A Scheme for a Higher-Level Semantic Algebra

William Clinger, Daniel P. Friedman, and Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

TECHNICAL REPORT NO. 133

A Scheme for a Higher-Level Semantic Algebra

by

William Clinger, Daniel P. Friedman, and Mitchell Wand

May, 1982 – Revised May, 1983

This material is based on work supported by the National Science Foundation under grant MCS 79-04183.

A preliminary version of this paper was presented at the US-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau, France, June 1982.

A Scheme for a Higher-Level Semantic Algebra

William Clinger, Daniel P. Friedman, and Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

May, 1982
revised May, 1983

This Material is based on work supported by the National Science Foundation under grant MCS79-04183.

A preliminary version of this paper was presented at the US-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau, France, June 1982.

1 Introduction

In order to model the behavior of a programming language, we must not only provide some domains, but also some operations on those domains. By doing this, we make the domains into an algebra. The issue we address in this paper is that of choosing an appropriate set of operations. If we choose the right set of operators, then we will be able to model many common languages in the algebra. The typical choice of operators makes the domains into a model of the λ-calculus. Several authors have observed that such complete generality is not necessary [Mosses 80, Raoult & Sethi 83, Wand 80, 82a, 82b]. In this paper we will present a set of operators which seem promising in their generality and which have the interesting property that they turn the semantic domains not merely into an algebra, but into a programming language. Thus writers of semantic equations can use some additional programming intuition when writing equations. Furthermore, if we can compile the algebra, then we can compile the source language, as was pointed out by [Mosses 80].

The starting point for this investigation was the programming language Scheme. Scheme is a lexically-scoped, full-funarg, applicative-order dialect of LISP. In a series of papers, Steele and Sussman [76, 78a, 78b] have shown how many programming language features can be translated into Scheme. As part of a comparative programming languages course, we translated still more constructs into Scheme. This experience made Scheme seem promising as a general-purpose intermediate language– we could translate the source language into Scheme and then use the semantics of Scheme to translate into the λ-calculus or other desired model. We then set out to see if this composite translation was in fact identical with the traditional approach of semantic equations. It turned out the semantics of Scheme was not quite right, but the necessary changes were suprisingly minor.

A second thread in the development of the algebra was that current semantic definitions seem to allow only a limited number of patterns of scoping. Almost all scoping in "conventional" languages (e.g. Algol, Pascal, LISP) is either static, fluid (dynamic), or global. Our language provides standard mechanisms for dealing with each pattern of use. In addition, the language provides orthogonality by providing essentially a single binding mechanism and a single sequencing mechanism.

The result is a Scheme-like semantic algebra. We can translate a source language into terms of the semantic algebra. The definition of the semantic algebra then gives a translation into the underlying domains. Alternatively, we may compile the terms of the semantic algebra into a suitable machine model [Wand 82a].

We sketch here the major comparisons with Mosses' work. Our picture owes a great deal to Mosses, who suggested that a single semantic algebra might provide a target for a variety of source languages [Mosses 80]. Our algebra is concrete, being based on a particular choice of domains, whereas Mosses provided an equational presentation for his algebras. (Our algebra is, however, abstract in a somewhat different sense; we will address this issue in Section 6.) Our algebra has a type structure which is much simpler than that in [Mosses 82].

In Section 2 we present the domains and some useful auxiliaries. Section 3 presents the algebra itself and illustrates its use with several short examples. Section 4 adds some useful syntactic sugar and gives some more examples. In Section 5 we present a longer example: the language in Mosses' paper on binding algebras [Mosses 81]. In Section 6 we discuss our conclusions.

2 Domains and auxiliaries

Table 1 presents our domains and some useful functions on them. We have presented the domains and auxiliaries using fairly standard notation:

$$\lambda x_1 \ldots x_k .\, body \quad \text{abbreviates} \quad (\lambda x_1 (\lambda x_2 \ldots (\lambda x_k\, body) \ldots))$$

and application associates to the left, as usual. We have used *pair, lson,* and *rson* to denote (non-surjective) pairing and projection functions.

