Different Advice on Structuring Compilers and Proving Them Correct

by

Mitchell Wand
Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 95

DIFFERENT ADVICE ON STRUCTURING COMPILERS AND PROVING THEM CORRECT

MITCHELL WAND

SEPTEMBER, 1980

This material is based upon work supported by the National Science Foundation under Grant MCS79-04183.

Different Advice on Structuring Compilers And Proving Them Correct

ABSTRACT: Our advice is this: the semantics of a language is itself an algebra, in which there are sorts for both syntactic and semantic objects; the semantic functions or valuations are among the operations of this algebra. An implementation of a language is a representation of the semantic algebra in the sense of Hoare. By the use of associative and distributive laws, one obtains representations of continuation functions which look like machine code; the function which simulates the application of continuations to states looks like a conventional stack machine. We prove the correctness of such an implementation for the language given in L. Morris's 1973 paper.

Different Advice on Structuring Compilers and Proving Them Correct

1. Introduction

The conventional advice about the structure of compiler correctness proofs is that the source language is an initial algebra whose signature is specified by its grammar, and that the source semantics, compiler, target semantics, and coding or decoding functions (relating source and target meanings) are (or should be) homomorphisms between algebras of this signature. This point of view was stated in various forms [1, 7, 10, 21]. Our purpose is to give somewhat different advice.

Our advice is this: the semantics of a language is itself an algebra, in which there are sorts for both syntactic and semantic objects; the semantic functions or valuations are among the operations of this algebra. An implementation of a language, whether it be an interpreter or a compiler, is a representation of the semantic algebra in the sense of Hoare [6]. To prove the correctness of such a representation, one must give an "abstraction function" which is a homomorphism from the implementation algebra to the semantic algebra. The initiality of the source language is still important, however, since it is used in the construction of the abstraction function.

The diagram for this notion of correctness is shown in Figure 1.1. We are using continuation semantics, so that the domain of "source meanings" consists of functions from states to answers. An important operation in the semantic algebra is therefore "apply," which applies these functions to states, yielding answers.

In the implementation algebra, we assume that source programs are represented by themselves, although we could instead choose to let the so-called source programs, which are actually parse trees, be represented by strings of characters, and label the leftmost vertical arrow with "parse." We have chosen to let states and answers represent themselves, but this assumption is also unnecessary.

The key choice in the diagram is the representation of source meanings. Reynolds [15] essentially used the programs themselves as representations, letting ϕ = "source semantics"; then the operation labelled "target machine" became an interpreter. In [28], we showed how we could introduce special-purpose combinators to eliminate bound variables and use associativity to reduce trees of combinators to an almost-linear format. The resulting trees of combinators have an obvious interpretation as source meanings and look like code for a virtual machine of the kind often considered in programming languages [14]. The function which simulates application, taking representations of source meanings and states to yield answers, looks like a typical stack machine with a standard fetch-execute cycle.

In this paper, we actually carry through the proof of correctness for the language in Lockwood Morris's 1973 paper [10]. The proof that the left-hand square of Figure 1.1

commutes is easy. The proof that the target machine is partially correct is also easy, using subgoal or fixpoint induction. The hardest part of the proof is showing that the target machine halts sufficiently often. For this we adapt Plotkin's proof of the completeness of his operational semantics for LCF [13].

Section 2 presents the language, using Morris's direct semantics, and converts it to continuation semantics. In section 3, we derive the representations of source meanings and the compiler. In section 4, we derive the target machine and prove its correctness. In section 5, we paste these results together to match the methodology we have presented. In section 6, we give a more leisurely tour of the antecedents of this work and present some extensions and conclusions.

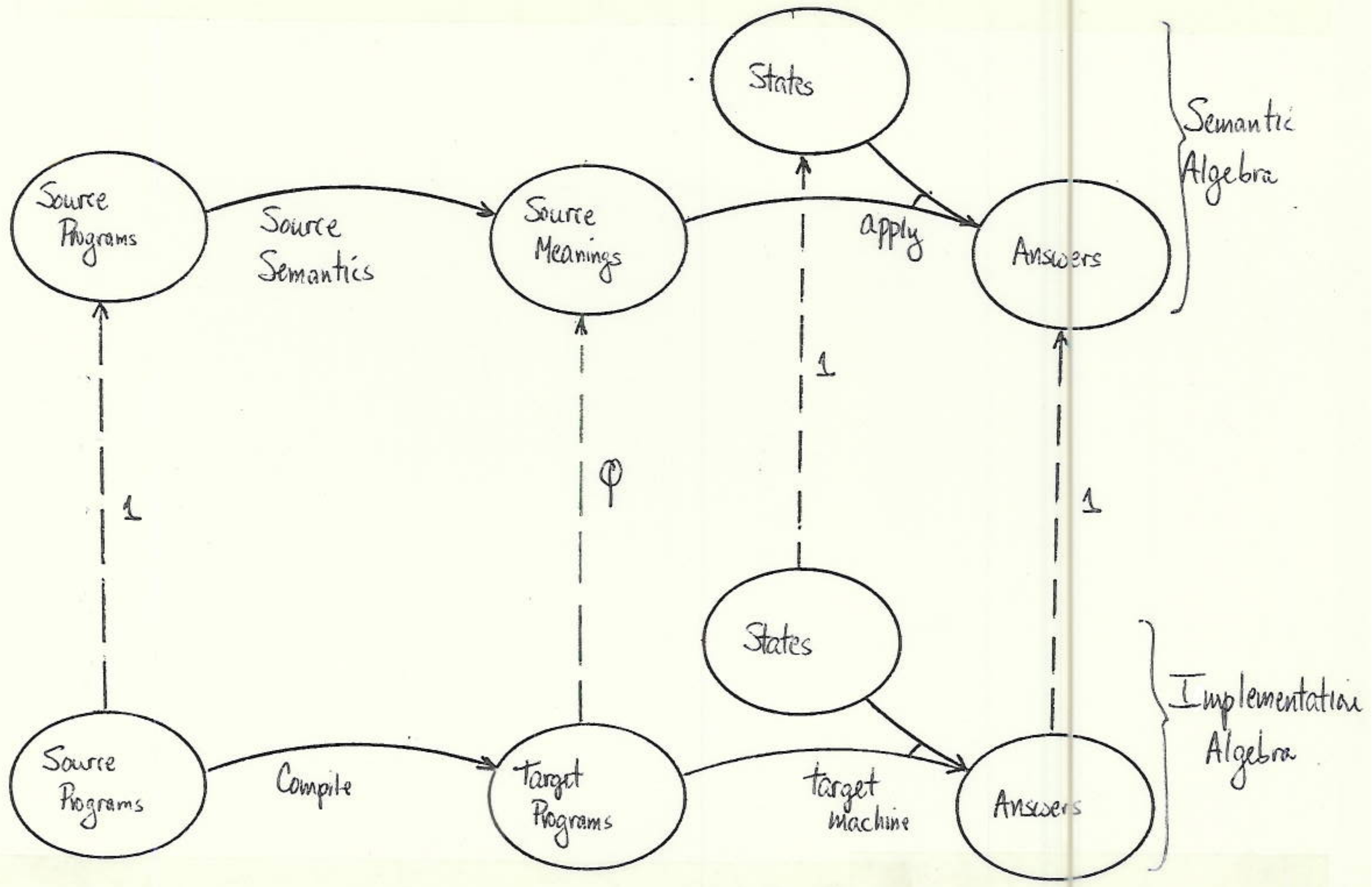


Figure 1.1 Master Commuting Diagram

2. The Language

Table 2.1 shows our version of the language.(*) Since [10, 12, 21] differ on several details, we have had to make some choices. N is the cpo of the integers with $\perp$. Id is an unspecified flat domain of identifiers. B is the 3 point cpo of booleans. The commands are standard, and include compounds, which were absent from [10]. Following [10], we have included addition as the only arithmetic operator; clearly nothing is to be learned by generalizing further. We have included conditional expressions, which were not in [10]. Again, we have simplified the syntax of boolean expressions to equality and conjunction. Additional relational operators would add nothing, but we have retained the "short-cutting" evaluation rule for conjunction.

The semantics follows Morris's. It is a direct semantics. We use Morris's *-notation:

$$p*q = \lambda\sigma.p(q\sigma\downarrow 1)(q\sigma\downarrow 2)$$

Here the $\downarrow$ is used to select elements from a tuple; application takes precedence over projection, so $q\sigma\downarrow 1$ means $(q\sigma)\downarrow 1$. We use three valuations: $\mathcal{DC}$, $\mathcal{DA}$, $\mathcal{DB}$, for Direct semantics of Commands, Arithmetic expressions, and Boolean expressions. $b \rightarrow x,y$ denotes the strict conditional defined by

$$\begin{aligned} tt\rightarrow x,y &= x \\ ff\rightarrow x,y &= y \\ \perp\rightarrow x,y &= \perp. \end{aligned}$$

(*) Tables and figures appear at the end of the sections which refer to them.

Y denotes the minimal-fixed-point operator, and $\sigma[a/I]$ denotes

$$\lambda J.(J=I)\rightarrow a,\sigma J$$

as usual; this operation takes precedence over application, so $\eta\sigma[a/I]$ denotes $\eta(\sigma[a/I])$. $<-,->$ denotes the pairing operator. With this notation, our transcription is very close to Morris's original. We use "state" for Morris's "environment" as more in keeping with current usage.

Since this is a direct semantics, our first task is to convert it to a continuation semantics. The techniques for this are fairly well-known [17, 20, 24, 26]. Table 2.3 gives the continuation semantics. $L \rightarrow_{\perp} M$ denotes the cpo of all strict continuous functions from L to M, i.e., those continuous $f:L\rightarrow M$ such that $f\perp=\perp$. In the absence of procedures, it is easy to prove the required equivalence result, using techniques such as those in [17, 26]. We first need some lemmata.

<u>Lemma 2.1</u>

(i) If η is strict, so is $\mathit{Semc}[\![\text{cmd}]\!]\eta$

(ii) If κ is strict, so is $\mathit{Sema}[\![\text{ae}]\!]\kappa$

(iii) If κ' is strict, so is $\mathit{Semb}[\![\text{be}]\!]\kappa'$.

<u>Proof</u>: By structural induction, noting that $(\lambda b.b \rightarrow f,g)$ is always strict, and if $f_0 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq \ldots$ are all strict, so is $\bigcup_n f_n$. $\square$

<u>Lemma 2.2</u> $\eta\circ(f^*g) = (\eta\circ f)^*g$. $\square$

<u>Lemma 2.3</u> If η is strict, then $\eta\circ(\lambda b.b\rightarrow f,g) = \lambda b.b\rightarrow\eta\circ f,\eta\circ g$. $\square$

<u>Lemma 2.4</u> If η is strict, and for all f, $\eta\circ\phi f = \psi(\eta\circ f)$, then $\eta\circ Y\phi = Y\psi$.

Proof: See [17]. □

Theorem 2.1 If η and κ are strict, then

(i) $\mathit{Semc}[\![\mathrm{cmd}]\!]\eta = \eta \circ \mathcal{DC}[\![\mathrm{cmd}]\!]$

(ii) $\mathit{Sema}[\![\mathrm{ae}]\!]\kappa = \kappa^* \mathcal{DA}[\![\mathrm{ae}]\!]$

(iii) $\mathit{Semb}[\![\mathrm{be}]\!]\kappa = \kappa^* \mathcal{DB}[\![\mathrm{be}]\!]$

Proof: See Appendix. □

Corollary. $\mathit{Semp}[\![\mathrm{cmd}]\!] = \mathcal{DC}[\![\mathrm{cmd}]\!]$. □

There is more than this, of course, in converting from direct to continuation semantics, e.g. when procedures are involved. Our methodology starts with a continuation semantics; we have done the translation explicitly because of our desire to be faithful to the benchmark of [10].

