ISSN 0105-8517

CONTROL FLOW TREATMENT IN A SIMPLE SEMANTICS-DIRECTED COMPILER GENERATOR

by

Neil D. Jones and

Henning Christiansen

PB-137

September 1981

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

CONTROL FLOW TREATMENT IN A SIMPLE SEMANTICS-DIRECTED COMPILER GENERATOR

Henning Christiansen
Computer Science Department
Aarhus University
Ny Munkegade
8000 Aarhus C, Denmark

Neil D. Jones*
Computer Science Department
University of Copenhagen
Sigurdsgade 41
2200 Copenhagen N, Denmark

ABSTRACT

A simple algebra-based algorithm for compiler generation is described. Its input is a semantic definition of a programming language, and its output is a "compiling semantics" which maps each source program into a sequence of compile-time actions whose net effect on execution is the production of a semantically equivalent target program. The method does not require individual compiler correctness proofs or the construction of specialized target algebras.

Source program execution is assumed to proceed by performing a series of elementary actions on a runtime state. A semantic algebra is introduced to represent and manipulate possible execution sequences. A source semantic definition has two parts: A set of semantic equations mapping source programs into terms of the algebra, and an interpretation which gives concrete definitions of the state and the elementary actions on it.

Although simple, the input semantic definitions allow natural expression of a wide variety of programming language features including many control flow disciplines. Further, an extension method to allow the use of more expressive semantic algebras is outlined.

Target programs are essentially flow charts with "computed goto"; they contain little overhead beyond the elementary actions specified in the source semantics. Machine code may be generated by macro expansion or traditional code generation techniques. The method is well suited to bootstrapping - for example the generated compilers or the compiler generator itself can be output in flowchart form.

---

*): at Aarhus University until January 1, 1982.

1. EARLIER WORK IN SEMANTICS-DIRECTED COMPILER GENERATION

Milne and Strachey [MiS76] transformed a fairly large denotational semantic definition into a compiler; subsequent transformational approaches are described in [Bjø77], [Gan80], [Ras80], [Wan80a] and [Wan80b]. At present these methods still require considerable creativity and complex correctness proofs (one per compiler) and so have not yet yielded automatic compiler generators.

Methods based on partial evaluation of semantic equations (e. g. [Mos79], [Ers78], [San75], [Set81], [Tur80]) provide more uniformity and generality but have efficiency problems since compilation proceeds by transformation of expressions in the semantic definition language. Implementations of this approach include [Mos79] and [Set81].

The method of [JoS80] automatically generates correct compilers from denotational definitions using the lambda calculus but produces very inefficient target programs. The approach is to define a homomorphism mapping lambda expressions into target programs and to compose this with the semantic equations, yielding a homomorphism mapping source program parse trees directly into target programs. The inefficiency comes chiefly from the generality needed in the target code to implement the lambda calculus.

An important decision is the choice of language in which to write the semantic definition, since several of the foundational and efficiency problems of the methods above seem to stem from the use of the lambda calculus.

Mosses has written a series of papers ([Mos78], [Mos80], [Mos81]) advocating the use of algebraic semantic formalisms which express better than the lambda calculus the intuitive concepts naturally present in computation-sequencing, production and consumption of values, binding etc. The benefits include not only more readable semantic definitions but also greater modularity. Some related ideas appear in [RaS81]. As we shall see these alternate algebras are also well-suited to compiler generation (this point is also made in [Mos80]).

Algebraic Approaches to Compiling

A number of papers have been written concerning compiler correctness proofs and generation in algebraic frameworks ([McP67], [BuL69], [Mor73], [ADJ79], [Gan80], [Mos80], [RaS81]). We now briefly describe two which are related to the new method.

The left part of figure 1 comes from [Mor73] and [ADJ79]. L is a source language regarded as a G-algebra, where G is the context-free grammar defining the abstract syntax of L. M is a set containing denotations (meanings) of pieces of source programs. T is a collection of target programs, and U contains their denotations. By the "denotational assumption" the source semantic equations define a homomorphism from L into a G-algebra derived from M, and similarly for the target semantics.

$$\begin{tikzcd}
L \arrow[r, "\text{compile}"] \arrow[d, swap, "\text{source semantics (equations and model)}"] & T \arrow[d, "\text{target semantics}"] \\
M \arrow[r, "\text{encode}"] & U
\end{tikzcd}
\qquad
\begin{tikzcd}
 & L \arrow[dl, swap, "\text{semantic equations}"] \arrow[dr, dashed, "\text{generated compiler}"] & \\
S \arrow[rr, "\text{implement}"] \arrow[d, swap, "\text{semantic algebra model}"] & & T \arrow[d, "\text{target algebra model}"] \\
M \arrow[rr, "\text{encode}"] & & U
\end{tikzcd}$$

FIGURE 1. Compiler Correctness Diagrams

"Compile" is also assumed to be given homomorphically, so all four corners can be made into G algebras with T, M, U derived. Compiler "correctness" is shown by defining a mapping encode: $M \to U$ and showing commutativity of the resulting diagram. Since L is initial this may be done by showing encode to be a homomorphism.

In order to avoid trivial "correctness" in case T and U are one-point algebras it has been suggested that encode be injective. However, this is insufficient: Both commutativity and injectivity are consistent with an erroneous compiler which maps "+" into the subtraction operator of T, provided encode does the same on a semantic level. The problem is that with this approach both encode and compile are supplied by hand.

In the right of figure 1 (from [Mos80]) a semantic algebra S has been inserted between L and M. The semantic equations map parse trees into elements of S, an equationally-defined abstract data type (e.g. [ADJ78]) designed to contain natural semantic primitives suitable for expressing the semantics of a wide variety of source languages. The function "implement" represents S by the target algebra T, also an abstract data type. Compiler generation may be accomplished by composing the semantic equations with "implement" yielding a new set of semantic equations mapping parse trees directly into the target algebra. Models are not central to this approach, being needed only to show nontriviality of the abstract data types.