Domains

Var	Variables (a flat domain)
L	Locations (a flat domain)
V_0	User Values (unspecified)
$Bool$	Booleans
Ans	Answers (unspecified)
$Env = Var \rightarrow V$	Environments
$S = L \rightarrow V +$ "unused"	Stores
$V = V_0 + F_0 + F_1 + L + Bool$	Values
$C = S \rightarrow Ans$	Command continuations
$K = Env \times [V \rightarrow C]$	Continuations
$F_0 = K \rightarrow C$	0-place abstractions
$F_1 = V \rightarrow K \rightarrow C$	1-place abstractions
$A = Env \rightarrow K \rightarrow C$	Actions

Useful Auxiliaries

$$pair,\ lson,\ rson — primitives for pairings$$

$$send = \lambda \kappa v . rson\, \kappa v$$

$$const = \lambda \kappa v \rho . send\, \kappa v$$

$$ext = \lambda\ var\ \rho\ v\, var_2 . (var_2 = var) \rightarrow v, (\rho\ var_2)$$

$$fext = \lambda var\ \kappa\ v . pair(ext\ var (lson\ \kappa)\, v)(rson\ \kappa)$$

$$D_n = \lambda a_1 a_2 \rho \kappa x_1 \ldots x_n . a_1 \rho (pair(lson\ \kappa)(a_2 \rho \kappa x_1 \ldots x_n)) \quad n \geq 0$$

Table 1. Domains and Auxiliaries

We introduce the domain *Var* of variables in place of the usual domain of identifiers. Variables are purely semantic objects, not to be confused with identifiers, which are syntactic objects that may appear in programs and which often denote variables. The valuation from identifiers to variables may be arranged so that certain variables are never the denotation of any identifier; this allows us to use "reserved variables" for occasional special purposes. This is an extension of the treatment of variables in [Mosses 82].

The other unusual feature of the domains is the treatment of continuations. A continuation consists not only of the usual $[V \rightarrow C]$ (the "sequel") but also an environment, which is intended to keep fluid variables. This corresponds to the conventional observation that dynamic scoping refers to the notion of "most recently entered procedure" or the like. Thus error handlers, for example, are dynamically scoped. Thus we will have two separate environments: one for statically scoped variables and one for dynamically scoped variables. The use of a separate fluid environment allows the use of dynamically scoped variables to coexist with the standard efficient techniques for treating lexically scoped variables. The idea of merging the fluid environment and the continuation dates back at least to [Steele & Sussman 76].

In order to hide the internal structure of these continuations as much as possible, we introduce a number of help functions. We use "*send*" to supply a value to the sequel inside a continuation. The function "*const*" is used to generate actions which return constant values. Corresponding to the usual "*ext*" function for extending environments, we use the "*fext*" function for extending fluid environments inside continuations.

Since continuations are no longer functions, we can no longer write things like:

$$\mathcal{E}[\![exp_1]\!]\rho(\lambda v.\mathcal{E}[\![exp_2]\!]\rho\kappa)$$

so we introduce some help functions for steering data to the second component of a continuation. These are the D_n functions, which essentially perform sequencing [Wand 82a]. Thus a typical function application might appear as

$$D_0(\mathcal{E}[\![rator]\!])(D_1(\mathcal{E}[\![rand]\!])(\lambda\rho\kappa f a.fa\kappa))$$

The reader is encouraged to expand this term to check that everything is passed correctly, including the fluid environments. An assortment of similar examples can be found in [Wand 80, 82a, 82b].

3 The Algebra

Table 2 shows the signature of the algebra itself. There are only two sorts, those of variables and actions. Just as we distinguished between variables and identifiers, we may distinguish between actions and expressions; an expression is a syntactic object whose denotation is an action. We may think of an action as being evaluated in a given environment and continuation and returning a single value by sending it to the continuation.

Table 3 presents the meanings of the operations on the domains. In order to increase orthogonality, we have made into constants a number of things which are normally treated as operators. As a result, the only sequencing in the language is that introduced by application. Similarly, the only binding in the language is that introduced by "abs" (and "fabs").