Domains

```
n:N            (⊥,0,1,2,...)
I:Id
σ:S = Id→N
b:B            Booleans (⊥,tt,ff)
```

Syntax

```
Cmd ::= continue
        Id := Ae
        if Be then Cmd else Cmd
        Cmd ; Cmd
        while Be do Cmd

Ae  ::= 0 | 1
        Id
        Ae plus Ae
        Cmd result Ae
        let Id be Ae in Ae
        if Be then Ae else Ae

Be  ::= Ae = Ae
        Be and Be
```

Table 2.1 Syntax of the Language

<u>Valuations</u>

$$\mathcal{D}C : \mathrm{Cmd} \to \mathrm{S} \to \mathrm{S}$$

$$\mathcal{D}A : \mathrm{Ae} \to \mathrm{S} \to \mathrm{N} \times \mathrm{S}$$

$$\mathcal{D}B : \mathrm{Be} \to \mathrm{S} \to \mathrm{B} \times \mathrm{S}$$

<u>Equations</u>:

$$\mathcal{D}C\llbracket \underline{\mathtt{continue}} \rrbracket \sigma = \sigma$$

$$\mathcal{D}C\llbracket \mathtt{I{:=}ae} \rrbracket = (\lambda v.\sigma[v/I])*\mathcal{D}A\llbracket \mathtt{ae} \rrbracket$$

$$\mathcal{D}C\llbracket \mathtt{if\ be}\ \underline{\mathtt{then}}\ \mathtt{cmd1}\ \underline{\mathtt{else}}\ \mathtt{cmd2} \rrbracket = (\lambda b.b \to \mathcal{D}C\llbracket \mathtt{cmd1} \rrbracket, \mathcal{D}C\llbracket \mathtt{cmd2} \rrbracket)*\mathcal{D}B\llbracket \mathtt{be} \rrbracket$$

$$\mathcal{D}C\llbracket \mathtt{cmd1;cmd2} \rrbracket = \mathcal{D}C\llbracket \mathtt{cmd2} \rrbracket \circ \mathcal{D}C\llbracket \mathtt{cmd1} \rrbracket$$

$$\mathcal{D}C\llbracket \underline{\mathtt{while}}\ \mathtt{be}\ \underline{\mathtt{do}}\ \mathtt{cmd} \rrbracket = Y(\lambda f.(\lambda b.b \to f \circ \mathcal{D}C\llbracket \mathtt{cmd} \rrbracket, \lambda\sigma.\sigma)*\mathcal{D}B\llbracket \mathtt{be} \rrbracket)$$

$$\mathcal{D}A\llbracket 0 \rrbracket = \lambda\sigma.<0,\sigma>$$

$$\mathcal{D}A\llbracket 1 \rrbracket = \lambda\sigma.<1,\sigma>$$

$$\mathcal{D}A\llbracket I \rrbracket = \lambda\sigma.<\sigma I,\sigma>$$

$$\mathcal{D}A\llbracket \mathtt{ae1}\ \underline{\mathtt{plus}}\ \mathtt{ae2} \rrbracket = (\lambda v_1.(\lambda v_2\sigma.<v_1+v_2,\sigma>)*\mathcal{D}A\llbracket \mathtt{ae2} \rrbracket)*\mathcal{D}A\llbracket \mathtt{ae1} \rrbracket$$

$$\mathcal{D}A\llbracket \mathtt{cmd}\ \underline{\mathtt{res}}\ \mathtt{ae} \rrbracket = \mathcal{D}A\llbracket \mathtt{ae} \rrbracket \circ \mathcal{D}E\llbracket \mathtt{cmd} \rrbracket$$

$$\mathcal{D}A\llbracket \underline{\mathtt{let}}\ \mathtt{I}\ \underline{\mathtt{be}}\ \mathtt{ae1}\ \underline{\mathtt{in}}\ \mathtt{ae2} \rrbracket =$$

$$(\lambda v_1\sigma_1.((\lambda v_2\sigma_2.<v_2,\sigma_2[\sigma_1 I/I]>)*\mathcal{D}A\llbracket \mathtt{ae2} \rrbracket)(\sigma_1[v_1/I]))*\mathcal{D}A\llbracket \mathtt{ae1} \rrbracket$$

$$\mathcal{D}B\llbracket \mathtt{ae1 = ae2} \rrbracket = (\lambda v_1.(\lambda v_2\sigma_2<v_1=v_2,\sigma_2>)*\mathcal{D}A\llbracket \mathtt{ae2} \rrbracket)*\mathcal{D}A\llbracket \mathtt{ae1} \rrbracket$$

$$\mathcal{D}B\llbracket \mathtt{be1}\ \underline{\mathtt{and}}\ \mathtt{be2} \rrbracket = (\lambda b.b \to \mathcal{D}B\llbracket \mathtt{be}_2 \rrbracket, (\lambda\sigma.<ff,\sigma>)*\mathcal{D}B\llbracket \mathtt{be1} \rrbracket$$

Table 2.2 Direct Semantics

Domains

n: N	Integers
I: Id	Identifiers
σ: $S=Id\rightarrow N$	States
b: B	Booleans
η: $C=S\rightarrow_{\perp} N$	Command continuations
κ: $K=N\rightarrow_{\perp} C$	Expression continuations
κ: $BC=B\rightarrow_{\perp} C$	Boolean continuations

Valuations

Semp: $\mathrm{Cmd}\rightarrow C$

Semc: $\mathrm{Cmd}\rightarrow C\rightarrow C$

Sema: $\mathrm{Ae}\rightarrow K\rightarrow C$

Semb: $\mathrm{Be}\rightarrow BC\rightarrow C$

Equations

$$\mathit{Semp}[\![\mathrm{cmd}]\!] = \mathit{Semc}[\![\mathrm{cmd}]\!](\lambda\sigma.\sigma)$$

$$\mathit{Semc}[\![\underline{\mathrm{continue}}]\!] = \lambda\eta.\eta$$

$$\mathit{Semc}[\![I{:=}\mathrm{ae}]\!] = \lambda\eta.\mathit{Sema}[\![\mathrm{ae}]\!](\lambda v\sigma.\eta\sigma[v/I])$$

$$\mathit{Semc}[\![\underline{\mathrm{if}}\ \mathrm{be}\ \underline{\mathrm{then}}\ \mathrm{cmd1}\ \underline{\mathrm{else}}\ \mathrm{cmd2}]\!] =$$

$$\lambda\eta.\mathit{Semb}[\![\mathrm{be}]\!](\lambda b.b\rightarrow\mathit{Semc}[\![\mathrm{cmd1}]\!]\eta,\mathit{Semc}[\![\mathrm{cmd2}]\!]\eta)$$

$$\mathit{Semc}[\![\mathrm{cmd1};\ \mathrm{cmd2}]\!] = \lambda\eta.\mathit{Semc}[\![\mathrm{cmd1}]\!](\mathit{Semc}[\![\mathrm{cmd2}]\!]\eta)$$

$$\mathit{Semc}[\![\underline{\mathrm{while}}\ \mathrm{be}\ \underline{\mathrm{do}}\ \mathrm{cmd}]\!] = \lambda\eta.Y(\lambda\theta.\mathit{Semb}[\![\mathrm{be}]\!](\lambda b.b\rightarrow\mathit{Semc}[\![\mathrm{cmd}]\!]\theta,\eta))$$

$$\mathit{Sema}[\![0]\!] = \lambda\kappa.\kappa 0$$

$$\mathit{Sema}[\![1]\!] = \lambda\kappa.\kappa 1$$

$$\mathit{Sema}[\![I]\!] = \lambda\kappa\sigma.\kappa(\sigma I)\sigma$$

$$\mathit{Sema}[\![\mathrm{ae1}\ \underline{\mathrm{plus}}\ \mathrm{ae2}]\!] = \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\lambda v_1.\mathit{Sema}[\![\mathrm{ae2}]\!](\lambda v_2.\kappa(v_1+v_2)))$$

$$\mathit{Sema}[\![\mathrm{cmd}\ \underline{\mathrm{res}}\ \mathrm{ae}]\!] = \lambda\kappa.\mathit{Semc}[\![\mathrm{cmd}]\!](\mathit{Sema}[\![\mathrm{ae}]\!]\kappa)$$

$$\mathit{Sema}[\![\underline{\mathrm{let}}\ I\ \underline{\mathrm{be}}\ \mathrm{ae1}\ \underline{\mathrm{in}}\ \mathrm{ae2}]\!] =$$

$$\lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\lambda v_1\sigma_1.\mathit{Sema}[\![\mathrm{ae2}]\!](\lambda v_2\sigma_2.\kappa v_2\sigma_2[\sigma_1 I/I])(\sigma_1[v_1/I]))$$

Table 2.3 Continuation Semantics

$Sema[\![\underline{\text{if}}\ \text{be}\ \underline{\text{then}}\ \text{ae1}\ \underline{\text{else}}\ \text{ae2}]\!] =$

$\quad \lambda\kappa.Semb[\![\text{be}]\!](\lambda b.b \rightarrow Sema[\![\text{ae1}]\!]\kappa, Sema[\![\text{ae2}]\!]\kappa)$

$Semb[\![\text{ae1} = \text{ae2}]\!] = \lambda\kappa.Sema[\![\text{ae1}]\!](\lambda v_1.Sema[\![\text{ae2}]\!](\lambda v_2.\kappa(v_1 = v_2)))$

$Semb[\![\text{be1}\ \underline{\text{and}}\ \text{be2}]\!] = \lambda\kappa.Semb[\![\text{be1}]\!](\lambda b.b \rightarrow Semb[\![\text{be2}]\!]\kappa, \kappa(ff))$

Table 2.3 Continuation Semantics Continued

3. Derivation of the Compiler

We next follow the strategy laid out in [28]. We first introduce special purpose combinators to eliminate bound variables from the equations in Table 2.3. This gives a simple representation of program phrases as trees. We then use distributive and associative laws to reduce the trees to standard forms. Last, we derive a machine which simulates the application of the tree representations as functions.

Table 3.1 gives a list of the special-purpose combinators we will use. We have adopted the convention of putting "compile-time" information on the left of the equality. Table 3.2 shows the semantic equations rewritten using the new combinators.

Lemma 3.1 (i) $\mathcal{B}_{k+1}(\alpha,\beta)x = \mathcal{B}_k(\alpha,\beta x)$

(ii) $\mathcal{B}_1(\alpha,\lambda x.x) = \alpha.$ □

Theorem 3.1 The equations of Table 3.2 agree with those of Table 2.3.

Proof: No induction is necessary; we merely expand the combinators in Table 3.2 and use $\alpha\beta\eta$-conversions to reclaim the equations of Table 2.3. We do five illustrative cases.

