ISSN 0105-8517

Mathematical Foundation of

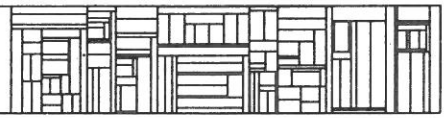
A SEMANTICS DIRECTED COMPILER GENERATOR

by

Henning Christiansen
and
Neil D. Jones

DAIMI PB-148
July 1982

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



PB-148

Mathematical Foundation of

A SEMANTICS DIRECTED COMPILER GENERATOR

- technical details and
  correctness proof

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

This paper describes technical details which were not included in the paper "Control Flow Treatment in a Simple Semantics-Directed Compiler Generator", [ChJ82], due to lack of space. The present paper cannot be read separately.

We describe our method in full detail, including the iterate-operator in the S-algebra. Furthermore we give complete formal descriptions of the semantic algebra S and its models (Section 1), the compiler generation function d^C and the compile time interpretation

1. Control Flow Semantics

This section describes the signature of the semantic algebra S (identical to the one given in [ChJ82]) and the model algebra M_I is extended to handle the iterate-construction.

Semantic Algebra S

The signature of S is given in table 1, for an explanation is referred to [ChJ82].

Table 1. Signature of S

Types:	Δ = {a} $\cup$ {integer, boolean, string, ...} 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 $\Leftarrow$ 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
	$\mid e(p_1,..,p_n)$	elementary action with parameters. Requirement: $\Pi e = \tau p_1 \ldots \tau p_n$
	$\mid$ iterate $L_1 = a_1, \ldots, L_n = a_n$ in a_0	

Models of S

The semantics of S is a traditional continuation model parameterized by a language dependent interpretation, as described in [ChJ82].

Given an interpretation I = (State, start, finish, α), we can define an extension, $\bar{\alpha}$, of α giving 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. The model of [ChJ82] has been extended with a domain of semantic environments, Menv, to hold meanings of labels introduced by the iterate-construct.

Note: we will write α instead of $\bar{\alpha}$.

Table 2. Model Algebra M_I

Carriers:

Sort ans: $\Sigma^* \xrightarrow{P} \Sigma$

Sort a : Menv → CT

Sort e : $D_1 \times \ldots \times D_n \rightarrow$ CT if e has parameter domains $D_1, \ldots, D_n$

Sort p : Cont → D if τp = atomic type D

Sort p : CT if τp = a

where Cont = State → Σ continuations

CT = Cont → Cont continuation transformers

Menv = S-labels → Cont semantic environments

Let $menv_0$ be the empty environment.

Operations:

2. Compiler Generation

The Compile Time Algebra C

C is actually identical to S. However, the iterate 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.

In order to handle iterate-labels, the interpretation, J, of C is equipped with a compile time environment mapping semantic labels into corresponding target program labels. The scope rules for semantic labels are realized by utilizing a stack of environments, one for each nested iterate-construction.

Compared with the descriptions in [ChJ82], C has been extended with a number of elementary actions to control these environments, so we give an informal description of C and its interpretation before we give a complete formal definition.

The compile time state is relatively simple since only control flow is handled. Its components are:

- a partially generated program $\pi = [0: str_0 \ldots k: str_k]$,
- a parameter stack,
- a stack for stream origins,
- a stack of compile time environments giving for each label occurrence in the semantic algebra a corresponding target program label.

oldstream - re-establish the old origin previously in use

pushstream - push the stream origin currently in use on the parameter stack

push(atom) - push an atomic parameter on the parameter stack

addcode$_n$(ins) - add instruction ins(p_1,...,p_n) to the current stream where p_1,...,p_n are parameters popped from the parameter stack

goback, goback-1 - end the current stream with an instruction "goto 1" to transfer control back to the stream previously in use; the difference between the two is very subtle and is not described here

newen(L_1,...,L_n) - allocate n new streams and bind them to L_1,...,L_n in the compile time environment stack

oldenv - pop a segment from the compile time environment stack, thus reestablishing previous bindings

nextstream - advance to the next L. stream when genera-

The compiler generation homorphism d^C

Table 3 contains a compiler generation homomorphism d^C in $\begin{array}{|c|}\hline S \to T \\ \hline \wedge \\ C \\ \hline \end{array}$. Let be a term in S and Dd^Ca its image in C. Then Dd^Ca 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 3. Compile Generation Definition d^C

d^C [[execute(a)]] = execute(d^C[[a]] ; addcode(finish))

d^C [[skip]] = skip

d^C [[a_1;a_2]] = d^C[[a_1]] ; d^C[[a_2]]

d^C [[e]] = addcode(e)

d^C [[e(p_1,..., p_n)]] = d^C[[p_1]] ;...; d^C[[p_n]] ; $addcode_n$(e)

d^C [[iterate L_1 = a_1,..., L_n = a_n in a_0]]