Sorts

Var	Variables
A	Actions

Operations

$$
\begin{aligned}
\mathit{static} &: \mathit{Var} \rightarrow A \\
\mathit{fluid} &: \mathit{Var} \rightarrow A \\[1em]
\mathit{abs} &: \mathit{Var} \times A \rightarrow A \\
\mathit{fabs} &: \mathit{Var} \times A \rightarrow A \\
\mathit{apply} &: A \times A \rightarrow A \\[1em]
\mathit{freeze} &: A \rightarrow A \\
\mathit{thaw} &: \rightarrow A \\[1em]
\mathit{call/cc} &: \rightarrow A \\[1em]
\mathit{branch} &: A \times A \rightarrow A \\
\mathit{true} &: \rightarrow A \\
\mathit{false} &: \rightarrow A \\[1em]
\mathit{cell} &: \rightarrow A \\
\mathit{store} &: \rightarrow A \\
\mathit{deref} &: \rightarrow A
\end{aligned}
$$

Table 2. Syntax of the algebra

$$\begin{aligned}
\mathit{static}\ \mathit{var} &= \lambda\rho\kappa.\mathit{send}\,\kappa\,(\rho\ \mathit{var})\\
\mathit{fluid}\ \mathit{var} &= \lambda\rho\kappa.\mathit{send}\,\kappa\,(\mathit{lson}\ \kappa\ \mathit{var})\\
\mathit{abs}\ \mathit{var}\ a &= \lambda\rho\kappa.\mathit{send}\,\kappa(\lambda v.a(\mathit{ext}\ \mathit{var}\ \rho\ v))\\
\mathit{fabs}\ \mathit{var}\ a &= \lambda\rho\kappa.\mathit{send}\,\kappa(\lambda v\kappa_1.a\rho(\mathit{fext}\ \mathit{var}\ \kappa_1 v))\\
\mathit{apply}\,a_1a_2 &= D_0a_1(D_1a_2(\lambda\rho\kappa f v.fv\kappa))\\
\mathit{freeze}\ a &= \lambda\rho\kappa.\mathit{send}\,\kappa\,(a\rho)\\
\mathit{thaw} &= \mathit{const}(\lambda v\kappa_1.v\kappa_1)\\
\mathit{call/cc} &= \mathit{const}(\lambda v\kappa_1.v(\lambda v\kappa_2.\mathit{send}\ \kappa_1 v)\kappa_1)\\
\mathit{branch}\,a_1a_2 &= \lambda\rho\kappa.\mathit{send}\,\kappa(\lambda v.v \rightarrow (a_1\rho),(a_2\rho))\\
\mathit{true} &= \mathit{const}\,\mathit{true}\\
\mathit{false} &= \mathit{const}\,\mathit{false}\\
\mathit{cell} &= \mathit{const}(\lambda v\kappa_1 s.((\lambda \mathit{loc}.\mathit{send}\ \kappa_1\,\mathit{loc}\,(\mathit{ext}\ \mathit{loc}\ s\ v))(\mathit{find\text{-}unused\text{-}loc}\ s)))\\
\mathit{store} &= \mathit{const}(\lambda v_1\kappa_1.\mathit{send}\,\kappa_1(\lambda v_2\kappa_2 s.\mathit{send}\ \kappa_2 v_2(\mathit{ext}\ v_1 s\ v_2)))\\
\mathit{deref} &= \mathit{const}(\lambda v\kappa s.\mathit{send}\ \kappa(sv)s)
\end{aligned}$$

Table 3. The Algebra

We may now discuss each of the constructs briefly. "*static*" creates an action which looks up a variable in the environment and returns its value. Similarly, "*fluid*" creates an action which looks up a variable in the fluid environment. Because the static and fluid environments are distinct, we can still use the standard techniques for static identifiers, while using a different set of implementation tricks for fluid variables.

"*abs*" creates an abstraction in the conventional way and returns it. Note that our treatment of fluids allows this equation to be written just as if no fluids were present. "*fabs*" creates a fluid abstraction, which is just like an abstraction except that the evaluated actual parameter is used to extend the fluid environment rather than the static environment. Note that these are the only binding mechanisms in the language (cf. "*call/cc*" below).

The sole sequencing operator in the language is "*apply*". The two arguments are evaluated in order and then the value of the first is applied to the value of the second. Presumably the value of the first is an element of F_1; we have not specified error handling, as that should be done as part of the source language (more on this in the conclusions.) This gives essentially call-by-value evaluation.