1. Assignment Command.

1. Assignment Command

$\mathcal{B}_1(\mathit{Sema}[\![\mathrm{ae}]\!],\mathit{store}\,\mathrm{I})$

$= \lambda\eta.\mathit{Sema}[\![\mathrm{ae}]\!](\mathit{store}\,\mathrm{I}\eta)$ (Def. of $\mathcal{B}_1$)

$= \lambda\eta.\mathit{Sema}[\![\mathrm{ae}]\!](\lambda v\sigma.\eta\sigma[v/\mathrm{I}])$ (Def. of *store*)

2. Conditional Command

$\mathcal{B}_1(\mathit{Semb}[\![\mathrm{be}]\!],\mathit{test}_1(\mathit{Semc}[\![\mathrm{cmd1}]\!],\mathit{Semc}[\![\mathrm{cmd2}]\!]))$

$= \lambda\eta.\mathit{Semb}[\![\mathrm{be}]\!](\mathit{test}_1(\mathit{Semc}[\![\mathrm{cmd1}]\!],\mathit{Semc}[\![\mathrm{cmd2}]\!])\eta)$ (Def. of $\mathcal{B}_1$)

$= \lambda\eta.\mathit{Semb}[\![\mathrm{be}]\!](\lambda b.b\rightarrow\mathit{Semc}[\![\mathrm{cmd1}]\!]\eta,\mathit{Semc}[\![\mathrm{cmd2}]\!]\eta)$ (Def. of test_1)

3. While Command

$\mathit{loop}(\mathit{Semb}[\![\mathrm{be}]\!],\mathit{Semc}[\![\mathrm{cmd}]\!])$

$= \lambda\eta.\mathrm{Y}(\lambda\theta.\mathit{Semb}[\![\mathrm{be}]\!](\lambda b.b\rightarrow\mathit{Semc}[\![\mathrm{cmd}]\!]\theta,\eta))$

4. Addition Expressions

$\mathcal{B}_1(\mathit{Sema}[\![\mathrm{ae1}]\!],\mathcal{B}_2(\mathit{Sema}[\![\mathrm{ae2}]\!],\mathit{add}))$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\mathcal{B}_2(\mathit{Sema}[\![\mathrm{ae2}]\!],\mathit{add})\kappa)$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\mathcal{B}_1(\mathit{Sema}[\![\mathrm{ae}]\!],\mathit{add}\kappa))$ (Lemma 3.1 (i))

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\lambda v_1.\mathit{Sema}[\![\mathrm{ae2}]\!](\mathit{add}\kappa v_1))$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\lambda v_1.\mathit{Sema}[\![\mathrm{ae2}]\!](\lambda v_2.\kappa(v_1+v_2)))$

5. Let-expressions

$\mathcal{B}_1(\mathit{Sema}[\![\mathrm{ae1}]\!],\mathcal{B}_1(\mathit{save}\,\mathrm{I},\mathcal{B}_2(\mathit{Sema}[\![\mathrm{ae2}]\!],\mathit{unsave}\,\mathrm{I})))$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\mathcal{B}_1(\mathit{save}\,\mathrm{I},\mathcal{B}_2(\mathit{Sema}[\![\mathrm{ae2}]\!],\mathit{unsave}\,\mathrm{I}))\kappa)$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\mathit{save}\,\mathrm{I}(\mathcal{B}_2(\mathit{Sema}[\![\mathrm{ae2}]\!],\mathit{unsave}\,\mathrm{I})\kappa))$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\mathit{save}\,\mathrm{I}(\mathcal{B}_1(\mathit{Sema}[\![\mathrm{ae2}]\!],\mathit{unsave}\,\mathrm{I}\kappa)))$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\lambda v_1\sigma_1.\mathcal{B}_1(\mathit{Sema}[\![\mathrm{ae2}]\!],\mathit{unsave}\,\mathrm{I}\kappa)(\sigma_1\mathrm{I})(\sigma_1[v_1/\mathrm{I}]))$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\lambda v_1\sigma_1.\mathit{Sema}[\![\mathrm{ae2}]\!](\mathit{unsave}\,\mathrm{I}\kappa(\sigma_1\mathrm{I}))(\sigma_1[v_1/\mathrm{I}]))$

$= \lambda\kappa.\mathit{Sema}[\![\mathrm{ae1}]\!](\lambda v_1\sigma_1.\mathit{Sema}[\![\mathrm{ae2}]\!](\lambda v_2\sigma_2.\kappa v_2\sigma_2[\sigma_1\mathrm{I}/\mathrm{I}])(\sigma_1[v_1/\mathrm{I}]))$. □

The formulation of Table 3.2 suggests a simple tree-structured representation, e.g.

$\mathit{Repc}[\![\mathrm{I:=ae}]\!] = [\underline{B}_1\ \mathit{Repa}[\![\mathrm{ae}]\!]\ [\underline{\mathrm{store}}\ \mathrm{I}]]$

$\mathit{Repa}[\![\mathrm{ae1}\ \underline{\mathrm{plus}}\ \mathrm{ae2}]\!] = [\underline{B}_1\ \mathit{Repa}[\![\mathrm{ae1}]\!]\ [\underline{B}_2\ \mathit{Repa}[\![\mathrm{ae2}]\!]\ \underline{\mathrm{add}}]]$

we can do better, however, by using the following properties of the combinators to linearize the trees as much as possible:

Theorem 3.2

(i) $\mathcal{B}_k(\mathcal{B}_p(\alpha,\beta),\gamma) = \mathcal{B}_{k+p-1}(\alpha,\mathcal{B}_k(\beta,\gamma))$ if $p\geq 1$

(ii) $\mathcal{B}_k(\mathit{continue},\beta) = \beta$

(iii) $\mathcal{B}_k(\mathit{test}_1(\alpha,\beta),\gamma) = \mathit{test}_k(\mathcal{B}_k(\alpha,\gamma),\mathcal{B}_k(\beta,\gamma))$

Proof:

(i) $\mathcal{B}_k(\mathcal{B}_p(\alpha,\beta),\gamma)x_1\dots x_k x_{k+1}\dots x_{k+p-1}$

$= \mathcal{B}_p(\alpha,\beta)(\gamma x_1\dots x_k)x_{k+1}\dots x_{k+p-1}$

$= \alpha(\beta(\gamma x_1\dots x_k)x_{k+1}\dots x_{k+p-1})$

$= \alpha(\mathcal{B}_k(\beta,\gamma)x_1\dots x_k x_{k+1}\dots x_{k+p-1})$

$= \mathcal{B}_{k+p-1}(\alpha,\mathcal{B}_k(\beta,\gamma))x_1\dots x_{k+p-1}.$

(ii) $\mathcal{B}_k(\mathit{continue},\beta)x_1\dots x_k = \mathit{continue}(\beta x_1\dots x_k) = \beta x_1\dots x_k.$

(iii) $\mathcal{B}_k(\mathit{test}_1(\alpha,\beta),\gamma)x_1\dots x_k$

$= \mathit{test}_1(\alpha,\beta)(\gamma x_1\dots x_k)$

$= \lambda b.b\rightarrow\alpha(\gamma x_1\dots x_k),\beta(\gamma x_1\dots x_k)$

$= \lambda b.b\rightarrow\mathcal{B}_k(\alpha,\gamma)x_1\dots x_k,\mathcal{B}_k(\beta,\gamma)x_1\dots x_k$

$= \mathit{test}_k(\mathcal{B}_k(\alpha,\gamma),\mathcal{B}_k(\beta,\gamma))$ $\square$

We can now write a simplification function *simpl*, as shown in Table 3.3. Let $\phi(t)$ be the value obtained by interpreting the symbols of t as the corresponding functions. (This is possible because the trees form an initial algebra [4]).

Theorem 3.3 For any t, $\mathit{simpl}(t)$ halts, and $\phi(\mathit{simpl}(t)) = \phi(t)$.

Proof: To see that *simpl* halts, observe that each recursive call on *simpl* either decreases the number of nodes in the tree or the number of B's which are left sons of B's. The second part is immediate from Theorem 3.2. □

We now write down the compiler in Table 3.4. Like most simple compilers, it runs by structural induction.

Theorem 3.4

(i) $\phi(\mathit{Compp}[\![\text{pgm}]\!]) = \mathit{Semp}[\![\text{pgm}]\!]$

(ii) $\phi(\mathit{Compc}[\![\text{cmd}]\!]) = \mathit{Semc}[\![\text{cmd}]\!]$

(iii) $\phi(\mathit{Compa}[\![\text{ae}]\!]) = \mathit{Sema}[\![\text{ae}]\!]$

(iv) $\phi(\mathit{Compb}[\![\text{be}]\!]) = \mathit{Semb}[\![\text{be}]\!]$

Proof: By structural induction. Comparing Table 3.4 with Table 3.2, most of the cases are immediate, being simple transcriptions. By Theorem 3.3, *simpl* preserves ϕ, and by Lemma 3.1 (ii), padding with continue preserves ϕ. This takes care of the two remaining cases. □

Figures 3.1-3.2 show the code produced for a simple program. The purpose of the continue's in the code for the while-command is to simplify the format of the output of *simpl*.

Let instructions and sequences be defined by the grammar of Table 3.5. Then we have:

Theorem 3.5 $\mathit{Compp}[\![\text{pgm}]\!]$ is a sequence.

Proof: We start by assuming that a sequence could be any tree, and show that no arrangements other than those in table 3.5 actually occur. First, note that *simpl* preserves the rightmost leaf of its argument. Since *simpl* is called with either halt or continue as its rightmost leaf, the result with have either halt or continue as its rightmost leaf. Furthermore, any right son node other than the original rightmost node will eventually be converted into a left son by the first rule. Finally, any B or test which appears as a left son is eliminated. □

$B_k(\alpha,\beta) = \lambda x_1 \ldots x_k . \alpha(\beta x_1 \ldots x_k) \qquad (k \geq 0)$

$\mathit{halt} = \lambda\sigma.\sigma \qquad (\varepsilon\ C)$

$\mathit{continue}_1 = \lambda\eta.\eta \qquad (\varepsilon\ [C \to C])$

$\mathit{continue}_2 = \lambda\kappa.\kappa \qquad (\varepsilon\ [BC \to BC])$

$\mathit{store}\mathrm{I} = \lambda\eta v\sigma.\eta\sigma[v/\mathrm{I}]$

$\mathit{test}_k\alpha\beta = \lambda x_1 \ldots x_k b.b \to (\alpha x_1 \ldots x_k),(\beta x_1 \ldots x_k) \qquad (k \geq 0)$

$\mathit{loop}\tau\alpha = \lambda\eta.Y(\lambda\theta.\tau(\lambda b.b \to \alpha\theta,\eta))$

$\mathit{loadi}v = \lambda\kappa.\kappa v$

$\mathit{load}\mathrm{I} = \lambda\kappa\sigma.\kappa(\sigma\mathrm{I})\sigma$

$\mathit{add} = \lambda\kappa v_1 v_2.\kappa(v_1 + v_2)$

$\mathit{save}\mathrm{I} = \lambda\kappa v\sigma.\kappa(\sigma\mathrm{I})\sigma[v/\mathrm{I}]$

$\mathit{unsave}\mathrm{I} = \lambda\kappa v_1 v_2 \sigma_2.\kappa v_2 \sigma_2[v_1/\mathrm{I}]$