Overview of the New Approach

A common characteristic of the approaches above is that the target algebra T and its semantics are rather complex. One reason is inherent in the approach: T must have a sufficiently rich structure so that derived operations modelling the syntax operators of L can be defined; which in turn implies that corresponding operations must be defined on U. This can lead to some complex and unnatural semantic rules. For example (from [Mor73], p. 149)

> "To compute in a flowchart sewn together from pieces is to compute by turns in the pieces, jumping back and forth as often as one pleases at the stitches".

We see no good reason for introducing such complexities into the semantics of a target language which is typically intended to model machine code instructions.

To simplify T we introduce the traditional distinction between compile time and runtime. T is split into two parts - a <u>compiling algebra</u> C whose elements describe compile-time actions and whose operations can combine these actions according to source program syntax, and a much simpler target language T.

$$\begin{tikzcd}
 & L \arrow[dl, "\text{semantic equations}"', "\mathcal{S}"] \arrow[dr, dashed, "\text{generated compiler}", "\mathcal{S}^c"'] & \\
S \arrow[rr, "\text{cgen}"] \arrow[dddd, "\text{model}_I"] & & C \arrow[d, "\text{model}_J"] \\
 & & M_J \arrow[d, "\mathrm{UI}"] \\
 & & T \arrow[dd, "\text{run}_I"] \\
 & & \\
M_I \arrow[rr, "\text{encode}"] & & U_I
\end{tikzcd}$$

FIGURE 2. New Correctness Diagram

Figure 2 indicates that semantic equations $\mathcal{S}$ map source program parse trees into terms of a semantic algebra S. Interpretation I maps terms of S to their denotations via model_I: $S \rightarrow M_I$. In the bulk of this paper S will be extremely simple, specifying only control flow; consequently source program and target program denotations will be identical.

Generality is obtained in spite of the simplicity of S by use of the interpretation parameter I. This specifies the form of the runtime state and the meanings of the elementary actions appearing in $\mathcal{S}$ (these are constants in S). The interpretation also serves to define the meanings of target program instructions.

The compiler generator will transform a set of semantic equations $\mathcal{S}$ into a "compiling semantics" $\mathcal{S}^c$, which maps each source program parse tree into a sequence of compile-time actions. The effect of performing these actions is to produce a target program in T which is semantically equivalent to the source program under the runtime interpretation I. The compile-time interpretation J specifies the meanings of the compile-time actions.

$\mathcal{S}^c$ is constructed by composing $\mathcal{S}$: $L \rightarrow S$ with the compiler generation homomorphism cgen: $S \rightarrow C$. Correctness of $\mathcal{S}^c$ is expressed via commutativity of the diagram: For any interpretation I of $\mathcal{S}$ and for any term a in S of appropriate sort (letting $(f \circ g)x$ denote $g(f(x))$)

$$(\text{model}_I \circ \text{encode})a = (\text{cgen} \circ \text{model}_J \circ \text{run}_I)a$$

Practical use of $\mathcal{S}^C$ will be discussed in section 4. In most of this paper S will be very simple, but a way to extend S will be exemplified in section 5.

Elements of C are "compile-time actions" which can specify the generation of target instructions and the updating of a compile-time state. It is not required (as it was in figure 1) that every element of C have a concrete meaning in the target semantic model U. Consequently a compiling semantics can specify compile-time loops and multipass compilation, both of which seem difficult to express in previous algebraic frameworks.

2. THE TARGET LANGUAGE

A target program is a flow chart built up from instructions each of which changes the computation's state and/or point of control. Syntactically a program is very simple: a sequence

$$[0\colon \text{str}_0\ 1\colon \text{str}_1\ \ldots.\ r\colon \text{str}_r]$$

of labelled instruction streams, where each instruction specifies either an elementary action or a jump. Instruction goto (l , disp) jumps to the instruction at displacement disp from the start of stream l.

As in traditional machine code we optain great flexibility in control flow discipline by incorporating changes of control in the semantics of the instructions rather than in the topology of the flow chart. To do this we allow labels as parameters to instructions (constant parameters are also allowed). For example an interpretation could specify the effect of instruction cond (L1, L2) to be to jump to stream L1 or L2 depending on a value found in the state. Alternatively labels may be saved in the state and later fetched and branched to, allowing performance of subroutine calls, coroutine resumptions etc.

Following are two examples of code produced by a generated compiler. The instruction interpretations are found in table 1.

```
Code  [[ x + (if y then 7 else z) + 8 ]]
  =   [0 : find(x);  find(y);  cond(1,2);  plus;  load(8);  plus
       1 : load(7);  goto(0,3)
       2 : find(x);  goto(0,3) ]          both goto "plus ..." in stream 0
```

```
Code  [[ (λx. xx) (λy. y) 7 ]]
  =   [0 : pushclosure(1);  pushclosure(2);  apply;  load(7);  apply
       1 : bind(x);  bind(x);  find(x); apply;  return
       2 : bind(y);  find (y);  return ]
```

<u>Semantics</u>

A conventional continuation semantics will be used for flow charts; details are omitted for brevity. Every program label (l, disp) thus denotes a continuation ρ(l, disp) in CONT = [State $\rightarrow$ Answer]. Instructions will denote continuation transformers in CT = [CONT $\rightarrow$ CONT].

An <u>interpretation</u> is a tuple I = (State, σ_0, Answer, c_∞, α) where State and Answer are domains, the initial state is σ_0 and the final continuation is c_∞ (so a program's final answer is c_∞ (final state)). Further, α maps each instruction without parameters to α e in CT and each e(p_1, ... , p_n) to a function

$$\alpha\ e \in [D_1 \times \ldots \times D_n \rightarrow CT]$$

where D_1, ... , D_n are appropriate domains. If p_i is an atomic type (e.g. integer) then D_i is its natural domain and if p_i is a label then D_i = CONT.