Since our standard sequencing operation is call-by-value, we need some additional device to control the order of evaluation. In Scheme this is conventionally done by wrapping forms in $(\lambda()\ldots)$. We have chosen to provide special operators to accomplish this. "*freeze*" creates a 0-argument abstraction (a thunk), in which free variables in the argument are frozen to their values at the time that freeze is called. This is what Gordon [79] calls "call by closure." Thus our $(\mathit{freeze}\ a)$ corresponds to Mosses' "delay$\{a\}$! freeze". The static environment can be frozen without loss of generality because variables which are to be evaluated dynamically are distinguished by the "*fluid*" operation. The operation "*thaw*" runs a thunk by applying it to a continuation.

To illustrate these features, we can define a derived operation $\mathit{block} : A \times A \rightarrow A$ by

```
(block a₁a₂) =
     (apply
          (apply(abs X (abs Y (apply thaw(static Y))))
               a₁)
          (freeze a₂))
```

Here X and Y are constants of type *Var* (i.e. identifiers). It is easy to show that

$$(\mathit{block}\ a_1 a_2) = \lambda\rho\kappa.a_1\rho(\mathit{pair}(\mathit{lson}\ \kappa)(\lambda v_1.a_2\rho\kappa)) = D_0 a_1(\lambda\rho\kappa v_1.a_2\rho\kappa)$$

Intuitively, $(\mathit{block}\, a_1 a_2)$ evaluates a_1 and a_2 in order and returns the value of a_2. The definition of block is very similar to that in Scheme [Steele & Sussman 78a]. The use of *freeze* and *thaw* is unnecessary in this case, since *apply* already evaluates its arguments from left to right, but is used to illustrate the use of *freeze* and *thaw* to control the order of evaluation. The calculation also shows that the meaning of "block" is independent of the choice of the identifiers X and Y.

ALGOL 60 style call-by-name could be handled by a translation of the form:

$$\mathcal{C}[\![\langle \textit{proc-name}\rangle(\langle \textit{actual-parameter}\rangle)]\!] =$$
$$(\textit{apply}(\textit{static}\langle \textit{proc-name}\rangle)(\textit{freeze}\langle \textit{actual-parameter}\rangle))$$
$$\mathcal{E}[\![\langle \textit{formal-parameter}\rangle]\!] = (\textit{apply thaw}\langle \textit{formal-parameter}\rangle)$$

Here, of course, the actual syntax of the metalanguage which describes the translation is meant only to be suggestive.

We have been able to code a wide variety of parameter passing regimes using these operators; we hope to report on these elsewhere.

Further control flow capabilities are provided by the "*call/cc*" operator. "*call/cc*" takes as an argument an abstraction (either static or fluid) and passes to the abstraction another abstraction, which takes the role of a continuation. When this "continuation abstraction" is applied, it sends its argument to the continuation of the "*call/cc*." Note that we have made "*call/cc*" a constant. Thus the CATCH operator in Scheme, which does both binding and catching, would be translated as

$$\mathcal{E}[\![(\ \mathbf{catch}\ \ \langle id\rangle\langle exp\rangle)]\!] = (\textit{apply call/cc}(\textit{abs}\langle id\rangle\, \mathcal{E}[\![\langle exp\rangle]\!]))$$

This treatment is similar in syntax to Landin's $\mathcal{J}$-operator [Landin 65]. This approach has the advantage that the argument to *call/cc* could be a fluid abstraction, as in the example in Section 5, or any action that returned an abstraction. Similarly, one might have code of the form $(fx(gx))$, where (fx) returned "*call/cc*" or "*thaw*" or the like, though we haven't yet thought of any good applications for this.

In Scheme, as in most languages, conditionals provide both branching and sequencing; thus in (**if** $e_0e_1e_2$), e_0 is evaluated first. In keeping with the principle of orthogonality, we have provided a form of conditional which is independent of sequencing. $(\mathit{branch}\, a_1 a_2)$ returns a value in F_1, that is, something which expects to be applied to a value. If the value is true, then a_1 is executed; otherwise a_2 is executed. Thus a "standard" conditional would be translated as

$$\mathcal{E}[\![(\ \mathbf{if}\ e_0 e_1 e_2)]\!] = (\mathit{apply}(\mathit{branch}\ \mathcal{E}[\![e_1]\!]\ \mathcal{E}[\![e_2]\!])\ \mathcal{E}[\![e_0]\!])$$

The reader is urged to expand this to see that it coincides with the usual semantic equation. Something like *branch* is typical in stack-oriented languages like FORTH or in microcode for Lisp machines.