$\mathit{equal} = \lambda\kappa v_1 v_2.\kappa(v_1 = v_2)$

Table 3.1 Auxiliary Combinators

$$Semp[\![\mathtt{cmd}]\!] = B_0(Semc[\![\mathtt{cmd}]\!], halt)$$

$$Semc[\![\underline{\mathtt{continue}}]\!] = continue_1$$

$$Semc[\![\mathtt{I{:=}ae}]\!] = B_1(Sema[\![\mathtt{ae}]\!], store\mathtt{I})$$

$$Semc[\![\underline{\mathtt{if}}\ \mathtt{be}\ \underline{\mathtt{then}}\ \mathtt{cmd1}\ \underline{\mathtt{else}}\ \mathtt{cmd2}]\!] = B_1(Semb[\![\mathtt{be}]\!], test_1(Semc[\![\mathtt{cmd1}]\!], Semc[\![\mathtt{cmd2}]\!]))$$

$$Semc[\![\mathtt{cmd1;cmd2}]\!] = B_1(Semc[\![\mathtt{cmd1}]\!], Semc[\![\mathtt{cmd2}]\!])$$

$$Semc[\![\underline{\mathtt{while}}\ \mathtt{be}\ \underline{\mathtt{do}}\ \mathtt{cmd}]\!] = loop(Semb[\![\mathtt{be}]\!], Semc[\![\mathtt{cmd}]\!])$$

$$Sema[\![\mathtt{0}]\!] = loadi\mathtt{0}$$

$$Sema[\![\mathtt{1}]\!] = loadi\mathtt{1}$$

$$Sema[\![\mathtt{I}]\!] = load\mathtt{I}$$

$$Sema[\![\mathtt{ae1}\ \underline{\mathtt{plus}}\ \mathtt{ae2}]\!] = B_1(Sema[\![\mathtt{ae1}]\!], B_2(Sema[\![\mathtt{ae2}]\!], add))$$

$$Sema[\![\mathtt{cmd}\ \underline{\mathtt{res}}\ \mathtt{ae}]\!] = B_1(Semc[\![\mathtt{cmd}]\!], Sema[\![\mathtt{ae}]\!])$$

$$Sema[\![\underline{\mathtt{let}}\ \mathtt{I}\ \underline{\mathtt{be}}\ \mathtt{ae1}\ \underline{\mathtt{in}}\ \mathtt{ae2}]\!] = B_1(Sema[\![\mathtt{ae1}]\!], B_1(save\mathtt{I}, B_2(Sema[\![\mathtt{ae2}]\!], unsave\mathtt{I})))$$

$$Sema[\![\underline{\mathtt{if}}\ \mathtt{be}\ \underline{\mathtt{then}}\ \mathtt{ae1}\ \underline{\mathtt{else}}\ \mathtt{ae2}]\!] = B_1(Semb[\![\mathtt{be}]\!], test_1(Sema[\![\mathtt{ae1}]\!], Sema[\![\mathtt{ae2}]\!]))$$

$$Semb[\![\mathtt{ae1 = ae2}]\!] = B_1(Sema[\![\mathtt{ae1}]\!], B_2(Sema[\![\mathtt{ae2}]\!], equal))$$

$$Semb[\![\mathtt{be1}\ \underline{\mathtt{and}}\ \mathtt{be2}]\!] = B_1(Semb[\![\mathtt{be1}]\!], test_1(Semb[\![\mathtt{be2}]\!], loadiff))$$

Table 3.2 Equations Rewritten Using Auxiliary Combinators

$\mathit{simpl}[\underline{B}_k\ [\underline{B}_p\ \alpha\ \beta]\ \gamma] = \mathit{Simpl}[\underline{B}_{k+p-1}\ \alpha\ [\underline{B}_k\ \beta\ \gamma]]$

$\mathit{simpl}[\underline{B}_k\ \underline{\mathrm{halt}}\ \gamma] = \mathit{simpl}\gamma$

$\mathit{simpl}[\underline{B}_k\ \underline{\mathrm{continue}}\ \gamma] = \mathit{simpl}\gamma$

$\mathit{simpl}[B_k\ [\underline{\mathrm{test}}_1\ \alpha\ \beta]\ \gamma] = [\underline{\mathrm{test}}_k\ \mathit{simpl}[\underline{B}_k\ \alpha\ \gamma]\ \mathit{simpl}[\underline{B}_k\ \beta\ \gamma]]$

$\mathit{simpl}[\underline{B}_k\ \alpha\ \beta] = [\underline{B}_k\ \mathit{simpl}\alpha\ \mathit{simpl}\beta]$ if none of above cases apply

$\mathit{simpl}t = t$ otherwise

Table 3.3 Simplification of Tree Representations

$$\mathit{Compp}\llbracket\text{cmd}\rrbracket = \mathit{simpl}[\underline{\text{B}}_0\ \mathit{Compc}\llbracket\text{cmd}\rrbracket\ [\underline{\text{halt}}]]$$

$$\mathit{Compc}\llbracket\text{continue}\rrbracket = [\underline{\text{continue}}_1]$$

$$\mathit{Compc}\llbracket\text{I := ae}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compa}\llbracket\text{ae}\rrbracket\ [\underline{\text{store}}\ \text{I}]]]$$

$$\mathit{Compc}\llbracket\underline{\text{if}}\ \text{be}\ \underline{\text{then}}\ \text{cmd1}\ \underline{\text{else}}\ \text{cmd2}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compb}\llbracket\text{be}\rrbracket\ [\underline{\text{test}}_1\ \mathit{Compc}\llbracket\text{cmd1}\rrbracket\ \mathit{Compc}\llbracket\text{cmd2}\rrbracket\ ]]$$

$$\mathit{Compc}\llbracket\text{cmd1; cmd2}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compc}\llbracket\text{cmd1}\rrbracket\ \mathit{Compc}\llbracket\text{cmd2}\rrbracket\ ]$$

$$\mathit{Compc}\llbracket\underline{\text{while}}\ \text{be}\ \underline{\text{do}}\ \text{cmd}\rrbracket\ [\underline{\text{loop}}\ \mathit{simpl}[\underline{\text{B}}_1\ \mathit{Compb}\llbracket\text{be}\rrbracket\ \underline{\text{continue}}_2]\ \mathit{simpl}[\beta_1\ \mathit{Compc}\llbracket\text{cmd}\rrbracket\ \underline{\text{continue}}_1]]$$

$$\mathit{Compa}\llbracket 0\rrbracket = [\underline{\text{loadi}}\ 0]$$

$$\mathit{Compa}\llbracket 1\rrbracket = [\underline{\text{loadi}}\ 1]$$

$$\mathit{Compa}\llbracket\text{I}\rrbracket = [\text{load I}]$$

$$\mathit{Compa}\llbracket\text{ae1}\ \underline{\text{plus}}\ \text{ae2}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compa}\llbracket\text{ae1}\rrbracket\ [\underline{\text{B}}_2\ \mathit{Compa}\llbracket\text{ae2}\rrbracket\ [\underline{\text{add}}]]]$$

$$\mathit{Compa}\llbracket\text{cmd}\ \underline{\text{res}}\ \text{ae}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compc}\llbracket\text{cmd}\rrbracket\ \mathit{Compa}\llbracket\text{ae}\rrbracket]$$

$$\mathit{Compa}\llbracket\underline{\text{let}}\ \text{I}\ \underline{\text{be}}\ \text{ae1}\ \underline{\text{in}}\ \text{ae2}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compa}\llbracket\text{ae1}\rrbracket\ [\underline{\text{B}}_1\ [\underline{\text{save}}\ \text{I}]\ [\underline{\text{B}}_2\ \mathit{Compa}\llbracket\text{ae2}\rrbracket\ [\text{unsave I}]]]]]$$

$$\mathit{Compa}\llbracket\underline{\text{if}}\ \text{be}\ \underline{\text{then}}\ \text{ae1}\ \underline{\text{else}}\ \text{ae2}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compb}\llbracket\text{be}\rrbracket\ [\underline{\text{test}}_1\ \mathit{Compa}\llbracket\text{ae1}\rrbracket\ \mathit{Compa}\llbracket\text{ae2}\rrbracket\ ]]$$

$$\mathit{Compb}\llbracket\text{ae1 = ae2}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compa}\llbracket\text{ae1}\rrbracket\ [\underline{\text{B}}_2\ \mathit{Compa}\llbracket\text{ae2}\rrbracket\ [\underline{\text{equal}}]]]$$

$$\mathit{Compb}\llbracket\text{be1}\ \underline{\text{and}}\ \text{be2}\rrbracket = [\underline{\text{B}}_1\ \mathit{Compb}\llbracket\text{be1}\rrbracket\ [\underline{\text{test}}_1\ \mathit{Compb}\llbracket\text{be2}\rrbracket\ [\underline{\text{loadi}}\ \text{ff}]]]$$

Table 3.4 The Compiler

ans := 0;
<u>while</u> $y=0$ <u>do</u> ans := ans + x

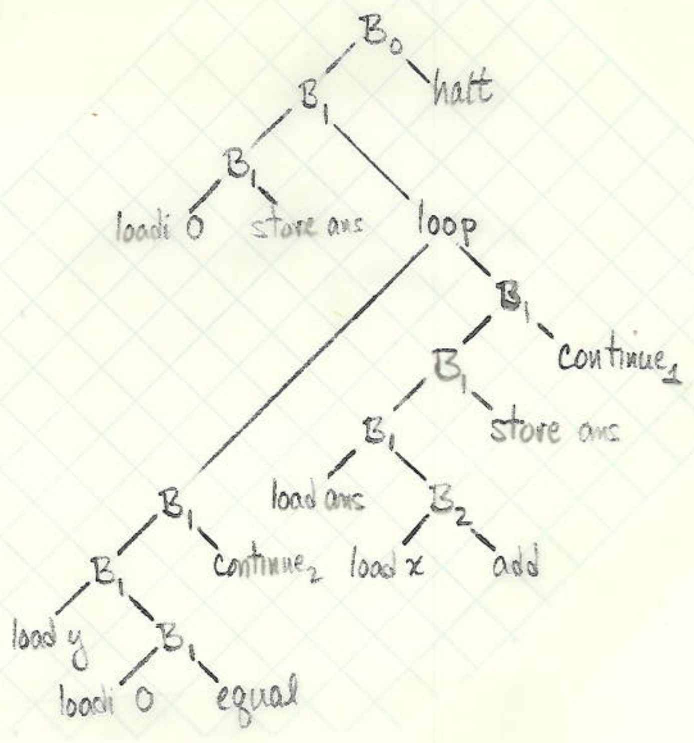
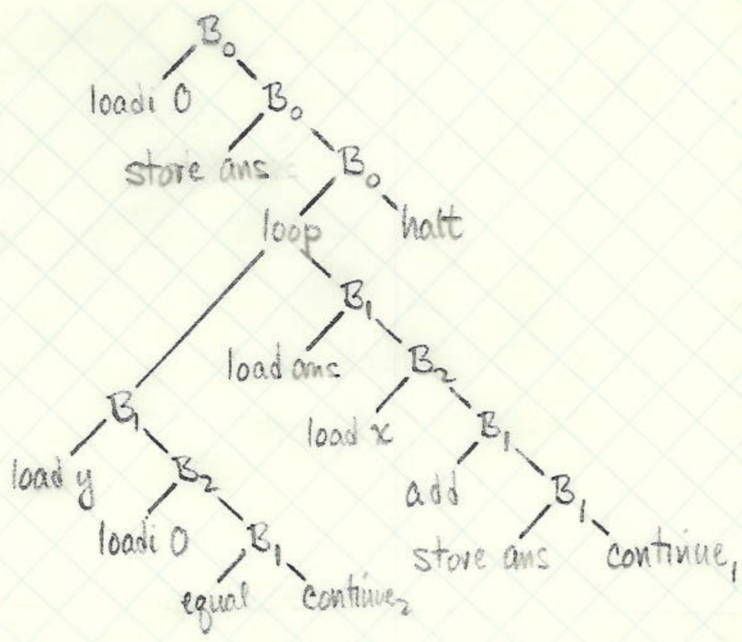