For implementation we assume instruction meanings are given by equations of the form

$$(\alpha\ \text{instr})\ c\ \sigma \quad = \quad c'(\sigma')$$

which specifies how the current state σ and continuation c are transformed into their successors c' and σ', specified via expressions built from c and σ. The successor c' can be c (normal control flow), or a constant (e.g. normal termination or an error exit), or a component fetched from σ (e.g. from a "return address stack"), or it can be a computed value.

During program execution the label parameters of an instruction $e(p_1, \ldots, p_n)$ will be mapped to their denoted continuations so that α e may be applied.

Table 1 contains the interpretation used to support the semantics of table 2.

TABLE 1. An Interpretation I

State = Env x Value*	typical element:	$\sigma = (e, s)$
where Env = [Variable $\rightarrow$ Value]	typical element:	e
Value = Number + CONT x ENV	typical elements:	a, b
σ_0 = (empty environment, empty stack)		
Answer = Value		

Final Continuation: $c_\infty(e, a \cdot s) = a$ [answer = stack top]

Elementary Action Denotations

αplus c(e, b · a · s)	=	c(e, (a+b) · s)	[pop a, b, push a+b]
αload(con) c(e, s)	=	c(e, con · s)	[push con]
αfind(var) c(e, s)	=	c(e, e(var) · s)	[push value of var]
αcond(c_1, c_2) c(e, a · s)	=	(<u>if</u> a <u>then</u> c_1 <u>else</u> c_2) (e, s)	[conditional]
αbind(var) c (e, a · s)	=	c(e{a/var}, s)	[bind var to a]
αpushclosure(c_1) c(e, s)	=	c(e, (c_1, e_1) · s)	[push closure]
αapply c (e, a · (c_1, e_1) · s)	=	c_1(e, a · (c, e) · s)	[subroutine call]
αreturn c(e, a · (c_1, e_1) · s)	=	c_1(e_1, a · s)	[return]

<u>Informal Description of I</u>

I will be used for lambda expression evaluation. A state consists of a value stack and an environment binding variables to values. The net effect of evaluating an expression is to be to push its value on the stack, leaving the environment and the rest of the stack unchanged. A value is either a number or a closure (c, e) representing an abstraction and the values of its free variables.

<u>Implementation</u>

For machine implementation the state should be defined by first-order operations - e. g. arithmetic and selection and construction functions. Continuations as data objects can of course be replaced by labels, so application of a continuation amounts to a "computed goto".

Code generation appears to be easily accomplished by macro expansion - replacing each instruction e by a sequence of machine language instructions semantically equivalent to α e.

3. CONTROL FLOW SEMANTICS

A simple semantic algebra S will now be developed to represent and manipulate sequences of elementary actions. S may be regarded as a formalization of (continuation-based) "store semantics" as used in [MiS76] and by many others, or simply as a notation for expressing flow charts. In spite ot its simplicity surprisingly many programming language features can be naturally expressed via S. Five examples may be found in [Chr81] encompassing escapes, gotos, nested blocks, recursive procedures and coroutines. Section 5 will show how larger semantic algebras may be represented in S.

Table 2 contains semantic equations $\mathcal{S}$ mapping parse trees into S terms, followed by an informal description which assumes the interpretation of the previous section. Sequencing of actions is indicated by semicolon (a fuller description of S follows the example).

Such a set of semantic equations defines a "generalized homomorphism" mapping parse trees to S terms - generalized due to the presence of semantic functions $\mathcal{P}$, $\mathcal{E}$, etc. In this case $\mathcal{P}$ and $\mathcal{E}$ can be identified to yield a homomorphism h: L $\rightarrow$ S but such will not be the case if two semantic functions are applied to the same parse tree (e. g. L- and R-values of expressions).

TABLE 2. Semantic Equations $\mathcal{S}$

Source Syntax: program :: = exp

$$
\begin{aligned}
\text{exp} ::= {} & \text{con} \mid \text{var} \mid \text{exp}_1 + \text{exp}_2 \\
& \mid \underline{\text{if}}\ \text{exp}_1\ \underline{\text{then}}\ \text{exp}_2\ \underline{\text{else}}\ \text{exp}_3 \\
& \mid \text{exp}_1\ (\text{exp}_2) \mid \lambda\text{var}.\ \text{exp}_1
\end{aligned}
$$

Semantic Equations:

$\mathcal{P}$ $[\![$program$]\!]$	=	execute ($\mathcal{E}$ $[\![$program$]\!]$)
$\mathcal{E}$ $[\![$con$]\!]$	=	load (con)
$\mathcal{E}$ $[\![$var$]\!]$	=	find (var)
$\mathcal{E}$ $[\![\text{exp}_1 + \text{exp}_2]\!]$	=	$\mathcal{E}$ $[\![\text{exp}_1]\!]$; $\mathcal{E}$ $[\![\text{exp}_2]\!]$; plus
$\mathcal{E}$ $[\![$if exp_1 then exp_2 else $\text{exp}_3]\!]$	=	$\mathcal{E}$ $[\![\text{exp}_1]\!]$; cond ($\mathcal{E}$ $[\![\text{exp}_2]\!]$, $\mathcal{E}$ $[\![\text{exp}_3]\!]$)
$\mathcal{E}$ $[\![\text{exp}_1\ (\text{exp}_2)]\!]$	=	$\mathcal{E}$ $[\![\text{exp}_1]\!]$; $\mathcal{E}$ $[\![\text{exp}_2]\!]$; apply
$\mathcal{E}$ $[\![\lambda\ \text{var}.\ \text{exp}]\!]$	=	pushclosure (bind (var); $\mathcal{E}$ $[\![$exp$]\!]$; return)

Example Applications of $\mathcal{S}$:

$\mathcal{E}$ $[\![$x + (if y then 7 else 8) + z$]\!]$ = find(x); find(y); cond(load(1), find(z)); plus ; load(z); plus

$\mathcal{E}$ $[\![(\lambda x.\ xx)(\lambda y.\ y)7]\!]$ = pushclosure (bind(x); find(x); find(x); apply; return); pushclosure (bind(y); find(y); return); apply ; load(7) ; apply

<u>Informal Reading of Example Semantics</u>

The intended effect of performing $\mathcal{E}$ $[\![$ex$]\!]$ is to push the value of "exp" on the stack, leaving the environment unchanged. The first three $\mathcal{E}$ equations clearly accomplish this.