Our treatment of the store is relatively standard except that the operators are constants rather than functions; again, this allows us to decouple the primitive actions on the store from sequencing.

4 Syntactic Sugar

In order to specify a translation from a source language to our algebra, we need a concrete syntax for the terms of the algebra. We will adopt a Lisp-like concrete syntax. This choice reflects both the origin of the algebra and our taste; others may prefer other flavors of concrete syntax. Thus we will write $(\mathit{abs}\ \mathit{var}\, a)$ rather than $\mathit{abs}[\mathit{var}, a]$ or $\mathit{abs}\ \mathit{var}\, a$. We also adopt some other syntactic conventions:

1. Occurrences of "*static*" are elided, so that we may write a term of type *Var* anywhere a term of type A may appear.

2. We use concatenation for the operator "*apply*". As usual, we make successive concatenations associate to the left, and use parentheses to control grouping. In keeping with our Lisp heritage, however, we always put at least one set

of parentheses around an outermost application. Hence we write $(a_1\ a_2\ a_3)$ instead of $(\mathit{apply}\,(\mathit{apply}\, a_1\ a_2)\ a_3)$.

3. In general, we elide mention of the valuation (necessarily one-to-one) which maps source-program identifiers to variables. We have done this already in the discussion of "*call/cc*" above. To be strictly correct, we should have introduced a valuation $I : \langle \mathit{identifier} \rangle \rightarrow \mathit{Var}$ and written

$$\mathcal{E}[\![(\ \mathbf{catch}\ \langle id\rangle\langle exp\rangle)]\!] = (\mathit{call/cc}\,(\mathit{abs}\ I[\![\langle id\rangle]\!]\ \mathcal{E}[\![\langle exp\rangle]\!]))$$

The resulting syntax looks very much like Lisp. The non-constant operators in the algebra (*static, fluid, abs, branch,* etc.) play the role of Lisp's "special forms." Other parenthesized expressions are applications, using our "*apply*" rather than functional application, but if the application is of length 3 or more, then it is automatically "Curried" rather than interpreted with Lisp's evlis. Using these conventions, the block example appears as:

$$(\mathit{block}\ a_1 a_2) = ((\mathit{abs}\,X\,(\mathit{abs}\,Y\,(\mathit{thaw}\,Y)))\,a_1\,(\mathit{freeze}\ a_2))$$

We might also abbreviate $(\mathit{abs}\,X\,(\mathit{abs}\,Y\ \mathit{action}))$ by $(\mathit{abs}(X\ Y)\ \mathit{action})$, but there seems no reason to do so for our examples.

We also introduce a derived operator *let*:

$$(\mathit{let}\ \mathit{var}\ a_1 a_2) = ((\mathit{abs}\ \mathit{var}\ a_2)\,a_1)$$

and we write $(\mathit{let}\,(\mathit{var}\ a_1)\,a_2)$ as syntactic sugar. (Scheme programmers have, we admit, a moderately weird sense of taste in "sugar.")

We can now write a somewhat larger example. We modify the call-by-name example above so that the actual parameter is evaluated only once (i.e., it is "memoized"). As before, we pass to the procedure a 0-argument abstraction. In this case, however, the abstraction, when thawed, refers to a cell in the store and executes the code stored in that cell. The piece of code in the cell is self-modifying, however, so that the actual parameter is evaluated only once.

$$\mathcal{C}[\![\langle \textit{proc-name}\rangle(\langle \textit{actual-param}\rangle)]\!] = (\langle \textit{proc-name}\rangle(\textit{by-need}\ \mathcal{E}[\![\langle \textit{actual-param}\rangle]\!]))$$

where

```
(by-need a) =
        (let(ACT(freeze a))
             (let(LOC(cell any))
                  (block
                       (store LOC(freeze
                            (let(ANS(thaw ACT))
                                 (block
                                      (store LOC(freeze ANS))
                                      ANS))))
                       (freeze(thaw(deref LOC))))))
```