= newenv(L_1,...,L_n) ; d^C[[a_0]] ;
newstream; d^C [[a_1]] ; goback-1;
nextstream; d^C[[a_2]] ; goback-1;
⋮
nextstream; d^C[[a_n]]; goback-1;
oldstream; oldenv

d^C [[go L]] = addgo(L)

d^C [[con]] = con

d^C [[apar(a)]] = newstream; d^C[[a]]; goback;
pushstream; oldstream

Table 4. Compile Time Interpretation

J = (State, start, finish, α) where
State = Program × P* × Stream-origin* × Cenv*
P = Stream-origin + Atomic-domains (parameter domain)
Cenv = S-labels → Stream-origin

Notation: upd(π,l,i) = π, with instruction i added to the end of its l'th stream.
open(π,n) = π, with n new empty streams following its last stream
<> empty stack and empty program

start: → state, initial state
start = (open(<>,1), <>, <0>, <>)
finish: Cont = State → Program, final continuation
finish(π,p*,l*,cenv*) = π

α, defining meanings of elementary compile time actions is defined as follows:

αaddcode(i) c(π,p*,l·l*,cenv*)
= c(upd(π,l,i),p*,l·l*,cenv*)

αnewstream c(π,p*,l*,cenv*)
= c(open(π,1),p*,(k+1)·l*,cenv*)
where π = [0: str_0 ... k: str_k]

αoldstream c(π,p*,l·l*,cenv*)
= c(π,p*,l*,cenv*)

Table 4. Compile Time Interpretation, contd.

αgoback-1 $c(\pi,p^*,k\cdot l\cdot l^*,cenv^*)$
$= c(upd(\pi,k,goto(l,i)),p^*,k\cdot l\cdot l^*,cenv^*)$
where i = length of the l'th stream of π

αnewenv$(L_1,\ldots,L_n)$ $c(\pi,p^*,l^*,cenv^*)$
$= c(open(\pi,n),p^*,l^*,cenv\cdot cenv^*)$
where $cenv = [L_1 \rightarrow (k+1,0),\ldots,L_n \rightarrow (k+n,0)]$
$\alpha = [0: stream_0;\ldots;k: stream_k]$

αoldenv $c(\pi,p^*,l^*,cenv\cdot cenv^*)$
$= c(\pi,p^*,l^*,cenv^*)$

αnextstream $c(\pi,p^*,l\cdot l^*,cenv^*)$
$= c(\pi,p^*,(l+1)\cdot l^*,cenv^*)$

αaddgo(L) $c(\pi,p^*,l\cdot l^*,cenv^*)$
$= c(upd(\pi,l,goto(find(L,cenv^*))),p^*,l\cdot l^*,cenv^*)$

where $find(L,cenv.cenv^*) = \begin{cases} cenv\,L & \text{if } L \in domain(cenv) \\ find(cenv^*) & \text{otherwise} \end{cases}$

3. Target Language Semantics

In [ChJ82] the semantics of the target language T is only described informally.

ing to

execute(find(x); test; load(2), load(3); load(4); plus))

is: 0: find(x); test((1,0), (2,0)); finish

1: load(2); goto(0,2)

2: load(3); load(4); plus; goto(0,2)

Given an interpretation I = (State, start, finish, α) it is a straightforward matter to define a target program semantic function

run_I: T $\rightarrow$ Σ* $\rightarrow$ Σ

This semantics maps instruction streams to continuation transformers and uses a runtime environment ρ to map each label (l, disp) to a continuation ρ(l,disp).

Such a semantics is unsatisfactory for direct implementation due to the presence of functional objects (the continuations) in the runtime state's data structures. This can be overcome by a simple strategy - namely to replace continuations by labels. This strategy works if we can assume that the only operations allowed on continuations are to copy them from parameters, to copy them to and from data structures and to apply them to new states, i.e. they may not be created. This implies that all runtime continuations are the meanings of label parameters, so the same effects are achieved by copying labels instead and implementing the applications of a continuation by a "computed goto".