$\mathcal{E}$ $[\![$<u>if</u> ...$]\!]$ first pushes the value of exp_1, and cond pops this and activates $\mathcal{E}$ $[\![\text{exp}_2]\!]$ or $\mathcal{E}$ $[\![\text{exp}_3]\!]$. After the appropriate $\mathcal{E}$ $[\![\text{exp}_i]\!]$ actions are performed, execution will continue with the first action following $\mathcal{E}$ $[\![$<u>if</u> ...$]\!]$ (this follows from the way parameters are modelled - see the definition below of the model induced by an interpretation).

$\mathcal{E}[\![\lambda x. \exp]\!]$ will push a certain "delayed action" onto the stack. The equation for $\mathcal{E}[\![\exp_1 (\exp_2)]\!]$ resembles that for $\mathcal{E}[\![\exp_1 + \exp_2]\!]$ but "apply" is essentially different from "plus": pushes its successor action onto the stack and then activates the delayed action which must be the value of $\mathcal{E}[\![\exp_1]\!]$ (i.e. the value of some abstraction $\lambda x. \exp$). This will bind x to the value of $\exp_2$, evaluate the body exp in the updated environment (leaving the result on the stack, and finally return to the control point and environment which were saved by "apply".

The Semantic Algebra S

We assume familiarity with many-sorted algebras, e.g. [ADJ78]. S is the term algebra whose sorts and operations are given in table 3. The notation used resembles that of [Mos80] and [Mos81]. Operator symbols are written in "mixfix" notation (called "distributed-fix" in [Gog78]). This is a generalization of prefix, infix and postfix notation: Operator symbols can be distributed freely around and between operands. Further, the arity and coarity of an operator are indicated by the notation

$$S_0 \Leftarrow f(S_1, \ldots, S_n)$$

where f has arity $S_1 \times \ldots \times S_n$ and coarity S_0.

Each elementary action with parameters will have certain type requirements on its parameters (e.g. find(x) requires x to be a variable). These requirements are expressed by equipping each elementary action e with a list πe of its parameter types. The symbol Δ designates the set of possible parameter types.

TABLE 3. Signature of S

Types:	$\Delta = \{a\} \cup \{\text{integer, boolean, string, } \dots\}$			parameter types
Sorts:	ans			answers (values of entire source programs)
	e			elementary actions, each with parameter type list πe in Δ^*
	p			parameters, each with a target type $\tau p \in \Delta$
	a			actions
Operators:	ans		execute(a)	perform an action sequence
	a	$\Leftarrow$	skip	empty action
		$\mid$	$a_1; a_2$	sequencing – do a_1, then a_2
		$\mid$	e	elementary action without parameters
		$\mid$	$e(p_1, \dots, p_n)$	elementary action with parameters. Requirement: $\pi e = \tau p_1 \ \dots \ \tau p_n$
		$\mid$	$\underline{\text{fixtr}}\ L_1 = a_1, \dots, L_n = a_n\ \underline{\text{in}}\ a_0$	recursively defined action
		$\mid$	$\underline{\text{go}}\ L$	proceed to an action defined in an enclosing $\underline{\text{fixtr}}$
	p	$\Leftarrow$	con	constant of atomic sort as parameter. p has target τp = sort of con
		$\mid$	a	action as parameter, target $\tau p = a$

Informal Explanation

Terms of S describe the same computation sequences as flow charts, but in a more abstract way, so the operations may be seen as flow chart constructors. The equations given below are not definitions as in an abstract data type but rather theorems about equivalence in any model induced by an interpretation.

A program meaning will be a term of sort ans and must have the form execute(a) for any action a. An action may be the null action skip, an elementary action (possibly with parameters) whose meaning will be specified by an interpretation, or it may consist of two actions to be done in sequence. These operators satisfy

$$a;\ \text{skip} \equiv \text{skip};\ a \equiv a$$

$$(a_1;\ a_2);\ a_3 \equiv a_1;\ (a_2;\ a_3)$$

An action appearing as a parameter can be thought of as a data value representing a "delayed action". This value may be put into the store by an elementary action (e.g. pushed onto a return stack), and can at some later time be fetched from the store and executed. An elementary action may change the flow of control dynamically by selectively activating delayed actions, thus giving the effects of loops, conditional branches, subroutine or coroutine calls and resumptions etc.

Suppose an action parameter (e.g. a_1 in cond(a_1, a_2)) has been activated, and that execution of a_1 has been completed without a control transfer to another action via _go_. Control then passes to the action following the elementary action containing a_1 (e.g. the successor of cond(a_1, a_2)). Consequently a source-language _if_-statement might naturally be mapped into a term

evaltest; cond(thenaction, elseaction)

An action may also be defined recursively: a = (_fixtr_ $L_1 = a_1, \dots, L_n = a_n$ _in_ a_0). A free occurrence of _go_ L_i within one of $a_0, \dots, a_n$ indicates that control is to pass to a_i irrevocably, so the successor of _go_ L_i will be ignored. Each of $a_0, \dots, a_n$ is chained to the natural successor of a in the same way that action parameters are. Consequently a _while_-statement might naturally be mapped into a term

fixtr L = evaltest; cond(bodyaction; _go_ L , skip) _in_ _go_ L

Note: The special case of n = 1 and a_0 = _go_ L can be shortened to _fixtr_ $L_1 = a_1$, essentially the conventional syntax for _fix_. However, in general it should be clear that the actions of any flow chart in T can be described by an S term involving a single _fixtr_. Following are some equivalences valid in all interpretations.