Note that the method of parameter transmission is transparent to the procedure body, so long as it thaws every formal parameter:

$$\mathcal{E}[\![\langle \textit{formal-param}\rangle]\!] = (\textit{thaw}\langle \textit{formal-param}\rangle)$$

As a second example, we can write a function which simulates fix:

```
Y = (abs f
       (let(d(abs g(abs x(f(g g)x))))
            (d d)))
```

This code for Y is adapted from [Steele & Sussman 78b]. Given appropriate arithmetic operations, we can then write examples like:

```
fib = (Y(abs f(abs n
             ((branch(const 1)
                   (+(f(- n 1))(f(- n 2))))
             (0? n)))))
```

5 A Benchmark Example

As an example, let us consider the language proposed in [Mosses 81]. This language has declarations of variables (storable cells) and of parameterless procedures. Procedure names are statically bound, but locations are dynamically bound. To handle this example, we set V_0, the domain of user values, to be the domain of integers, and assume we can handle numerical constants in the obvious way.

To illustrate the flexibility of the method, we have added one new feature: error handlers. The production

$$\langle cmd \rangle ::= \textbf{handler}\ \langle id \rangle_1\ \textbf{is}\ (\langle id \rangle_2)\langle cmd \rangle_0\ \textbf{in}\ \langle cmd \rangle_1$$

declares $\langle id \rangle_1$ to be an exception-name in the dynamic scope of $\langle cmd \rangle_1$ and associates the body $(\langle id \rangle_2)\langle cmd \rangle_0$ with it. Anywhere within the dynamic scope of $\langle cmd \rangle_1$, the exception $\langle id \rangle_1$ may be raised by the command

$$\langle cmd \rangle ::= \textbf{raise}\ \langle id \rangle(\langle num \rangle)$$

The intention is that when the handler is called, $\langle id \rangle_2$ will be bound to $\langle num \rangle$. The body $\langle cmd \rangle_0$ may exit normally, in which case control will exit from the block in which the handler was declared, or it may exit by executing the command **resume** , in which case control returns to the point immediately following the **raise** . Since exception names are controlled by the dynamic scope, they are stored in the fluid environment. Table 4 gives the syntax and semantics of this language.

Syntax

$$\begin{aligned}
\langle pgm\rangle &::= \textbf{res}\ \langle id\rangle\ \textbf{in}\ \langle cmd\rangle\\
\langle cmd\rangle &::= \langle cmd\rangle_1;\langle cmd\rangle_2 \mid \langle id\rangle_1 := \langle id\rangle_2 \mid \textbf{var}\ \langle id\rangle := \langle num\rangle\ \textbf{in}\ \langle cmd\rangle\\
&\mid \textbf{call}\ \langle id\rangle \mid \textbf{proc}\ \langle id\rangle\ \textbf{is}\ \langle cmd\rangle_1\ \textbf{in}\ \langle cmd\rangle_2 \mid\\
&\mid \textbf{handler}\ \langle id\rangle_1\ \textbf{is}\ (\langle id\rangle_2)\langle cmd\rangle_0\ \textbf{in}\ \langle cmd\rangle_1 \mid \textbf{raise}\ \langle id\rangle(\langle num\rangle) \mid \textbf{resume}
\end{aligned}$$