In [Chr81] it is shown how the semantics described here can be transformed into another more concrete and operational.

Given an interpretation I = (State, start, finish, α) we say that run_I is the target language semantics induced by I. A formal definition of run_I is given in table 5. We use start to set up the initial state. The output from the program is "taken out" of the state by the final continuation finish. Note that in equation (I6) of table 5 the symbol "finish" is used to denote both the instruction finish (cf. table 3) and the final continuation given by α. The semantics is written as a traditional denotational semantics, e.g. [Sto77].

Table 5. Target Language Semantics

The target language semantics, run_I, induced by an interpretation I = (State, start, finish, α) is defined as follows:

Semantic Domains

State given by α

Cont = State → Σ

CT = Cont → Cont

Tenv = T-label → Cont

T-label = Stream-origin × Displacement

Syntactic Domains

Program ::= Stream-origin : Stream

Table 5. Target Language Semantics, contd.

Notation

To simplify notation we define:

$[\![program]\!] = [\![0: stream_0, 1: stream_1, \ldots; r: stream_r]\!]$ *)

$n_i = [length\ of\ stream_i]-1, \quad i = 1,\ldots,r$

$(l_0,\ldots,l_m) = ((0,0),(0,1),\ldots,(0,n_0)$
$(1,0),(1,1),\ldots,(1,n_1)$
$\vdots$
$(r,0),(r,1),\ldots,(r,n_r)$

where $m = n_0 + n_1 + \ldots + n_r + r$

$[\![program \downarrow_j]\!]$ denotes the j'th ,j = 0,...,m, instruction in $[\![program]\!]$ as indicated above.

c_{dummy} is a continuation that is never applied

Semantic Functions

run_I : Program → Σ* → Σ
$\mathbb{R}$: Program → Tenv
$\mathbb{I}$: [Ins + Stream] → Tenv → CT

Semantic Equations

(P1) $run_I [\![program]\!] x^* = \mathbb{R}[\![program]\!]\ (0,0)(start\ x^*)$

(R1) $\mathbb{R}[\![program]\!] = tenv$ where

$(tenv, c_0,\ldots,c_m) = fix\ \lambda(tenv,c_0,\ldots,c_m).$
$([l_0 \rightarrow c_0,\ldots,l_m \rightarrow c_m],$
$\mathbb{I}[\![program \downarrow 0]\!]\ tenv\ c_1, \mathbb{I}[\![program \downarrow 1]\!]\ tenv\ c_2,\ldots,$
$\mathbb{I}[\![program \downarrow m-1]\!]\ tenv\ c_m, \mathbb{I}[\![program \downarrow m]\!]\ tenv\ c_{dummy})$

(I1) $\mathbb{I}[\![1: stream]\!] = \mathbb{I}[\![stream]\!]$

4. Proof of Correctness of Compiler Generation

To show that the compiler generator is correct, we have to verify conditions I and II of Theorem 1 in [ChJ82]. Condition I was proved in [ChJ82] and this section will concentrate on condition II. Condition II states that compiler generation homomorphism d^C describes a correct compiler for terms of the semantic algebra S of sort ans, or in other words that the meanings of a term of sort ans and the corresponding target program are identical. We express this formally in a theorem.

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

$$model_I(ans) = run_I(model_J(Dd^C ans))$$

In the rest of this section let ans = execute (a_π) for some action a_π and t = $model_J(Dd^C ans)$ be the corresponding target program. The target semantic environment in which t is executed is denoted ρ, and is defined as $\rho = \mathbb{R}[\![t]\!]$, cf. table 5.

We prove the theorem by induction over the structure of a_π, using the fact that each subterm of a_π is mapped into a contigous sequence of instructions in one of t's streams.

In the proof we need to relate subterms of a to corresponding compiler state information and pieces of target code. Due to the fact that an action (say plus) may occur several times in a_ we cannot

We now describe three occurrence-indexed objects used in the proof. First we give an informal description, then a formal definition. The definitions is not used directly in the proof, but only indirectly through a few simple properties stated after the definitions.

$entry^{occ}$ is the label of the first instruction of the t code for a^{occ}

$exit^{occ}$ is the label of the first instruction after the t code for a^{occ}

$cenv^{occ}$ each a^{occ} is compiled with a unique stack of compile time environments cenv* which we "merge" into one environment $cenv^{occ}$.

