_2) !

sem$\llbracket\neg$ BExp$\rrbracket$ = sem$\llbracket$BExp$\rrbracket$ $\succ$ tt? ff! / ff? tt!

sem$\llbracket$BExp$_1$ $\wedge$ BExp$_2\rrbracket$ = sem$\llbracket$BExp$_1\rrbracket$ $\succ$ tt? sem$\llbracket$BExp$_2\rrbracket$ / ff? ff!

sem$\llbracket$BExp$_1$ $\vee$ BExp$_2\rrbracket$ = sem$\llbracket$BExp$_1\rrbracket$ $\succ$ tt? tt! / ff? sem$\llbracket$BExp$_2\rrbracket$

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.

Actually, our T is not as general as ADJ's: they considered flow-charts of arbitrary shape, whereas we shall make do with "regular" flow-charts, corresponding to the "algebraic" flow diagrams of Scott (1970). This loss of generality doesn't seem to matter in connection with compiling L, which has no goto-command.

Table 4. Stack Theory T

sorts	(indices: $\delta \in \Delta$; $\tau \in \Delta^*$, 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
	$A' \underset{n}{\rightarrow} A''$	$\sigma A' \cdot s''$	$t' \cdot \tau A''$
		where $\sigma A'' = d_1 \cdots d_n \cdot s''$, and	$\tau A' = t' \cdot d_n \cdots d_1$
	tt? A' / ff? A''	$T \cdot \sigma A'$	$\tau A'$
		where $\sigma A' = \sigma A''$ and	$\tau A' = \tau A''$
	fix $Y.A'$	σY	τY
		where $\sigma A' = \sigma Y$ and	$\tau A' = \tau Y$
	Y	σY	τY
	contents_{id}	( )	Z
	update_{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	

A comparison of Tables 2 and 4 shows that T is rather similar to S. However, this should not be too surprising: many of 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 $A <= A' \underset{n}{>\!\!-} A''$ in S, but it is the <u>last</u> 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, very similar to those for S, which the operators of T must satisfy. The one equation which is crucially different is

(4.) $$(A_1 \, ; \, V!) \underset{n}{\rightarrow} A_2 = A_1 \underset{n-1}{\rightarrow} (V! \underset{1}{\rightarrow} A_2)$$

expressing a sort of associativity for $\rightarrow$, something which is lacking for $>\!\!-$ in S. In fact it is this property which allows us to think of terms in T as representing ordinary flow-charts.

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 $\rightarrow$ T, then let us say that imp is a <u>correct implementation</u> of S by T if it is an injective homomorphism (into the implicit derived algebra of T with the same signature as S). In other words, imp respects the equations of S: for any s, s' in S, $\text{imp}[\![s]\!] = \text{imp}[\![s']\!]$ iff s = s'. Having found such an imp, the composite imp $\circ$ sem: L $\rightarrow$ T is a correct compiler from L to T.

Unfortunately, it is actually impossible to implement S correctly 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 <u>unknown</u> 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 such a family in T.)

This is annoying, because one can easily implement the <u>closed</u> 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 <u>all</u> the terms of S, including those with free variables.

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'.

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'	$\delta X \cdot \sigma A'$	$\tau A'$
values V	**domain δV**	
V <= X	δX	
value variables X	**domain δX**	
X <= t	T	
t_n	T	
z	Z	
z_n	Z	

equations

1. $V! \rightarrow (X.A) = A\{X \leftarrow V\}$
2. switch $= z_1 .\ z_2 .\ (z_2! \ ;\ z_1!)$

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 that they are implemented by closed terms in Tx. It just 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 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: $\mathrm{up}^{d}_{()}$ and $\mathrm{down}^{d}_{()}$ have no effect, and may be removed.

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	$d \cdot d$
	$\mathrm{up}^{d}_{d_1 \cdots d_n}$	$d_n \cdots d_1 \cdot d$	$d_1 \cdots d_n \cdot d$
	$\mathrm{down}^{d}_{d_1 \cdots d_n}$	$d \cdot d_n \cdots d_1$	$d \cdot d_1 \cdots d_n$
	$\mathrm{flip}^{n}_{d_1 \cdots d_m}$	$d_m \cdots d_1$	$d_{n+1} \cdots d_m \cdot d_n \cdots d_1$

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

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

---

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

Our implementation of S by T seems to have two rather independent aspects: (i) the "serialization" of value terms in S into action sequences in T; (ii) the representation of the binding action A <= X. A' of S in T. The second part caused us considerably more trouble than the first. Whilst it might be tempting to use this as an excuse to throw the binding operators out of S, it would be prefereble to find a way of implementing binding more systematically than by the introduction of combinators.