Figure 3.1



B_0
loadi 0
B_0
store ans
B_0
loop
halt
B_1
load ans
B_2
B_1
load x
B_1
load y
B_2
add
B_1
loadi 0
B_1
store ans
continue$_1$
equal
continue$_2$

Figure 3.2

```
<ins> ::= [store I]
          [loop <seq> <seq>]
          [loadi v]
          [load I]
          [add]
          [save I]
          [unsave I]
          [equal]
<seq> ::= [halt]
          [continue]
          [B_k <ins> <seq>]
          [test_k <seq> <seq>]
```

Table 3.5 Grammar for Simplified Representations

4. Derivation of the Machine

We next develop a machine to interpret sequences. If α is a sequence, $\phi(\alpha)$ may be of type $W^n \rightarrow C$ where

$$W = N + B + C + BC \quad \text{(witnessed values)}$$

We wish to define a function M_n: $\text{Seq} \rightarrow (\text{Rep}_W)^n \rightarrow S \rightarrow S$ such that

$$M_n \alpha x_1 \ldots x_n \sigma = (\phi\alpha)(\phi_W x_1)\ldots(\phi_W x_n)\sigma$$

where ϕ_W denotes the decoding function for Rep_W. The last two summands in W are required because unlike [10], we have only trees in our representations, not graphs. Hence a while-loop must stack "return addresses" for its test (a boolean continuation) and for its body (a command continuation). To facilitate this, we introduce two more combinators:

$$\mathit{loopbody}_k \tau\alpha\beta x_1 \ldots x_k = \lambda b.b \rightarrow \alpha(\mathit{loop}\tau\alpha(\beta x_1 \ldots x_k)), \beta x_1 \ldots x_k$$

$$\mathit{looptop}_k \tau\alpha\beta x_1 \ldots x_k = \mathit{loop}\tau\alpha(\beta x_1 \ldots x_k)$$

A representation of W will be either a number, a boolean, or built from these combinators. ϕ_W is the evident decoding function.

The machine is given in Table 4.1. Conceptually, M_n has a "program counter" α, a stack $x_1, \ldots, x_n$ (with the top at the right-hand end), and a state σ. The subscript n indicates the number of entries in the stack. Typically, the contents of the program counter is a sequence $[\underline{B}_p \text{ ins } \beta]$. The action of the machine may be derived by considering:

$$M_n[\underline{B}_p \text{ ins } \beta]x_1 \ldots x_p x_{p+1} \ldots x_n \sigma$$

$$= \mathcal{B}_p(\text{ins}, \beta)x_1 \ldots x_p x_{p+1} \ldots x_n \sigma$$

$$= \text{ins}(\beta x_1 \ldots x_p)x_{p+1} \ldots x_n \sigma$$

where we have suppressed the ϕ's. The instruction sees the continuation, the top n-p entries on the stack, and the store. This agrees nicely with the functionalities of auxiliary combinators, now reborn as opcodes, in Table 3.1. The parameter p gives the number of stack entries which are protected from the current instruction. In most cases, the invariant that $[\underline{B}_p \text{ ins } \beta]x_1 \ldots x_n$ represents a value of type C provides an adequate constraint on n and p. Note that the machine has something quite close to a conventional fetch-execute cycle: the instruction ins is fetched, and the program counter is advanced to β. A number of such machines are constructed in [28], which also shows how to derive a machine from the definition of the combinators, and how to eliminate the subscripts on the B's. (All this owes much to [15]). Forsaking the constructive approach momentarily, we state:

<u>Theorem 4.1</u> If $\phi(\text{seq})(\phi_W x_1)\ldots(\phi_W x_n) \in C$ and $M_n(\text{seq})x_1 \ldots x_n\sigma$ halts, then $M_n(\text{seq})x_1\ldots x_n\sigma = \phi(\text{seq})(\phi_W x_1)\ldots(\phi_W x_n)\sigma$.

<u>Proof</u>: We interpret Table 4.1 as a set of recursion equations defining the functions M_k, with the proviso that if no left-hand side is applicable, the resulting computation fails to halt. We may then use subgoal induction [11]. Alternatively, since the definition is tail-recursive, we may regard it as a rewriting system and use induction on the length of the computation. In this example the two are equivalent. We do selected cases:

1. $M_0[\underline{\text{halt}}]\sigma = \sigma = \mathit{halt}\sigma$

2. $M_k[\underline{B}_k\ [\underline{\text{store}}\ I]\ \beta]x_1\ldots x_k\sigma$

   $= M_{k-1}\beta x_1\ldots x_{k-1}(\sigma[x_k/I])$ ($x_k \in N$ to get type compatibility)

   $= (\phi\beta)(\phi_W x_1)\ldots(\phi_W x_{k-1})(\sigma[\phi_w x_k/I])$ (Induc. Hyp)

   $= \mathit{store}I((\phi\beta)(\phi_W x_1)\ldots(\phi_W x_k))\sigma$

   $= \mathcal{B}_k(\mathit{store}I,\phi\beta)(\phi_W x_1)\ldots(\phi_W x_k)\sigma$

   $= \phi([\underline{B}_k\ [\underline{\text{store}}\ I]\ \beta])(\phi_W x_1)\ldots(\phi_W x_k)\sigma.$

From this example, we learn that the ϕ's only get in the way most of the time; we shall omit them whenever possible from now on.

3. $M_k[\underline{B}_k\ [\underline{\text{load}}\ I]\ \beta]x_1\ldots x_k\sigma$

   $= M_{k+1}\beta x_1\ldots x_k(\sigma I)\sigma$ (Def. of M)

   $= (\beta x_1\ldots x_k)(\sigma I)\sigma$ (IH)

   $= \mathit{load}I(\beta x_1\ldots x_k)\sigma$

   $= \mathcal{B}_k(\mathit{load}I,\beta)x_1\ldots x_k\sigma$

Note that the subscripts on M and $\underline{B}$ must agree for the types to be compatible.

4. $M_{k+2}[\underline{B}_k\ [\underline{\text{add}}]\ \beta]x_1\ldots x_{k+2}\sigma$

   $= M_{k+1}\beta x_1\ldots x_k(x_{k+1}+x_{k+2})\sigma$ (Def. of M)

   $= (\beta x_1\ldots x_k)(x_{k+1}+x_{k+2})\sigma$ (IH)

   $= \mathit{add}(\beta x_1\ldots x_k)x_{k+1}x_{k+2}\sigma$

   $= \mathcal{B}_k(\mathit{add},\beta)x_1\ldots x_k x_{k+1}x_{k+2}\sigma$

5. $M_k[\underline{B}_k\ [\underline{\text{loop}}\ \tau\ \alpha]\ \beta]x_1\ldots x_k\sigma$

   $= M_1\tau[\underline{\text{loopbody}}_k\ \tau\ \alpha\ \beta\ x_1\ldots x_k]\sigma$

   $= \tau(\lambda b.b \rightarrow \alpha(\mathit{loop}\tau\alpha(\beta x_1\ldots x_k)),\beta x_1\ldots x_k)\sigma$

   $= \mathit{loop}\tau\alpha(\beta x_1\ldots x_k)\sigma$

   $= \mathcal{B}_k(\mathit{loop}\tau\alpha,\beta)x_1\ldots x_k\sigma.$

6. To show the correctness of the two continue instructions, note that the only values of the correct type which can appear on the stack must be built by loopbody and looptop. Hence

$$\begin{aligned}
&M_2[\underline{\text{continue}}_2][\underline{\text{loopbody}}_k\ \tau\ \alpha\ \beta\ x_1 \ldots x_k]b\sigma = \\
&\quad = b \rightarrow M_1\alpha[\underline{\text{looptop}}_k\ \tau\ \alpha\ \beta\ x_1 \ldots x_k]\sigma, M_k\beta x_1 \ldots x_k\sigma \\
&\quad = b \rightarrow \alpha(\mathit{looptop}_k\tau\alpha\beta x_1 \ldots x_k)\sigma, \beta x_1 \ldots x_k\sigma \qquad \text{(IH)} \\
&\quad = b \rightarrow \alpha(\mathit{loop}\tau\alpha(\beta x_1 \ldots x_k))\sigma, \beta x_1 \ldots x_k\sigma \\
&\quad = (b \rightarrow \alpha(\mathit{loop}\tau\alpha(\beta x_1 \ldots x_k)), \beta x_1 \ldots x_k)\sigma \\
&\quad = (\mathit{loopbody}\tau\alpha\beta x_1 \ldots x_k)b\sigma \\
&\quad = \mathit{continue}_2(\mathit{loopbody}_k\tau\alpha\beta x_1 \ldots x_k)b\sigma. \\
&M_1[\underline{\text{continue}}_1][\underline{\text{looptop}}_k\ \tau\ \alpha\ \beta\ x_1 \ldots x_k]\sigma \\
&\quad = M_k[\underline{B}_k\ [\underline{\text{loop}}\ \tau\ \alpha]\ \beta]x_1 \ldots x_k\sigma \\
&\quad = B_k(\mathit{loop}\tau\alpha, \beta)x_1 \ldots x_k\sigma \\
&\quad = \mathit{loop}\tau\alpha(\beta x_1 \ldots x_k)\sigma \\
&\quad = \mathit{continue}_1(\mathit{loop}\tau\alpha(\beta x_1 \ldots x_k))\sigma \\
&\quad = \mathit{continue}_1(\mathit{looptop}_k\tau\alpha\beta x_1 \ldots x_k)\sigma. \quad \square
\end{aligned}$$

Theorem 3.6 may be regarded as showing that M is sound (or partially correct) with respect to ϕ. We must now show that M is complete with respect to ϕ. It is at this point that congruence relations seem to be introduced [19, pp. 340-347]. Continuing our metaphor of machine code as syntactically vinegared lambda-terms, we proceed instead by adapting Plotkin's proof of the completeness of the operational semantics for PCF [13].

Theorem 4.2 If $\phi(\text{seq})(\phi_W x_1)\ldots(\phi_W x_n) \in C$ and $\phi(\text{seq})(\phi_W x_1)\ldots(\phi_W x_n)\sigma \neq \perp$ then $M_n(\text{seq})x_1 \ldots x_n\sigma$ halts.

Proof: We define the notion of computability for terms of type W and $W^n \to S \to S$ as follows:

(i) a term β representing an element of W is computable iff

(a) β is a constant of type N or B

or (b) β is of type C and for every σ, if $(\phi_W\beta)\sigma \neq \perp$, then $M_1[\underline{\text{continue}_1}]\beta\sigma$ halts

or (c) β is of type BC and for every σ and every constant b of type B, if $(\phi_W\beta)b\sigma \neq \perp$, then $M_2[\underline{\text{continue}_2}]\beta b\sigma$ halts

(ii) a sequence α of type $W^n \to S \to S$ is computable iff for any computable $x_1,\ldots x_n \in W$, and any σ, if $(\phi\alpha)(\phi_W x_1)(\phi_W x_2)\ldots(\phi_W x_n) \in C$ and $(\phi\alpha)(\phi_W x_1)\ldots(\phi_W x_n)\sigma \neq \perp$, then $M_n\alpha x_1\ldots x_n\sigma$ halts. (end of definition of computability.)