Formal definitions of $entry^{occ}$, $exit^{occ}$, and $cenv^{occ}$

Lemma 1 For any action-sorted term a of S there is function f_a: $State_J \to State_J$ such that for any $Cont_J$,c, and $State_J$, S:

$$model_J(Dd^c\ a)\ menv_0\ c\ s = c(f_a s)$$

Further, if $s = (\pi,p^*,l^*,cenv^*)$ then $f_a(s) = (\pi',p^*,l^*,cenv^*)$, where each stream of π is a prefix of the corresponding stream of π. Notation: $menv_0$ is the empty semantic environment (since d^c does not apply iterate), the subscript J is used for objects defined by the compile time interpretation J.

For each occurrence in a_π we let the state $state^{occ} \in State_J$ be the (clearly unique) state entered just before processing a^{occ}. This can easily be defined precisely following d^c. For example

$$\begin{array}{lll} & state^0 & = start_J \\ \text{if} & a^{occ} & = a_1;a_2 \text{ then} \\ & state^{occ.1} & = state^{occ} \text{ and} \\ & state^{occ.2} & = f_{a_1}(state^{occ.1}) \end{array}$$

and so forth. Details are straightforward and so are omitted.

Now suppose $state^{occ} = (\pi, p^*, l{\cdot}l^*, e_1{\cdot}e_2{\cdot}\ldots{\cdot}e_n)$, and $f_{a^{occ}}(state^{occ}) = (\pi', p^*, l{\cdot}l^*, e_1{\cdot}e_2\ldots{\cdot}e_n)$. We can now define

$$\begin{aligned} cenv^{occ} &= e_n[e_{n-1}]\ldots[e_2][e_1] \\ entry^{occ} &= (l, \text{ length of stream } l \text{ of } \pi) \\ exit^{occ} &= (l, \text{ length of stream } l \text{ of } \pi') \end{aligned}$$

Properties of $cenv^{occ}$, $entry^{occ}$ and $exit^{occ}$

(CE0) $cenv^0$ = the empty compile time environment

(CE1) if $a^{occ} = a^{occ.1}; a^{occ.1}$ then
$cenv^{occ.1} = cenv^{occ.2} = cenv^{occ}$

(CE2) if $a^{occ} = e(\ldots,a^{occ.i},\ldots)$ then
$cenv^{occ.i} = cenv^{occ}$

(CE3) if $a^{occ} = \underline{\text{iterate}}\ L_1 = a^{occ.1},\ldots L_n = a^{occ.1}\ \underline{\text{in}}\ a^{occ.0}$ then
$cenv^{occ.i} = cenv^{occ}[L_1 \to entry^{occ.1},\ldots,L_n \to entry^{occ.n}]_{i=0,\ldots,n}$

(L2) if $a^{occ} = e(...,a^{occ.i},...)$ then
$\rho\ exit^{occ.i} = \rho\ exit^{occ}$

(L3) if $a^{occ} = \underline{iterate}\ L_1 = a_1,...,L_n = a_n\ \underline{in}\ a_0$ then
$entry^{occ.0} = entry^{occ}$
$exit^{occ.0} = exit^{occ}$
$\rho\ exit^{occ.i} = \rho\ exit^{occ}$ $\quad i = 0,...,n$

The equations involving ρ are consequences of the fact that code for action parameters and semantic label definitions is nested to whatever follows the construct in which they occur. This nesting is realized by the instructions goback and goback-1 (cf. table 3) which emits goto's to the desired destinations. The remaining properties are simple consequences of the definitions of $entry^{occ}$ and $exit^{occ}$.

(C0) $\rho\ exit^{0} = finish_J$

(C1) if $a^{occ} = e$ then
$(\alpha e)(\rho\ exit^{occ}) = \rho\ entry^{occ}$

(C2) if $a^{occ} = e(...,a^{occ.i},...,const_j,...)$ then
$(\alpha e)(...,\alpha a^{occ.i} cenv^{occ.i},...,const_j,...)(\rho\ exit^{occ})$
$= \rho\ entry^{occ}$

(C3) if $a^{occ} = \underline{go}\ L$ then
$\rho\ entry^{occ} = \rho(cenv^{occ}\ L)$

Follow from inspection of the target language semantics (table 5) and table 3, d^c.