Semantics

$$\begin{aligned}
&\mathcal{P}[\![\ \textbf{res}\ \langle id\rangle\ \textbf{in}\ \langle cmd\rangle]\!] = ((\mathit{fabs}\,\langle id\rangle\,(\mathit{block}\ \mathcal{C}[\![\langle cmd\rangle]\!]\,(\mathit{deref}(\mathit{fluid}\,\langle id\rangle)))))(\mathit{cell}\,(\mathit{const}\ \mathit{unbound})))\\
&\quad \text{— the result of the whole program is the contents of the } \textbf{res} \text{ identifier}\\
&\mathcal{C}[\![\langle cmd\rangle_1;\langle cmd\rangle_2]\!] = (\mathit{block}\ \mathcal{C}[\![\langle cmd\rangle_1]\!]\ \mathcal{C}[\![\langle cmd\rangle_2]\!])\\
&\mathcal{C}[\![\langle id\rangle_1 := \langle id\rangle_2]\!] = (\mathit{store}(\mathit{fluid}\,\langle id\rangle_1)(\mathit{deref}(\mathit{fluid}\ id_2)))\\
&\mathcal{C}[\![\ \textbf{var}\ \langle id\rangle := \langle num\rangle\ \textbf{in}\ \langle cmd\rangle]\!] = ((\mathit{fabs}\ id\ \mathcal{C}[\![\langle cmd\rangle]\!])(\mathit{cell}(\mathit{const}\,\langle num\rangle)))\\
&\mathcal{C}[\![\ \textbf{call}\ \langle id\rangle]\!] = (\mathit{thaw}(\mathit{static}\,\langle id\rangle))\\
&\mathcal{C}[\![\ \textbf{proc}\ \langle id\rangle\ \textbf{is}\ \langle cmd\rangle_1\ \textbf{in}\ \langle cmd\rangle_2]\!] =\\
&\quad ((\mathit{abs}\ id\ \mathcal{C}[\![\langle cmd\rangle_2]\!])(\mathit{freeze}(\mathit{call/cc}(\mathit{fabs}\ RETURN\ \mathcal{C}[\![\langle cmd\rangle_1]\!]))))\\
&\mathcal{C}[\![\ \textbf{handler}\ \langle id\rangle_1\ \textbf{is}\ (\langle id\rangle_2)\langle cmd\rangle_0\ \textbf{in}\ \langle cmd\rangle_1]\!] =\\
&\quad (\mathit{call/cc}\ K\ ((\mathit{fabs}\,\langle id\rangle_1\ \mathcal{C}[\![cmd_1]\!])(\mathit{abs}\,\langle id\rangle_2\,(\mathit{abs}\ RESUME(K\ \mathcal{C}[\![cmd_1]\!])))))\\
&\mathcal{C}[\![\ \textbf{raise}\ \langle id\rangle(\langle num\rangle)]\!] = (\mathit{call/cc}((\mathit{fluid}\,\langle id\rangle)(\mathit{const}\,\langle num\rangle)))\\
&\mathcal{C}[\![\ \textbf{resume}\ ]\!] = (RESUME\ \mathit{true})
\end{aligned}$$

Table 4. Mosses' example

A few words of explanation are in order. The equation for **handler** binds the fluid identifier $\langle id\rangle_1$ to a procedure which binds, successively, a message (to $\langle id\rangle_2$) and a resumption point (to *RESUME*). If the body $\mathcal{C}[\![\langle cmd\rangle_1]\!]$ of the handler is finally called and if it exits normally, then it sends its result to the continuation K of the entire block. The equation for **raise** searches in the fluid environment for the appropriate handler, and sends the message $\langle num\rangle$ to it. Since the handler was of the form $(\mathit{abs}\,\langle id\rangle_2\,(\mathit{abs}\ RESUME\ldots))$, the result is a function which expects a continuation; the continuation is conveniently supplied by the *call/cc*. To execute a **resume** , the semantics merely invokes *RESUME.*

6 Conclusions

The algebra we have presented seems to be a good medium for writing semantic equations. Its particular virtue, in comparison with the conventional approach, is that it imposes a discipline on the writer. It forces him to consider the patterns of scoping of each piece of information: whether it is static, fluid, or global. This is an important tool for analysis. We can then consider whether to impose more structure on the static, fluid, or global contexts. For example, one might add files to the global context. Some languages, such as Lisp and Forth, use a global environment, distinct from the store, as well. The orthogonality of the algebra, with the use of a single sequencing mechanism and essentially a single binding mechanism, imposes still more discipline on the writer.

The main departure from Scheme was the separation of storage allocation from the procedure calling mechanism. This change allowed us to control sequencing without allocating storage, as we did in the example of "*block*". Indeed, it was this example that convinced us that a precise correspondence could be made. The other major change was the specification that rators be evaluated before rands, which gave us effective control over sequencing. Scheme specifies that the order of evaluation of the parts in an application is unspecified. One can still control the order of evaluation, but the mechanism for doing so is cumbersome. The other changes from Scheme, such as the introduction of *freeze, thaw,* and *branch,* seem to be consistent with the spirit of the original. Note that though *abs, branch,* etc. look like "special forms" they are actually ordinary operators in the semantic algebra.