We need only show that every sequence is computable, and every stacked term is computable. We proceed by structural induction on sequences. All the cases except those pertaining to loops are easy; we do store for an example:

Assume $\beta, x_1,\ldots,x_k$ are computable, $\phi([\underline{B}_n\ [\underline{\text{store}}\ I]\ \beta)(\phi_W x_1)\ldots(\phi_W x_n) \in C$ and $\phi([\underline{B}_k\ [\underline{\text{store}}\ I]\ \beta])(\phi_W x_1)\ldots(\phi_W x_n)\sigma \neq \perp$. Note that x_n must be of type N, so $\phi_W(x_n) = x_n$. Then

$$\perp \neq \phi([\underline{B}_k\ [\underline{\text{store}}\ I]\ \beta)(\phi_W x_1)\ldots(\phi_W x_n)\sigma = \mathit{store}I((\phi\beta)(\phi_W x_1)\ldots(\phi_W x_n))\sigma$$
$$= (\phi\beta)(\phi_W x_1))\ldots(\phi_W x_{n-1})(\sigma[\phi_W x_n/I])$$

Since $\beta, x_1,\ldots x_n$ are computable, $M_{n-1}\beta x_1\ldots x_{n-1}(\sigma[x_n/I])$ halts. Since this equals $M_n[\underline{B}_n\ [\underline{\text{store}}\ I]\ \beta]x_1\ldots x_n\sigma$, the latter halts also. Hence $[\underline{B}_n\ [\underline{\text{store}}\ I]\ \beta]$ is computable.

The only hard case is showing that <u>loop</u>, <u>looptop</u>, and <u>loopbody</u> preserve computability. To do this, introduce the "bounded" loop combinators:

$$y^{(0)} = \lambda f.\bot$$
$$y^{(p+1)} = \lambda f.f(y^{(p)}f)$$
$$\mathit{loop}^{(p)}\tau\alpha = \lambda\eta.y^{(p)}(\lambda\theta.\tau(\lambda b.b \to \alpha\theta,\eta))$$
$$\mathit{loopbody}_k^{(p)}\tau\alpha\beta x_1\ldots x_k = \lambda b.b \to \alpha(\mathit{loop}^{(p)}\tau\alpha(\beta x_1\ldots x_k)),\beta x_1\ldots x_k$$
$$\mathit{looptop}_k^{(p)}\tau\alpha\beta x_1\ldots x_k = \mathit{loop}^{(p)}\tau\alpha(\beta x_1\ldots x_k)$$

and add to the machine the rules:

$$M_k[\underline{B}_k\ [\underline{\text{loop}}^{(p+1)}\ \tau\ \alpha]\ \beta]x_1\ldots x_k\sigma = M_1\tau[\underline{\text{loopbody}}_k^{(p)}\ \tau\ \alpha\ \beta\ x_1\ldots x_k]\sigma \quad (p \geq 0)$$

$$M_2[\underline{\text{continue}}_2][\underline{\text{loopbody}}_k^{(p)}\ \tau\ \alpha\ \beta\ x_1\ldots x_k]b\sigma = b \to M_1\alpha[\underline{\text{looptop}}_k^{(p)}\ \tau\ \alpha\ \beta\ x_1\ldots x_k]\sigma,\ M_k\beta x_1\ldots x_k\sigma \quad (p \geq 0)$$

$$M_1[\underline{\text{continue}}_1][\underline{\text{looptop}}_k^{(p)}\ \tau\ \alpha\ \beta\ x_1\ldots x_k]\sigma = M_k[\underline{B}_k\ [\underline{\text{loop}}^{(p)}\ \tau\ \alpha]x_1\ldots x_k\sigma. \quad (p \geq 0)$$

Clearly each bounded combinator preserves computability (by induction on p).

If t and t' are terms, define $t \leq t'$ iff t can be obtained from t' by replacing some occurrences of <u>loop</u>, <u>looptop</u>, and <u>loopbody</u> by bounded versions. Now, if $M_n\alpha x_1\ldots x_n\sigma$ halts, $\alpha \leq \alpha'$, and $x_i \leq x_i'$, then $M_n\alpha' x_1'\ldots x_n'\sigma = M_n\alpha x_1\ldots x_n\sigma$, since every move of M involving bounded combinators can be mimicked by unbounded combinators performing the same manipulations on σ. (This is like Lemma 3.2 in [13]).

Now we can show that the <u>loop</u> instruction preserves computability. Let $\tau,\alpha,\beta,x_1,\ldots,x_n$ all be computable, and

$\phi([\underline{B}_k\ [\underline{\text{loop}}\ \tau\ \alpha]\ \beta])(\phi_W x_1)\ldots(\phi_W x_k)\sigma \neq \perp$. Let $\eta = (\phi\beta)(\phi_W x_1)\ldots(\phi_W x_k)$. Then $\mathit{loop}\tau\alpha\eta\sigma \neq \perp$. Since $\mathit{loop}\tau\alpha\eta\sigma = \bigcup_p \mathit{loop}^{(p)}\tau\alpha\eta\sigma$ there must be some p such that $\mathit{loop}^{(p)}\tau\alpha\eta\sigma \neq \perp$. Hence $M_k[\underline{B}_k[\underline{\text{loop}}^{(p)}\ \tau\ \alpha]\ \beta]x_1\ldots x_k\sigma$ halts. Hence $M_k[\underline{B}_k\ [\underline{\text{loop}}\ \tau\ \alpha]\ \beta]x_1\ldots x_k\sigma$ halts. The cases for <u>loopbody</u> and <u>looptop</u> are similar. □

$M_o[\underline{\text{halt}}]\sigma = \sigma$

$M_{k+1}[\underline{\text{test}}_k\ \alpha\ \beta]x_1\ldots x_k b\sigma = b \rightarrow M_k\alpha x_1\ldots x_k\sigma, M_k\beta x_1\ldots x_k\sigma$

$M_k[\underline{B}_k\ [\underline{\text{store}}\ I]\ \beta]x_1\ldots x_k\sigma = M_{k-1}\beta x_1\ldots x_{k-1}\sigma[x_k/I]$

$M_k[\underline{B}_k\ [\underline{\text{loadi}}\ v]\ \beta]x_1\ldots x_k\sigma = M_{k+1}\beta x_1\ldots x_k v\sigma$

$M_k[\underline{B}_k\ [\underline{\text{load}}\ I]\ \beta]x_1\ldots x_k\sigma = M_{k+1}\beta x_1\ldots x_k(\sigma I)\sigma$

$M_{k+2}[\underline{B}_k\ [\underline{\text{add}}]\ \beta]x_1\ldots x_k x_{k+1} x_{k+2} = M_{k+1}\beta x_1\ldots x_k(x_{k+1}+x_{k+2})\sigma$

$M_{k+1}[\underline{B}_k\ [\underline{\text{save}}\ I]\ \beta]x_1\ldots x_k x_{k+1}\sigma = M_{k+1}\beta x_1\ldots x_k(\sigma I)\sigma[x_{k+1}/I]$

$M_{k+2}[\underline{B}_k\ [\underline{\text{unsave}}\ \underline{I}]\ \beta]x_1\ldots x_k x_{k+1} x_{k+2}\sigma = M_{k+1}\beta x_1\ldots x_k x_{k+2}\sigma[x_{k+1}/I]$

$M_{k+2}[\underline{B}_k\ [\underline{\text{equal}}]\ \beta]x_1\ldots x_k x_{k+1} x_{k+2}\sigma = M_{k+1}\beta x_1\ldots x_k(x_{k+1}=x_{k+2})\sigma$

$M_k[\underline{B}_k\ [\underline{\text{loop}}\ \tau\ \alpha]\ \beta]x_1\ldots x_k\sigma = M_1\tau[\underline{\text{loopbody}}_k\ \tau\ \alpha\ \beta\ x_1\ldots x_k]\sigma$

$M_2[\underline{\text{continue}}_2][\underline{\text{loopbody}}_k\ \tau\ \alpha\ \beta\ x_1\ldots x_k]b\sigma =$
$\qquad b\rightarrow M_1\alpha[\underline{\text{looptop}}_k\ \tau\ \alpha\ \beta\ x_1\ldots x_k]\sigma,\ M_k\beta x_1\ldots x_k\sigma$

$M_1[\underline{\text{continue}}_1][\underline{\text{looptop}}_k\ \tau\ \alpha\ \beta\ x_1\ldots x_k]\sigma = M_k[\underline{B}_k\ [\underline{\text{loop}}\ \tau\ \alpha]\ \beta]x_1\ldots x_k\sigma$

Table 4.1 Machine for Interpreting Representations

5. Putting It All Together

In Section 1, we stated that the data type of the implementation was to be a representation of the data type of the semantics, that is, there was to be a homomorphism (abstraction function) from the implementation algebra to the semantic algebra. In this section we will list the sorts and operations of these algebras, and show how the theorems of Sections 3 and 4 fit together to show the existence of the homomorphism.

Tables 5.1-5.3 show the two algebras. They have 9 sorts: 4 for the four kinds of phrases (syntactic sorts), 4 for the meanings of the phrase sorts (semantic sorts), and one for states. There are 5 operations: one for the semantics of each sort and one showing the result of a program-meaning applied to a state. In both algebras, the carriers for the syntactic sorts are the sets of phrases of each syntactic category, as given in Table 2.1, and the carrier for the sort of states is just Id → N, the domain of states.

In the semantic algebra S, the carriers for the semantic sorts are the appropriate semantic domains. The functions for the four semantic operations are the four valuations defined in Table 2.3, and the function for application is just functional application.

In the implementation algebra T, the carriers for the semantic sorts are the sets of representations. For pgm, this is the set of sequences; for the others it is the set of tree representations of appropriate type. The functions for the four semantic operations are the four compiling functions defined in Table 3.5. The function for application of a program-meaning α to a state σ is the result of running the machine M_o on α and σ. We may now state the main theorem:

Theorem 5.1 The representation function ϕ extends to a homorphism from T to S.

Proof: Let the homomorphism be given by ϕ on the four semantic sorts and by the identity on the other five sorts. We must show that the homomorphism preserves the five operations of the algebras. For the semantic operations, this is just Theorem 3.4. For application, we calculate

$$\begin{aligned} ap^T(\alpha,\sigma) &= M_0\alpha\sigma \\ &= (\phi\alpha)\sigma \qquad \text{(Thms 4.1-4.2)} \\ &= ap^S(\phi\alpha)\sigma. \qquad \Box \end{aligned}$$

Corollary. $\mathit{Semc}[\![\mathrm{pgm}]\!]\sigma = M_o(\mathit{Compp}[\![\mathrm{pgm}]\!])\sigma = \mathcal{DC}[\![\mathrm{cmd}]\!]\sigma.$ $\Box$

Sorts

$$\left.\begin{array}{l}\underline{\text{pgm}}\\ \underline{\text{cmd}}\\ \underline{\text{ae}}\\ \underline{\text{be}}\end{array}\right\}\quad\text{sorts for program phrases}$$

$$\left.\begin{array}{l}\underline{\text{mpgm}}\\ \underline{\text{cmd}}\\ \underline{\text{mae}}\\ \underline{\text{mbe}}\end{array}\right\}\quad\text{sorts for meanings of phrases}$$

$$\underline{\text{st}}\quad\text{sort of states}$$

Operations

$$\begin{array}{lll}
\text{spgm}: & \underline{\text{pgm}} & \rightarrow \underline{\text{mpgm}}\\
\text{scmd}: & \underline{\text{cmd}} & \rightarrow \underline{\text{mcmd}}\\
\text{sae}\ : & \underline{\text{ae}} & \rightarrow \underline{\text{mae}}\\
\text{sbe}\ : & \underline{\text{be}} & \rightarrow \underline{\text{mbe}}\\
\text{ap}\ : & \underline{\text{mpgm}},\underline{\text{st}} & \rightarrow \underline{\text{st}}
\end{array}$$