1. _go_ L; a ≡ _go_ L
2. (_fixtr_ $L_1 = a_1, \dots, L_n = a_n$ _in_ a_0); a ≡ (_fixtr_ $L_1 = a_1$; a, ..., $L_n = a_n$; a) _in_ a_0; a

Equation 1 implies any S term may be reduced to an equivalent "tail recursive" form in which <u>go</u> L appears only in actions found at the ends of actions found in parameters or <u>fixtr</u> clauses. In such a term any reference to <u>go</u> L_i can be replaced by a_i without changing the meaning (provided no free variable references are captured). Equation 2 expresses the chaining of defined actions.

<u>Omission</u> For reasons of brevity we will not treat <u>fixtr</u> in the remainder of this paper. A full treatment involves the addition of semantic environments mapping labels to continuations in the definition of model_I and the addition of compile-time environments to compiler states.

<u>Models of S</u>

Given an interpretation $I = (\text{State}, \sigma_0, \text{Answer}, c_\infty, \alpha)$, one can extend α to give meaning to all terms of S. The set of such meanings can be made into an algebra M_I of the same signature as S by defining operations corresponding to the operations of S. Since S is initial, α is uniquely extendible to a homomorphism $\text{model}_I: S \rightarrow M_I$.

<u>TABLE 4. Model Algebra M_I</u>

Carriers:

Sort ans	:	Answer	
Sort a	:	$CT = [CONT \rightarrow CONT]$	
Sort e	:	$D_1 \times \ldots \times D_n \rightarrow CT$	$\left\{ \begin{array}{l} \text{for appropriate } D_1, \ldots, D_n \\ \text{if } \pi e = d_1 \ldots d_n \end{array} \right.$
Sort p	:	$CONT \rightarrow D$	if τp = atomic type D
Sort p	:	$CONT \rightarrow CONT$	if $\tau p = a$

Operations:

Arity and Co-Arity	Definition	
ans $\Leftarrow$ a	$\alpha ans = \alpha a\, c_\infty\, \sigma_0$	[final answer]
a $\Leftarrow$ skip	$(\alpha a)c = c$	[null action]
$\mid a_1; a_2$	$(\alpha a)c = (\alpha a_1)((\alpha a_2) c)$	[sequencing = composition]
$\mid$ e	$\alpha a = \alpha e$	[as found in I]
$\mid e(p_1, \ldots, p_n)$	$\alpha a = \alpha e(\alpha p_1 c, \ldots, \alpha p_n c)\, c$	[parameters chained to next c]
p $\Leftarrow$ con	$(\alpha p)c = con$	[constant parameter]
$\mid$ a	$(\alpha p)c = (\alpha a)c$	[action parameter]

4. COMPILER GENERATION

The Compiling Algebra and Its Interpretation

C is actually identical to S, making the method suitable for bootstrapping. However, the fixtr operator is not needed, and a fixed interpretation J will be used which specifies the compile-time state and the elementary actions needed to generate target code. For brevity J is only described informally; a more complete specification may be found in [ChJ81].

The compile-time state is very simple since only control flow is handled. Its components are: A partially generated program $\pi = [0: str_0 \ldots k: str_k]$ and two stacks, one for parameters and one for stream origins. The following elementary actions are enough to translate fixtr-free S terms into flow chart code:

addcode(ins)	- add a new instruction "ins" to the end of the instruction stream currently being generated
neworig	- establish the origin of a new stream (used when an action parameter is processed)
oldorig	- re-establish the old origin previously in use; further, save the new origin in a compile-time parameter stack p^*
push(atom)	- push an atomic parameter on the parameter stack p^*
$addcode_n(ins)$	- add instruction ins $(p_1, \ldots, p_n)$ to the current stream where $p_1, \ldots, p_n$ are parameters popped from p^*
goback	- end the current stream with an instruction "goto (destination)" to transfer control back to the stream previously in use

The CGEN Homomorphism

Table 5 contains a "compiler generation" homomorphism cgen: $S \rightarrow C$ mapping each operator of S into a derived operator on C. Let a be a term in S and cgen (a) its image in C. Then cgen (a) when interpreted by J will transform compile-time state $(\pi_1, \ldots)$ into $(\pi_2, \ldots)$ where π_2 equals π_1 augmented by instructions to perform a.

TABLE 5. Compiler Generation Function cgen: $S \rightarrow C$

S Operators		Derived Operators on C
ans	$\Leftarrow$ execute(a)	execute(cgen(a))
a	$\Leftarrow$ skip	skip
	$\mid a_1; a_2$	cgen(a_1); cgen(a_2)
	$\mid$ e	addcode(e)
	$\mid e(p_1, \ldots, p_n)$	cgen(p_1); ... cgen(p_n); addcode$_n$(e)
p	$\Leftarrow$ con	push(con)
	$\mid$ a	neworig; cgen(a); goback; oldorig

Correctness Theorem

Theorem Let I be any interpretation of S and ans any S term of the answer sort. Then

$$\text{model}_I\,(\text{ans}) = (\text{cgen} \circ \text{model}_J \circ \text{run}_I)\ \text{ans}$$

Proof is by structural induction on terms of S; it is omitted due to lack of space. A closely related proof may be found in [Chr81]. Note that ans must be of the form execute(a).

Application to Compiler Generation

A set of semantic equations defines a generalized homomorphism $\mathcal{S}$: syntax algebra $\rightarrow$ S. Suppose the initial equation of $\mathcal{S}$ has the form p$[\![$program$]\!]$ = execute(a). By the correctness theorem ($\mathcal{S} \circ$ cgen $\circ$ model$_J$) $[\![$program$]\!]$ is a flow chart equivalent to "program".

This expression represents a three step compilation process which can be reduced to two by constructing (at compiler generation time) a compiling semantics $\mathcal{S}^C = \mathcal{S} \circ$cgen. An example is found in Table 6.