At last we can implement S, by Tx'. The implementation function, imp: $S \to Tx'$, is defined in Table 7, using the same notation as was used for defining the semantics of L. S-operators now occur inside $\llbracket\ \rrbracket$ (in contrast to Table 2). As one can see, the implementation itself is really quite trivial: most of the operators go straight over from S to Tx'. The exceptions are value transfers $A \Leftarrow A' \succ A''$, which cause some "shuffling" on the stack; and the production of compound values $A \Leftarrow V!$, which get sequentialized.

Table 7. Implementation of S by Tx'

operators	$A \Leftarrow \mathrm{imp}\llbracket A' \rrbracket$	$\sigma A = \sigma A',\quad \tau A = \tau A'$
	$Y \Leftarrow \mathrm{imp}\llbracket Y' \rrbracket$	$\sigma Y = \sigma Y',\quad \tau Y = \tau Y'$
	$A \Leftarrow \mathrm{imp}\llbracket V \rrbracket$	$\sigma A = (\,),\quad \tau A = \delta V$
	$X \Leftarrow \mathrm{imp}\llbracket X' \rrbracket$	$\delta X = \delta X'$

$\mathrm{imp}\llbracket A \rrbracket$ equations

$\mathrm{imp}\llbracket \mathrm{skip} \rrbracket = \mathrm{skip}$

$\mathrm{imp}\llbracket A_1 ; A_2 \rrbracket = \mathrm{imp}\llbracket A_1 \rrbracket ; \mathrm{imp}\llbracket A_2 \rrbracket$

$\mathrm{imp}\llbracket V! \rrbracket = \mathrm{imp}\llbracket V \rrbracket$

$\mathrm{imp}\llbracket X.A \rrbracket = \mathrm{imp}\llbracket X \rrbracket .\ \mathrm{imp}\llbracket A \rrbracket$

$\mathrm{imp}\llbracket A_1 \succ_n A_2 \rrbracket = \mathrm{imp}\llbracket A_1 \rrbracket \to \mathrm{flip}^{n}_{\tau A_1} \underset{n}{\to} \mathrm{imp}\llbracket A_2 \rrbracket$

$\mathrm{imp}\llbracket tt?\ A_1 \,/\, ff?\ A_2 \rrbracket = tt?\ \mathrm{imp}\llbracket A_1 \rrbracket \,/\, ff?\ \mathrm{imp}\llbracket A_2 \rrbracket$

$\mathrm{imp}\llbracket \mathrm{fix}\ Y.A \rrbracket = \mathrm{fix}\ \mathrm{imp}\llbracket Y \rrbracket .\ \mathrm{imp}\llbracket A \rrbracket$

$\mathrm{imp}\llbracket Y \rrbracket = \mathrm{imp}\llbracket Y \rrbracket$ (the Y on the left is an action)

$\mathrm{imp}\llbracket \mathrm{contents}_{id} \rrbracket = \mathrm{contents}_{id}$

$\mathrm{imp}\llbracket \mathrm{update}_{id} \rrbracket = \mathrm{update}_{id}$

$\mathrm{imp}\llbracket V \rrbracket$ equations

$\mathrm{imp}\llbracket X \rrbracket = X!$

$\mathrm{imp}\llbracket \mathrm{aconst} \rrbracket = \mathrm{aconst}!$

$\mathrm{imp}\llbracket \mathrm{aop1}\ V \rrbracket = \mathrm{imp}\llbracket V \rrbracket \underset{1}{\to} \mathrm{aop1}$

$\mathrm{imp}\llbracket V_1\ \mathrm{aop2}\ V_2 \rrbracket = (\mathrm{imp}\llbracket V_1 \rrbracket ; \mathrm{imp}\llbracket V_2 \rrbracket) \underset{2}{\to} \mathrm{aop2}$

$\mathrm{imp}\llbracket \mathrm{bconst} \rrbracket = \mathrm{bconst}!$

$\mathrm{imp}\llbracket \mathrm{prop}\ V \rrbracket = \mathrm{imp}\llbracket V \rrbracket \underset{1}{\to} \mathrm{prop}$

$\mathrm{imp}\llbracket V_1\ \mathrm{rel}\ V_2 \rrbracket = (\mathrm{imp}\llbracket V_1 \rrbracket ; \mathrm{imp}\llbracket V_2 \rrbracket) \underset{2}{\to} \mathrm{rel}$

($\mathrm{imp}\llbracket X \rrbracket$, $\mathrm{imp}\llbracket Y \rrbracket$ are identities– equations omitted)

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.

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.

Table 8. Abstraction from Tx' to S

operators	A <= abs ⟦A'⟧	$\sigma A = \sigma A'$, $\tau A = \tau A'$
	Y <= abs ⟦Y'⟧	$\sigma Y = \sigma Y'$, $\tau Y = \tau Y'$
	V <= abs ⟦V'⟧	$\delta V = \delta V'$
	X <= abs ⟦X'⟧	$\delta X = \delta X'$

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

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

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

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

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

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

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. 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! $\rightarrow$ 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 $\rightarrow$ Tx' as imp $\circ$ sem, and, using the fact that imp: S $\rightarrow$ 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' $\underset{n}{\succ}$ A'' is only used for n = 1. It can be shown from the equations of Tx' that flip^{1}_{d} 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.