Using the definitions of $cenv^{occ}$, $entry^{occ}$, and $exit^{occ}$ and the

For each step in the proof of Lemma 2 and the correctness theorem we note (in curly brackets) which properties or equations that are used. A, CE, L, and C properties are defined in this section, I, P, and R refer to equations in the target language semantics (table 5) and M to equations for the model algebra M_I (table 2). When we use the induction hypothesis for some occurrence we write this occurrence among the other properties.

Proof of Correctness Theorem, assuming Lemma 2

Let x* be any expression in Σ^*, I = (State, start, finish, α) any interpretation, then

$$\begin{aligned} \text{model}_I(\text{ans}) &= \alpha a^0\ \text{menv}_0\ \text{finish}\ (\text{start}\ x^*) && \{A0, M0\} \\ &= \alpha a^0\ (\text{cenv}^0 \circ \rho)(\rho\ \text{exit}^0)(\text{start}\ x^*) && \{CE0, C0\} \\ &= (\rho\ \text{entry}^0)(\text{start}\ x^*) && \{\text{Lemma 2}\} \\ &= \text{run}_I(t)\ x^* && \{P1, R1\} \end{aligned}$$

Proof of Lemma 2

The induction goes as follows

a^{occ} = skip

trivial! {M1,L0}

$a^{occ} = a^{occ.1};\ a^{occ.2}$

$a^{occ}(\text{cenv}^{occ} \circ \rho)(\rho\ \text{exit}^{occ})$

$\underline{a^{occ} = e}$

$\alpha a^{occ}(cenv^{occ} \circ \rho)(\rho\ exit^{occ})$

$= \alpha e(\rho\ exit^{occ})$ {M3,L2}

$= \rho\ entry^{occ}$ {C1}

$\underline{a^{occ} = e(p_1,\ldots,p_n)}$

For simplicity we let $a^{occ} = e(a^{occ.1},27)$

$\alpha a^{occ}(cenv^{occ} \circ \rho)(\rho\ exit^{occ})$

$= \alpha e(\alpha a^{occ.1}(cenv^{occ.1} \circ \rho)(\rho\ exit^{occ}),27)$ {M4,M7,M8,CE2,L2}

$= \rho\ entry^{occ}$ {C2}

$\underline{a^{occ} = \underline{iterate}\ L_1 = a_1,\ldots,L_n = a_n\ \underline{in}\ a_0}$

$\alpha a^{occ}(cenv^{occ} \circ \rho)(\rho\ exit^{occ})$

$= \alpha a^{occ.0} env(\rho\ exit^{occ.0})$ {L3,M5}

where $env = (cenv^{occ} \circ \rho)[L_1 \rightarrow \alpha a^{occ.1} env(\rho\ exit^{occ}),\ldots]$
If we can show $env = cenv^{occ.0} \circ \rho$ we get the desired result
from {occ.0,L3}

$cenv^{occ.0} \circ \rho$

$= (cenv^{occ}[L_1 \rightarrow entry^{occ.1},\ldots] \circ \rho$ {CE3}

$= (cenv^{occ} \circ \rho)[L_1 \rightarrow \alpha a^{occ.1}(cenv^{occ.1} \circ \rho)(\rho\ exit^{occ.1}),\ldots]$ {occ.1,...}

$= (cenv^{occ} \circ \rho)[L_1 \rightarrow \alpha a^{occ.1}(cenv^{occ.0} \circ \rho)(\rho\ exit^{occ}),\ldots]$ {CE3,L3}

We see that $cenv^{occ.0} \circ \rho$ satisfy the equation defining env

We have now proved Lemma 2 and hence the correctness theorem.

We conclude this section with an interesting corollary which makes it possible to use alternate compiler generation homomorphisms d^C and compile time algebra C without constructing a new correctness proof.

Corollary A compiler generation homomorphism $d^{C'}$ and compile time algebra C' with interpretation J', that satisfies properties (CE0) - (CE3), (L0) - (L3), and (C0) - (C3) will generate correct compilers.

Proof The properties of d^C, C, and J used in the proof of Lemma 2 are captured in the list described above.

REFERENCES

ChJ82 Christiansen, H. and Jones, N. Control Flow Treatment in a Simple Semantics-Directed Compiler Generator, Formal Description of Programming Concepts II, IFIP IC-2 Working Conference, Proceedings, North-Holland Publishing Company, Amsterdam (1982).

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

Chr82 Christiansen, H. Users manual for CERES, Aarhus University, DAIMI MD- (1982).