TABLE 6. Compiling Semantics $\mathcal{S}^c$ Generated from Table 3

$$\begin{aligned}
\mathcal{P}^c[\![\text{program}]\!] &= \text{execute } (\mathcal{E}^c[\![\text{program}]\!]) \\
\mathcal{E}^c[\![\text{con}]\!] &= \text{push (con); addcode}_1\text{(load)} \\
\mathcal{E}^c[\![\text{var}]\!] &= \text{push (var); addcode}_1\text{(find)} \\
\mathcal{E}^c[\![\text{exp}_1 + \text{exp}_2]\!] &= \mathcal{E}^c[\![\text{exp}_1]\!];\ \mathcal{E}^c[\![\text{exp}_2]\!];\ \text{addcode (plus)} \\
\mathcal{E}^c[\![\text{if exp}_1 \text{ then exp}_2 \text{ else exp}_3]\!] &= \mathcal{E}^c[\![\text{exp}_1]\!]; \\
&\ \text{neworig; } \mathcal{E}^c[\![\text{exp}_2]\!]\text{; goback; oldorig;} \\
&\ \text{neworig; } \mathcal{E}^c[\![\text{exp}_3]\!]\text{; goback; oldorig;} \\
&\ \text{addcode}_2\text{ (cond)} \\
\mathcal{E}^c[\![\text{exp}_1 (\text{exp}_2)]\!] &= \mathcal{E}^c[\![\text{exp}_1]\!];\ \mathcal{E}^c[\![\text{exp}_2]\!];\ \text{addcode (apply)} \\
\mathcal{E}^c[\![\lambda\, x.\ \text{exp}]\!] &= \text{neworig;} \\
&\ \qquad \text{push (x); addcode}_1\text{(bind);} \\
&\ \qquad \mathcal{E}^c[\![\text{exp}]\!]; \\
&\ \qquad \text{addcode (return);} \\
&\ \text{oldorig;} \\
&\ \text{addcode}_1\text{ (pushclosure)}
\end{aligned}$$

Practical Use of a Compiling Semantics

We now show that compilation may be done in one step without construction of any C terms. The equations of table 6 all have a very simple form:

$$a^c[\![\text{syn } (t_1, \ldots, t_n)]\!] = b_1; b_2; \ldots ; b_n$$

where syn is a constructor from the source syntax algebra (i.e. a production), $t_1, \ldots, t_n$ are parse trees and each b_i is either an elementary compile-time action without subactions as parameters or has the form $\mathcal{B}^c[\![t_j]\!]$ for some t_j and semantic function $\mathcal{B}$. In other words $\mathcal{S}^c$ is a syntax-directed translation [AhU73].

Examination of table 5 reveals that $\mathcal{S}^c$ can always be put into this form (associativity of ; may be needed). The same holds true even when J and cgen are extended to handle fixtr, so we may assume a compiling semantics is always a syntax-directed translation.

Compilation can be done by traversing the parse tree according to $\mathcal{S}^c$, simultaneously performing any compile-time actions specified in $\mathcal{S}^c$. The syntax-directed translation will furthermore be <u>simple</u> if the order of subtree references in each semantic rule matches that of the corresponding production. In this case compiling can be overlapped with parsing. If the syntax is LL(k) the resulting algorithm bears a strong resemblance to a traditional handcrafted recursive descent compiler.

Final note: Suppose $\mathcal{S}$ contains no semantic rule which refers twice to the same syntax subtree. Then the subtree references on the right sides of $\mathcal{S}^c$ equations may be rearranged into the order found in the productions by the addition of <u>fixtr</u>. Consequently any semantic definition without multiple subtree references may be automatically converted into a one pass compiler.

<u>An Implementation</u>
An experimental LISP implementation of the method is currently being tested at Aarhus. An early use of the system was to produce a compiler generator which accepts as input a semantic definition and produces as output a compiler accepting source program parse trees. The generated compiler, its target programs and the compiler generator are all in flow chart form. Clearly the whole system could be "ported" to a new machine by writing routines for the elementary compiling and compiler generation actions (provided a parser is available).

5. LARGER SEMANTIC ALGEBRAS

Construction of a control flow semantics for a programming language places a considerable burden on the interpretation due to the simplicity of S. The kind of reasoning needed to write such a semantics is quite similar to that used in ordinary compiler design - for instance the invention of value and return address stacks and data access mechanisms. From this view S is a compiler writing language which can handle control flow and from which compilers' semantic components can automatically be produced.

It would clearly be desirable to have a more expressive, higher-level S, perhaps along the lines of [Mos80] and [Mos81]. One way to achieve this would be to redo sections 2 through 4, adding sorts and operations to S, extending the notion of interpretation and model_T, and augmenting the cgen function. A simpler alternative we briefly exemplify here is to implement such an extended algebra S^+ in terms of S, as shown in figure 3.

An implementation of S^+ by S will consist of a homomorphism ex: $S^+ \rightarrow S$, and a function (also named ex) which transforms an interpretation I^+ of S^+ into an appropriate interpretation $ex(I^+)$ of S. Compiler generation amounts to transforming an extended semantic definition $\mathcal{S}^+$ into

$$\begin{tikzcd}
 & L \arrow[dl, "\mathcal{S}^+"'] \arrow[dr, dashed, "\mathcal{S}^c"] & \\
S^+ \arrow[r, "ex"] & S \arrow[r, "cgen"] & C
\end{tikzcd}$$

FIGURE 3. Extension of S to S^+

$\mathcal{S}^c = (\mathcal{S}^+ \circ ex) \circ cgen = \mathcal{S}^+ \circ (ex \circ cgen)$. The second equality follows from associativity of (generalized) homomorphism composition and shows that the method of section 4 can be used, provided cgen is replaced by ex $\circ$ cgen.