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!

Table 9. Compiler from L to T

operators	$A \Leftarrow \mathrm{comp}[\![\mathrm{Cmd}]\!]$	$\sigma A = (\,)$,	$\tau A = (\,)$
	$A \Leftarrow \mathrm{comp}[\![\mathrm{AExp}]\!]$	$\sigma A = (\,)$,	$\tau A = Z$
	$A \Leftarrow \mathrm{comp}[\![\mathrm{BExp}]\!]$	$\sigma A = (\,)$,	$\tau A = T$

comp⟦Cmd⟧ equations

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

$\mathrm{comp}[\![\mathrm{id} := \mathrm{AExp}]\!] = \mathrm{comp}[\![\mathrm{AExp}]\!] \rightarrow \mathrm{update}_{\mathrm{id}}$

$\mathrm{comp}[\![\mathrm{if\ BExp\ then\ Cmd}_1\ \mathrm{else\ Cmd}_2]\!] = \mathrm{comp}[\![\mathrm{BExp}]\!] \rightarrow \mathrm{tt?}\ \mathrm{comp}[\![\mathrm{Cmd}_1]\!]\ /\ \mathrm{ff?}\ \mathrm{comp}[\![\mathrm{Cmd}_2]\!]$

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

$\mathrm{comp}[\![\mathrm{while\ BExp\ do\ Cmd}]\!] = \mathrm{fix}\ a.\ \mathrm{comp}[\![\mathrm{BExp}]\!] \rightarrow \mathrm{tt?}\ \mathrm{comp}[\![\mathrm{Cmd}]\!]\ ;\ a\ /\ \mathrm{ff?}\ \mathrm{skip}$

comp⟦AExp⟧ equations

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

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

$\mathrm{comp}[\![\mathrm{aop1\ AExp}]\!] = \mathrm{comp}[\![\mathrm{AExp}]\!] \rightarrow \mathrm{aop1}$

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

$\mathrm{comp}[\![\mathrm{if\ BExp\ then\ AExp}_1\ \mathrm{else\ AExp}_2]\!] = \mathrm{comp}[\![\mathrm{BExp}]\!] \rightarrow \mathrm{tt?}\ \mathrm{comp}[\![\mathrm{AExp}_1]\!]\ /\ \mathrm{ff?}\ \mathrm{comp}[\![\mathrm{AExp}_2]\!]$

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

$\mathrm{comp}[\![\mathrm{let\ id\ be\ AExp}_1\ \mathrm{in\ AExp}_2]\!] = \mathrm{contents}_{\mathrm{id}} \rightarrow \mathrm{comp}[\![\mathrm{AExp}_1]\!] \rightarrow \mathrm{update}_{\mathrm{id}};$
$\qquad \mathrm{comp}[\![\mathrm{AExp}_2]\!] \rightarrow \mathrm{switch} \rightarrow \mathrm{update}_{\mathrm{id}}$

comp⟦BExp⟧ equations

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

$\mathrm{comp}[\![\mathrm{prop\ AExp}]\!] = \mathrm{comp}[\![\mathrm{AExp}]\!] \rightarrow \mathrm{prop}$

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

$\mathrm{comp}[\![\neg\ \mathrm{BExp}]\!] = \mathrm{comp}[\![\mathrm{BExp}]\!] \rightarrow \mathrm{tt?}\ \mathrm{ff!}\ /\ \mathrm{ff?}\ \mathrm{tt!}$

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

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

Finally, why did 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, one could conjecture that any 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 ($\subseteq$ {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) 68-85.

(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.

(1976a) "Rational algebraic theories and fixed-point solutions", Proc. 17th IEEE Symp. on Foundations of Computing, Houston, 1976.

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

Berkling, K.J.
(1976) "A symmetric complement to the lambda-calculus", Interner Bericht ISF-76-7, GMD-Bonn, 1976.

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 algebraich 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. CRSG-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.W. & Strachey, C.
1976) A Theory of Programming Language Semantics, Chapman & Hall (UK), John Wiley (USA), 1976.

Milner, R.
(1979) Algebraic Concurrency, unpublished lecture notes.

Morris, F.L.
(1973) "Advice on structuring compilers and proving them correct", Proc. ACM Symp. on Principles of Programming Languages, Boston, 1973.

Scott, D.S.
(1971) "The lattice of flow diagrams", Tech. Mono. PRG-3, Oxford Univ., 1971. Lect. Notes in Maths. 182: Semantics of Algorithmic Languages (E. Engeler, ed.), Springer, 1971.

Scott, D.S. & Strachey, C.
(1971) "Toward a mathematical semantics for computer languages", Tech. Mono. PRG-6, Oxford Univ., 1971. Computer and Automata (J. Fox, ed.), John Wiley, 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.