Table 5.1 Signature of the Algebras

Sorts

$\underline{\text{pgm}}^S = \text{Cmd}$ (See Table 2.1)

$\underline{\text{cmd}}^S = \text{Cmd}$

$\underline{\text{ae}}^S = \text{Ae}$

$\underline{\text{be}}^S = \text{Be}$

$\underline{\text{mpgm}}^S = \text{C}$ (See Table 2.3)

$\underline{\text{mcmd}}^S = \text{C} \to \text{C}$

$\underline{\text{mae}}^S = \text{K} \to \text{C}$

$\underline{\text{mbe}}^S = \text{BC} \to \text{C}$

$\underline{\text{st}}^S = \text{Id} \to \text{N}$

Operations

$\text{spgm}^S = \mathit{Semp}$ (See Table 2.3)

$\text{scmd}^S = \mathit{Semc}$

$\text{sae}^S = \mathit{Sema}$

$\text{sbe}^S = \mathit{Semb}$

$\text{ap}^S = \lambda\eta\sigma.\eta\sigma$

Table 5.2 The Semantic Algebra S

Sorts

$\underline{\text{pgm}}^T$ = Cmd

$\underline{\text{cmd}}^T$ = Cmd

$\underline{\text{ae}}^T$ = Ae

$\underline{\text{be}}^T$ = Be

$\underline{\text{mpgm}}^T$ = sequences of type C

$\underline{\text{mcmd}}^T$, $\underline{\text{mae}}^T$, $\underline{\text{mbe}}^T$ = tree representations of appropriate type

$\underline{\text{st}}^T$ = Id → N

Operations

spgm^T	= *Compp*	(See Table 3.5)
scmd^T	= *Compc*	
sae^T	= *Compa*	
sbe^T	= *Compb*	
ap^T	= M_0	(See Table 4.1)

Table 5.3 The Implementation Algebra T

6. Related Work, Extensions, and Conclusions

In this section we will discuss the precessors of this work. Reynolds [15] introduced the basic techniques for converting from direct to continuation semantics. Sussman & Steele [20] used continuation-passing as a key element of their implementation of SCHEME. Wand & Friedman [24] considered algorithms for the conversion from direct to continuation form and various devices for executing representations of continuations (see, in particular, the second paragraph of Section 4 of [24]). Conversion from direct to continuation semantics has also been studied by Reynolds [16] and Sethi & Tang [17, 18]. Motivated by [2], Wand [26] studied the optimizations available by using continuation representations cleverer than the simple list structures used previously.

In [23], which extended the material in [15] on removal of higher-order functions, we introduced the idea of putting syntactic and semantic objects in the same algebra and making evaluation an operation. This proposal was in distinction to the approach of the ADJ group [4], in which evaluation appeared as a homomorphism between algebras. Hoare [6] presented the idea of an implementation as a simulation, and introduced the notion of an abstraction function. In [25], we used this idea to argue that a "data type" was the final object in the category of its implementations. This improved on [23], which made no clear distinction between models and implementations.

This distinction was studied at length in [27]. Similar ideas were expressed in [3]. Our idea of implementation is merely a change in emphasis from the definition in [5], which defined an implementation of algebra A_1 in algebra A_2 as (very roughly) an injective homomorphism from A_1 into a quotient algebra of A_2. We have merely concentrated on the map from A_2 to its quotient (the abstraction function).

Mosses [12] extended [23] to postulate that the target of a denotational semantics should be an "abstract data type", i.e. an equationally defined algebra, rather than a particular lattice. He then suggested that a target machine was an implementation in the sense of [5]. His work is, of course, a direct continuation of [21] and its predecessors, including [1, 7, 9, 10].

In [28], we applied the ideas of [26] to [12] and considered cleverer representations of continuations. The idea of code as a finite representation of a function also appears in [8]. This paper extends [28] by presenting the techniques for actually doing the proof of correctness for the implementation.

In contrast to Mosses [12], we have used "concrete" data types, i.e. specific algebras, rather than "abstract" ones. The notion of implementation seems to be more complex for "abstract" data types than for "concrete" ones, because one must deal not only with the semantic "abstraction function" but with syntactic functions ("interpretation") which run the other way. For example, a map $\Sigma_1 \rightarrow \Sigma_2$ (translation of operator symbols) yields a forgetful functor from Σ_2-algebras to Σ_1-algebras. This seems to be a source of confusion in the literature. This is discussed in a somewhat different context in [27]. It would be useful to rework the results of this paper in an abstract axiomatic setting.

Our proof is surely not so elegant as that of [21]. Nevertheless, we feel our proof gains in comprehensibility what it lacks in formality. The separation of representation from recursion-removal (i.e. passage from direct to continuation semantics) seems to be an important step towards cleaner proofs. Furthermore, our approach directly attacks the problem, acknowledged by the authors of [21], that their commutative diagram "is not, in itself, 'compiler correctness.'" Note also that, notwithstanding our general advice, we still rely heavily on the initiality of the source language to construct the abstraction function ϕ and most of the operations in the algebras. A more formal treatment should be developed.

Another area for improvement is the treatment of loops. Our machine stacks a "return address" on loop entry. It would be better to compile into a flow chart with a loop in its graph, as [10] and [21] do. We conjecture that we can do this by modifying the compiler to produce an infinite tree as the representation of a loop. We could then produce flow charts as a finite representation of this infinite tree.

Our use of the "computability" property, adapted from [22] by Plotkin [13], provides an alternative to the use of congruence relations for proving termination (which is their primary purpose in [19]). We did not need to define this property by induction on types, as did [13], because of the simplicity of our machine. This suggests that it may be possible to use this technique even in the presence of reflexively defined types. We hope to explore this possibility further.

<u>Appendix</u>. Proof of Theorem 2.1.

<u>Proof</u>: By structural induction. In fact, (i)-(iii) are enough to derive the equations for *Semc*, *Sema*, and *Semb* [2, 22]. We do a few illustrative cases.

1. Assignment Command:

$$\eta \circ \mathcal{DC}[\![I := ae]\!] = \eta \circ ((\lambda v\sigma.\sigma[v/I])^* \mathcal{DA}[\![ae]\!])$$

$$= (\eta \circ \lambda v\sigma.\sigma[v/I])^* \mathcal{DA}[\![ae]\!] \qquad \text{(Lemma 2.2)}$$

$$= (\lambda v\sigma.\eta\sigma[v/I])^* \mathcal{DA}[\![ae]\!]$$

$$= \mathit{Sema}[\![ae]\!](\lambda v\sigma.\eta\sigma[v/I]) \qquad \text{(IH)}$$

$$= \mathit{Semc}[\![I{:=}ae]\!]\eta$$

2. Conditional Command

$$\eta \circ \mathcal{DC}[\![\underline{\text{if}}\ be\ \underline{\text{then}}\ cmd1\ \underline{\text{else}}\ cmd2]\!]$$

$$= \eta \circ ((\lambda b.b \rightarrow \mathcal{DC}[\![cmd1]\!], \mathcal{DC}[\![cmd2]\!])^* \mathcal{DB}[\![be]\!])$$

$$= (\eta \circ (\lambda b.b \rightarrow \mathcal{DC}[\![cmd1]\!], \mathcal{DC}[\![cmd2]\!]))^* \mathcal{DB}[\![be]\!]$$

$$= (\lambda b.b \rightarrow \eta \circ \mathcal{DC}[\![cmd1]\!], \eta \circ \mathcal{DC}[\![cmd2]\!])^* \mathcal{DB}[\![be]\!] \qquad \text{(Lemma 2.3)}$$

$$= \mathit{Semb}[\![be]\!](\lambda b.b \rightarrow \eta \circ \mathcal{DC}[\![cmd1]\!], \eta \circ \mathcal{DC}[\![cmd2]\!]) \qquad \text{(IH)}$$

$$= \mathit{Semb}[\![be]\!](\lambda b.b \rightarrow \mathit{Semc}[\![cmd1]\!]\eta, \mathit{Semc}[\![cmd2]\!]\eta) \qquad \text{(IH)}$$

$$= \mathit{Semc}[\![\underline{\text{if}}\ be\ \underline{\text{then}}\ cmd1\ \underline{\text{else}}\ cmd2]\!]\eta$$

3. While Command

$$\eta \circ \mathcal{DC}[\![\underline{\text{while}}\ be\ \underline{\text{do}}\ cmd]\!]$$

$$= \eta \circ (Y(\lambda f.(\lambda b.b \rightarrow f \circ \mathcal{DE}[\![cmd]\!], \lambda\sigma.\sigma)^* \mathcal{DB}[\![be]\!]))$$

We want to use Lemma 2.4 with

$$\phi = \lambda f.(\lambda b.b \rightarrow f \circ \mathcal{DC}[\![cmd]\!], \lambda\sigma.\sigma)^* \mathcal{DB}[\![be]\!]$$

Then, using lemmas 2.2 and 2.3, we get

$$\eta \circ \phi f = (\lambda b.b \rightarrow \eta \circ f \circ \mathcal{DC}[\![cmd]\!], \eta)^* \mathcal{DB}[\![be]\!]$$

Therefore, we set

$$\psi = \lambda\theta.(\lambda b.b \rightarrow \theta\circ\mathcal{DC}[\![\text{cmd}]\!],\eta)*\mathcal{DB}[\![\text{be}]\!]$$

so $\psi(\eta\circ f) = \eta\circ\phi f$. So,

$\eta\circ\mathcal{DC}[\![\text{while be }\underline{\text{do}}\text{ cmd}]\!]$

$$= \eta\circ Y\phi$$

$$= Y\psi \qquad \text{(by Lemma 2.4)}$$

$$= Y(\lambda\theta.(\lambda b.b \rightarrow \theta\circ\mathcal{DC}[\![\text{cmd}]\!],\eta)*\mathcal{DB}[\![\text{be}]\!])$$

$$= Y(\lambda\theta.\mathit{Semb}[\![\text{be}]\!](\lambda b.b \rightarrow \mathit{Semc}[\![\text{cmd}]\!]\theta,\eta)) \qquad \text{(IH)}$$

$$= \mathit{Semc}[\![\underline{\text{while}}\text{ be }\underline{\text{do}}\text{ cmd}]\!]\eta$$

4. Addition expressions. For expressions, it is useful to define two auxiliary valuations

$$\mathcal{V}[\![\text{ae}]\!]\sigma = \mathcal{DA}[\![\text{ae}]\!]\sigma\downarrow 1$$

$$\mathcal{S}[\![\text{ae}]\!]\sigma = \mathcal{DA}[\![\text{ae}]\!]\sigma\downarrow 2$$

which extract the value and store components of the semantics. Expanding the $*$-compositions in $\mathcal{DA}[\![\text{ae1 }\underline{\text{plus}}\text{ ae2}]\!]\sigma$, we deduce

$$\mathcal{V}[\![\text{ae1 }\underline{\text{plus}}\text{ ae2}]\!]\sigma = \mathcal{V}[\![\text{ae1}]\!]\sigma + \mathcal{V}[\![\text{ae2}]\!](\mathcal{S}[\![\text{ae1}]\!]\sigma)$$

$$\mathcal{S}[\![\text{ae1 }\underline{\text{plus}}\text{ ae2}]\!]\sigma = \mathcal{S}[\![\text{ae2}]\!](\mathcal{S}[\![\text{ae1}]\!]\sigma)$$