Example Recursively Defined Actions

In this case S is augmented by adding a non-necessarily-tail-recursive fixpoint operator. The new operations are

$$\begin{aligned} a \;\Leftarrow\;& \underline{fix}\, A_i = a_1, \ldots, A_n = a_n \;\underline{in}\; a_0 \\ \mid\;& \underline{do}\; A_i \end{aligned}$$

The intention of $\underline{do}\ A_i$ is that a_i is to be performed, after which control will be returned to the successor of $\underline{do}\ A_i$. Implementation in terms of S is by a familiar device - adding a return address stack to the state of S^+ and treating $\underline{do}\ A_i$ as a call. The homomorphism ex: $S^+ \rightarrow S$ is given by:

1. $ex(\underline{fix}\ A_1 = a_1, \ldots, A_n = a_n\ \underline{in}\ a_0) = \underline{fixtr}\ A_1 = ex(a_1);\ return, \ldots,$
   $A_n = ex(a_0);\ return\ \underline{in}\ ex(a_0)$
2. $ex(\underline{do}\ A_i) = call\ (A_i)$
3. $ex(skip) = skip,\quad ex(a_1;\ a_2) = ex(a_1);\ ex(a_2),$ etc.

An interpretation of S^+ is the same from as one of S in this case; in general some extensions might use a substantially different form of interpretation. Given interpretation $I^+ = (\text{State}, \sigma_0, \text{Answer}, c_\infty^+, \alpha^+)$ of S^+ the corresponding interpretation of S is

$$ex(I^+) = (\text{State} \times \text{CONT}^*, (\sigma_0, \text{emptystack}), \text{Answer}, c_\infty, \alpha)$$

where $c_\infty((\sigma, r)) = c_\infty^+(\sigma)$ and α is defined as follows:

1. $(\alpha\ \text{call})(c_1)\ c(\sigma, r) = c_1(\sigma, c \bullet r)$ [push return address and jump]
2. $(\alpha\ \text{return})\ c(\sigma, c_1 \bullet r) = c_1(\sigma, r)$ [return]
3. Suppose $\alpha^+ e$ is defined by an equation [otherwise imitate α^+]
   $(\alpha^+ e)\ c\ \sigma = c'(\sigma')$ where c', σ' are expressions containing c and σ. Then α e is defined by the equation
   $(\alpha\ e)\ c(\sigma, r) = c'(\sigma', r)$

A similar implementation of the produced and consumed values of [Mos80] has been constructed (using a value stack), and further extensions to include binding and other primitives are being investigated.

The implementation concept is essentially the same as that of [Mos80] but with one simplification: Since we use term algebras rather than abstract data types such an implementation requires no correctness proof (since there are no equations to verify). It is simply a definition and as such may be used for compiler generation by forming $\mathcal{S}^+ \circ ex \circ cgen$.

Clearly, however, some implementations of $\mathcal{S}^+$ are "better" than others. For example one would hope the extension just given would satisfy a "loop unrolling property" so for instance <u>fix</u> A = a is equivalent to a{A ← <u>fix</u> A = a}

To define this "equivalence" more precisely, suppose S^+ is an extension of S and ex an implementation as above. Then an equation $a^+ \equiv b^+$ is taken to signify that for all interpretations I^+ of S^+ the following holds:

$$(ex \circ \text{model}_{ex(I^+)})\ a = (ex \circ \text{model}_{ex(I^+)})\ b$$

6. CONCLUSIONS

The method just described uses many-sorted algebras with operators natural for expressing primitive computing concepts, a clear formal separation between compile-time and runtime, and a simple target language which is a slight abstraction of traditional machine codes.

In contrast to [Mor73] and [ADJ80] the method applies uniformly to a class of semantic definitions (i.e. it is a compiler generator rather than a method for proving individual compilers correct). This property is shared by [Mos80], but our method uses term algebras instead of abstract data types, thus avoiding the need for proving correct the representation of one abstract data type by another. The approach resembles [Gau80] in that it is based on algebras and homomorphisms and has been implemented (at least a trial version); in contrast, however, our method seems to have firmer mathematical foundations and closer connections with traditional compiler methodology and target program structure.

Following are some directions for future work:

1. More powerful semantic algebras and implementation strategies are needed.
2. A general method to isolate the evaluation of static information (e.g. symbol tables) and move it into the compiler needs to be developed. This has mostly been done by ad hoc methods until now, although [Gan80] is an exception. It appears likely that flow analysis methods (e.g. [MuJ81]) can be applied to this problem.
3. A general method for transforming definitions of elementary actions into code generation modules should be developed.

Finally, it should be mentioned that the method cannot handle all programming language constructs. Counter examples include self-modification, concurrency and parallelism.

Acknowledgements

Discussions with Peter Mosses, Gordon Plotkin, Flemming Nielson, Brian Mayoh, Hanne Riis and others at the University of Aarhus have been very helpful during the development of these ideas.

REFERENCES

ADA80 Formal definition of the ADA programming language, preliminary version for public review, Honeywell Inc., Cii Honeywell Bull, INRIA France (1980).

ADJ77 Goguen, J.A., J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, JACM 24, 68-95 (1977).

ADJ78 Goguen, J.A., J.W. Thatcher, E.G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in Current Trends in Programming Methodology vol IV (ed. R.T. Yeh) Prentice-Hall, Englewood Cliffs, N.J. (1978).

ADJ79 Thatcher, J.W., E.G. Wagner, J.B. Wright, More on advice on structuring compilers and proving them correct, Proc. 6th ICALP, Springer LNCS 74, Berlin (1979).

AhU73 Aho, A.V., J.D. Ullman, The Theory of Parsing, Translation and Compiling vol. II, Prentice-Hall, Englewood Cliffs, N.J. (1973).

Bjø77 Bjørner, D. Formal development of interpreters and compilers, Danmarks Tekniske Højskole ID673 (1977).

BuL69 Burstall, R.M., P.J. Landin, Programs and their proofs: an algebraic approach, Machine Intelligence 4 (1969).

Cat79 Cattel, R.G.G., J.M. Newcomer, B.W. Leverett, Code generation in a machine-independent compiler, SIGPLAN Notices, vol. 14, no. 8 (1979).