The induction hypothesis may be restated as $\mathit{Sema}[\![\text{ae}]\!]\kappa\sigma=\kappa(\mathcal{V}[\![\text{ae}]\!]\sigma)(\mathcal{S}[\![\text{ae}]\!]\sigma)$ we may then calculate:

$(\kappa*\mathcal{DA}[\![\text{ae1 }\underline{\text{plus}}\text{ ae2}]\!])\sigma$

$$= \kappa(\mathcal{V}[\![\text{ae1 }\underline{\text{plus}}\text{ ae2}]\!]\sigma)(\mathcal{S}[\![\text{ae1 }\underline{\text{plus}}\text{ ae2}]\!]\sigma)$$

$$= \kappa(\mathcal{V}[\![\text{ae1}]\!]\sigma + \mathcal{V}[\![\text{ae2}]\!](\mathcal{S}[\![\text{ae1}]\!]\sigma))(\mathcal{S}[\![\text{ae2}]\!](\mathcal{S}[\![\text{ae1}]\!]\sigma))$$

$$= (\lambda v_1\sigma_1.\kappa(v_1+\mathcal{V}[\![\text{ae2}]\!]\sigma_1)(\mathcal{S}[\![\text{ae2}]\!]\sigma_1))(\mathcal{V}[\![\text{ae1}]\!]\sigma)(\mathcal{S}[\![\text{ae1}]\!]\sigma) \qquad (\beta^{-1})$$

$$= \mathit{Sema}[\![\text{ae1}]\!](\lambda v_1\sigma_1.\kappa(v_1+\mathcal{V}[\![\text{ae2}]\!]\sigma_1)(\mathcal{S}[\![\text{ae2}]\!]\sigma_1)) \qquad \text{(IH)}$$

$$= \mathit{Sema}[\![\text{ae1}]\!](\lambda v_1\sigma_1.((\lambda v_2\sigma_2.\kappa(v_1+v_2)\sigma_2)(\mathcal{V}[\![\text{ae2}]\!]\sigma_1)(\mathcal{S}[\![\text{ae2}]\!]\sigma_1))) \qquad (\beta^{-1})$$

$$= \mathit{Sema}[\![\text{ae1}]\!](\lambda v_1\sigma_1.\mathit{Sema}[\![\text{ae2}]\!](\lambda v_2\sigma_2.\kappa(v_1+v_2)\sigma_2)\sigma_1) \qquad \text{(IH)}$$

$$= \mathit{Sema}[\![ae1]\!](\lambda v_1.\mathit{Sema}[\![ae2]\!](\lambda v_2.\kappa(v_1+v_2))) \qquad (\eta\text{-reduction})$$

$$= \mathit{Sema}[\![ae1\ \underline{plus}\ ae2]\!]\eta$$

The strategy of inverse beta-reduction in the lines marked (β^{-1}) was independently discovered and named "factorization" in both [24] and [17].

5. Let-expressions. We again use the auxiliary equations.

$$\mathcal{DA}[\![\underline{let}\ I\ \underline{be}\ ae1\ \underline{in}\ ae2]\!]\sigma$$

$$= (\lambda v_1\sigma_1.((\lambda v_2\sigma_2.<v_2,\sigma_2[\sigma_1 I/I]>)*\mathcal{DA}[\![ae2]\!])(\sigma_1[v_1/I]))$$
$$(\mathcal{V}[\![ae1]\!]\sigma)(\mathcal{S}[\![ae1]\!]\sigma)$$

$$= ((\lambda v_2\sigma_2.<v_2,\sigma_2[\mathcal{S}[\![ae1]\!]\sigma I/I]>)*\mathcal{DA}[\![ae2]\!])((\mathcal{S}[\![ae1]\!]\sigma)[\mathcal{V}[\![ae1]\!]\sigma/I])$$

$$= (\lambda v_2\sigma_2.<v_2,\sigma_2[\mathcal{S}[\![ae1]\!]\sigma I/I>)(\mathcal{V}[\![ae2]\!]((\mathcal{S}[\![ae1]\!]\sigma)[\mathcal{V}[\![ae1]\!]\sigma/I]))$$
$$(\mathcal{S}[\![ae2]\!]((\mathcal{S}[\![ae1]\!]\sigma)[\mathcal{V}[\![ae1]\!]\sigma/I]))$$

Hence,

$$(\kappa*\mathcal{DA}[\![\underline{let}\ I\ \underline{be}\ ae1\ \underline{in}\ ae2]\!])\sigma =$$

$$= (\lambda v_2\sigma_2.\kappa v_2\sigma_2[\mathcal{S}[\![ae1]\!]\sigma I/I])(\mathcal{V}[\![ae2]\!]((\mathcal{S}[\![ae1]\!]\sigma)[\mathcal{V}[\![ae1]\!]\sigma/I]))$$
$$(\mathcal{S}[\![ae2]\!]((\mathcal{S}[\![ae1]\!]\sigma)[\mathcal{V}[\![ae1]\!]\sigma/I]))$$

$$= ((\lambda v_1\sigma_1.(\lambda v_2\sigma_2.\eta v_2\sigma_2[\sigma_1 I/I])(\mathcal{V}[\![ae2]\!](\sigma_1[v_1/I]))$$
$$(\mathcal{S}[\![ae2]\!](\sigma_1[v_1/I])))$$
$$(\mathcal{V}[\![ae1]\!]\sigma)(\mathcal{S}[\![ae1]\!]\sigma)) \qquad (\text{factorization})$$

$$= \mathit{Sema}[\![ae1]\!](\lambda v_1\sigma_1.(\lambda v_2\sigma_2.\eta v_2\sigma_2[\sigma_1 I/I])(\mathcal{V}[\![ae2]\!](\sigma_1[v_1/I])) \qquad (\text{IH})$$
$$(\mathcal{S}[\![ae2]\!](\sigma_1[v_1/I])))$$

$$= \mathit{Sema}[\![ae1]\!](\lambda v_1\sigma_1.\mathit{Sema}[\![ae2]\!](\lambda v_2\sigma_2.\kappa v_2\sigma_2[\sigma_1 I/I])(\sigma_1[v_1/I])) \qquad (\text{IH})$$

$$= \mathit{Sema}[\![\underline{let}\ I\ \underline{be}\ ae1\ \underline{in}\ ae2]\!]\kappa\sigma \qquad \square.$$

REFERENCES

1. R. M. Burstall, and P. J. Landin, "Programs and Their Proofs: An Algebraic Approach" in *Machine Intelligence 4* (B. Meltzer & D. Michie, eds), pp. 17-44. American Elsevier, New York, 1969.

2. R. M. Burstall, and John Darlington, "A Transformation System for Developing Recursive Programs" *J. ACM 24* (1977), 44-67.

3. H. Ehrig, H. J. Kreowski, and P. Padawitz, "Algebraic Implementation of Abstract Data Types: Concept, Syntax, Semantics and Correctness," *Automata, Languages, and Programming, Seventh Colloquium* (1980), (J. W. de Bakker and J. van Leeuwen, eds), pp. 142-156. Lecture Notes in Computer Science, Vol. 85, Springer, Berlin, 1980.

4. J. L. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial Algebra Semantics and Continuous Algebras," *J. ACM 24* (1977), 68-95.

5. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types" *Current Trends in Programming Methodology, IV: Data Structuring* (R. Yeh, ed.) Prentice-Hall, New Jersey, (1978), pp. 80-149.

6. C. A. R. Hoare, "Proving Correctness of Data Representations," *Acta Informatica 1* (1972), 271-281.

7. J. McCarthy, and J. Painter, "Correctness of a Compiler for Arithmetic Expressions," in *Proc. Symp. in Appl. Math., Vol. 19, Mathematical Aspects of Computer Science* (J. T. Schwartz, ed.), pp. 33-41. Amer. Math. Soc., Providence, R.I., 1967.

8. R. Milne, and C. Strachey, *A Theory of Programming Language Semantics*, Chapman & Hall, London, and Wiley, New York, 1976.

9. R. Milner, and R. Weyhrauch, "Proving Compiler Correctness in a Mechanized Logic," in *Machine Intelligence 7* (B. Meltzer & D. Michie, eds), pp. 51-72. Edinburgh University Press (1972).

10. F. L. Morris, "Advice on Structuring Compilers and Proving Them Correct," *Proc. ACM Symp. on Principles of Programming Languages* (Boston, 1973), 144-152.

11. J. H. Morris and B. Wegbreit, "Subgoal Induction," *Comm. ACM* 20 (1977), 209-222.

12. P. Mosses, "A Constructive Approach to Compiler Correctness," *Automata, Languages, and Programming, Seventh Colloquium* (1980) (J. W. deBakker and J. van Leeuwen, eds.), pp. 449-469. Lecture Notes in Computer Science, Vol. 85, Springer, Berlin, 1980.

13. G. D. Plotkin, "LCF Considered as a Programming Language," Theoret. Comp. Sci. 5 (1977) 223-255.

14. T. W. Pratt, Programming Languages: Design and Implementation, Prentice-Hall, Englewood Cliffs, N.J. 1975.

15. J. C. Reynolds, "Definitional Interpreters for Higher-Order Programming Languages," Proc. ACM Nat'l. Conf. (1972), 717-740.

16. J. C. Reynolds, "On the Relation Between Direct and Continuation Semantics," Proc. 2nd Colloq. on Automata, Languages, and Programming (Saarbrucken, 1974) Springer Lecture Notes in Computer Science, Vol. 14 (Berlin: Springer, 1974), pp. 141-156.

17. R. Sethi and A. Tang, "Transforming Direct into Continuation Semantics for a Simple Imperative Language," unpublished manuscript (April, 1978).

18. R. Sethi and A. Tang, "Constructing Call-by-Value Continuation Semantics," J. ACM 27 (1980), 580-597.

19. J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, MA 1977.

20. G. J. Sussman and G. L. Steele, Jr., "SCHEME: An Interpreter for Extended Lambda Calculus," Mass. Inst. of Tech., AI Memo 349 (December, 1975).

21. J. W. Thatcher, E. G. Wagner, and J. B. Wright, "More on Advice on Structuring Compilers and Proving Them Correct," in Automata, Languages, and Programming, Sixth Colloquium, Graz, July 1979, (H. A. Maurer, ed.), pp. 596-615. Lecture Notes in Comp. Sci., Vol. 71, Springer, Berlin (1979).

22. A. S. Troelstra, (ed.) Methamathematical Investigation of Intuitionistic Arithmetic and Analysis, Lecture Notes in Mathematics, Vol. 344, Springer, Berlin, 1973.

23. M. Wand, "First-Order Identities as a Defining Language," Acta Informatica 14 (1980), 337-357.

24. M. Wand and D. P. Friedman, "Compiling Lambda Expressions Using Continuations and Factorizations," J. of Computer Languages 3 (1978), 241-263.

25. M. Wand, "Final Algebra Semantics and Data Type Extensions," J. Comp. & Sys. Sci. 19 (1979), 27-44.

26. M. Wand, "Continuation-Based Program Transformation Strategies," J. ACM 27 (1980), 164-180.

27. M. Wand, "Specifications, Models, and Implementations of Data Abstractions," Indiana University Computer Science Department Technical Report No. 88 (March, 1980), to appear, Theoretical Computer Science.

28. M. Wand, "Deriving Target Code as a Representation of Continuation Semantics," Indiana University Computer Science Department Technical Report No. 94 (June, 1980), submitted for publication.