Chr81 Christiansen, H. A New Approach to Compiler Generation, Master's thesis in computer science, Aarhus University (1981).

Don79 Donegan, M.K., R.E. Noonan, S. Feycock, A code generator language, SIGPLAN Notices, vol. 14, no. 8 (1979).

EKM79 Eriksen, S.H., B.B. Jensen, B.B. Kristensen, O.L. Madsen, The BOBS-system, Aarhus University, DAIMI report PB-71 (1979).

Ers78 Ershov, A.P. On the essence of computation, in Formal Description of Programming Language Concepts, ed. E.J. Neuholdt, North-Holland (1978).

Gan77 Ganzinger, H., K. Ripken, R. Wilhelm, Automatic generation of optimizing multipass compilers, in Proc. IFIP Congress 77, Toronto, ed. B. Gillchrist, pp. 535-540, New York: North-Holland (1977).

Gan80 Ganzinger, H. Transforming denotational semantics into practical attribute grammars, in [Jon80], pp. 1-69.

Gau80 Gaudel, M.C. Specification of compilers as abstract data type representations, in [Jon80], pp. 140-164.

Gog78 Goguen, J.A. Order sorted algebras: exceptions and error sorts, coercions and overloaded operators, Semantics and Theory of Comp. Rpt. 14, UCLA (1978).

Gor79 Gordon, M.J.C. The Denotational Description of Programming Languages, An Introduction. Springer Verlag, Berlin (1979).

Joh74 Johnson, S.C. YACC - yet another compiler compiler, CSTR32, Bell Laboratories, Murray Hill, N.J.

Jon80	Jones, N.D. (editor) Semantics-Directed Compiler Generation. Lecture Notes in Computer Science 94, Springer-Verlag, Berlin (1980).
JoS80	Jones, N.D., D.A. Schmidt, Compiler generation from denotational semantics. Found in [Jon80], pp. 70-93.
Knu68	Knuth, D.E. Semantics of context-free languages, Math. Syst. Theory 2, no. 2, pp. 127-145 (1968).
Lor75	Lorho, B. Semantic attributes processing in the system DELTA, in Methods of Algorithmic Language Implementation (C.H.A. Koster ed.), pp. 21-40, Springer LNCS 47, Berlin (1977).
Luc69	Lucas, P., K. Walk, On the formal description of PL/I, Annual Review in Automatic Programming, vol. 6, pt. 3, Pergamon Press (1969).
McK70	McKeeman, W.M., J.J. Horning, D.B. Wortman, A Compiler Generator, Prentice-Hall, Englewood Cliffs, N.J. (1970).
McP67	McCarthy, J., J. Painter, Correctness of a compiler for arithmetic expressions, Proc. Symp. Appl. Math. 19, pp. 33-41 (1967).
MiS76	Milne, R.W., C. Strachey, A Theory of Programming Language Semantics, Chapman and Hall (UK), John Wiley (USA) (1976).
Mor73	Morris, F.L. Advice on structuring compilers and proving them correct, Proc. ACM Symp. Prin. Prog. Langs., Boston, pp. 144-152 (1973).
Mos74	Mosses, P. The Mathematical semantics of Algol 60. PRG-12, Oxford University Programming Research Group (1974).
Mos78	Mosses, P. Modular denotational semantics, draft paper, Computer Science Department, Aarhus University (1978).
Mos79	Mosses, P. SIS - Semantics Implementation System, Reference Manual and Users Guide. Aarhus University, DAIMI MD-30 (1979).
Mos80	Mosses, P. A constructive approach to compiler correctness. Found in: [Jon80], pp. 189-210.
Mos81	Mosses, P. A semantic algebra for binding constructs. In Formalization of Programming Concepts, LNCS 107, Springer-Verlag, pp. 408-418 (1981).
MuJ81	Muchnick, S., N.D. Jones, Program Flow Analysis, Prentice-Hall, Englewood Cliffs, N.J. (1981).
Oll75	Ollongren, A. A Definition of Programming Languages by Interpreting Automata, Academic Press (1975).
Plo81	Plotkin, G. A Structural Approach to Operational Semantics, Lecture notes, Computer Science Department, Aarhus University (1981).
Ras80	Raskovsky, M., P. Collier, From standard to implementation denotational semantics. Found in [Jon80], pp. 94-139.
RiM81	Riis, H., M. Madsen, The NEATS System, Computer Science Department, Aarhus University (1981).
Räi80	Räihä, K.-J. Experiences with the compiler writing system HLP, in [Jon80], pp. 350-362.
RaS81	Raoult, J.-C., R. Sethi, On meta languages for a compiler generator: Properties of a notation for combining functions, Preprint, Bell Laboratories (1981)

San75 Sandewall, E. Ideas about management of LISP data bases, 4th IJCAI, Tbilisi, vol. 2, pp. 585-592 (1975).

Sco76 Scott, D. Data types as lattices, SIAM Journal on Computing, vol. 5, no. 3, pp. 522-587 (1976).

ScS71 Scott, D., C. Strachey, Towards a mathematical semantics for computer languages, in Proc. Symp. "Computer and Automata", MRI Symposia, vol. XXI, Polytechnic Inst. of Brooklyn, N.Y. (1971).

Set81 Sethi, R., Circular expressions: Elimination of static environments, in Proc. 8th ICALP, Springer LNCS (1981)

Sto77 Stoy, J.E. Denotational semantics: The Scott-Strachey approach to programming language theory. MIT Press (1977).

Ten76 Tennent, R.D. The denotational semantics of programming languages, CACM vol. 19, no. 8 (1976).

Tur80 Turchin, V.F. Semantic definitions in REFAL and the automatic production of compilers, in [Jon80], pp. 441-474.

Wan80a Wand, M. Deriving target code as a representation of continuation semantics. Indiana University, Comp. Sc. Dept., Technical Report no. 94 (1980).

Wan80b Wand, M. Different advice on structuring compilers and proving them correct. Indiana University, Comp. Sc. Dept., Technical Report no. 95 (1980).