There are two important differences between our algebra and the ones proposed by Peter Mosses. The first apparent difference is that our algebras are "concrete" whereas Mosses' algebras are equationally specified. Our algebras, however, are not quite as concrete as they look. In fact, except for the issue of user types, the definition in Table 3 gives an interpretation of the operators in the untyped λ-calculus. Thus any model of the λ-calculus can serve as the carrier for the algebra. (Indeed, even the interpretation is not fully determined, since the pairing function is given axiomatically)

The issue seems to be how much you need to know in order to do the required proofs. It was previously thought that one needed a great deal of information e.g., a coding into the theory of partial orders, to do the hard proofs such as congruences. It now appears that one can get away with a coding into the theory of the λ-calculus plus a modest induction scheme in order to do compiler correctness. Some hints on this were given in [Wand 82a], where the example did not even require an induction scheme. An example illustrating the use of induction is given in [Wand 83]. It would be interesting to see if the proof of the system in [Wand 82a] can be carried out in Mosses' equational system.

A second difference is in the sort structure of the algebras. We have only two sorts, where Mosses' sorts appear to be closed under tupling and exponentiation. Our sort structure appears adequate to handle a wide variety of examples, including the "within" declarations of [Milne & Strachey 76], which we had originally thought would require tuple values.

To make this useful, we need not only a semantic algebra but a metalanguage in which the translation of the source language to the semantic algebra is specified. Such a language would probably be built on the style of a parser generator. In addition, the language would need to include provisions for binding (e.g. the definition for by-need above), type-checking, etc. It is interesting that the metalanguage will have binding, while the semantic algebra does not.

A semantic algebra is close to an UNCOL: a universal intermediate computer language. Some of our design decisions were motivated by this resemblance. For example, we have not built any error handling into the algebra. (The result of $(apply(const2)\ true)$, for example, depends on the choice of the λ-calculus model). Checking for type errors of this flavor should be part of the language, not the algebra. User types in general remain a major unresolved question. We might want to allow V_0, the domain of user types, to be parameterized, as we did in the example of Section 5. An alternative is to code user types in the "system types" using abstractions and cells, in the fashion of Smalltalk objects or Hewitt's actors. Similarly, we will need to modify the store to allow files, global dictionaries, or other pieces of information needed for various languages. Other languages, such as SNOBOL or PROLOG, may have quite different structures which may make our algebra entirely inappropriate as a translation medium. Still, this algebra seems promising for the definition of "garden-variety" languages.

Acknowledgements

We thank Peter Mosses for his insightful comments on the original version of this paper.

References

[Gordon 79]
Gordon, M.J.C. *The Denotational Description of Programming Languages*, Springer, Berlin, 1979.

[Landin 65]
Landin, P.J. "A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I," *Comm. ACM 8* (1965), 89–101.

[Milne & Strachey 76]
Milne, R. and Strachey C. *A Theory of Programming Language Semantics*, Chapman & Hall, London, and Wiley, New York, 1976.

[Mosses 80]
Mosses, P. "A Constructive Approach to Compiler Correctness," *Automata, Languages, and Programming, Seventh Colloquium* (1980).

[Mosses 81]
Mosses, P. "A Semantic Algebra for Binding Constructs" Proc. of Int'l Colloq. on Formalization of Programming Concepts, Peniscola, Spain, April 1981.

[Mosses 82]
Mosses, P. "Abstract Semantic Algebras!" *Proc. TC-2 Working Conference: Formal Description of Programming Concepts II* (D. Bjorner, ed.) (Garmisch-Partenkirchen, 1982), preliminary proceedings, pp. 63–88.

[Raoult & Sethi 83]
Raoult, J.-C. and Sethi, R. "Properties of a notation for combining functions," *J. ACM 30* (1983), 595-611.

[Steele & Sussman 76]
Steele, G.L. Jr. and Sussman, G.J. "LAMBDA: The Ultimate Imperative," Mass. Inst. of Tech. AI Memo 353 (March, 1976).

[Steele & Sussman 78a]
Steele, G.L. and Sussman, G.J. "The Revised Report on SCHEME," Mass. Inst. of Tech. Artif. Intell. Memo No. 452, Cambridge, MA (January, 1978).

[Steele & Sussman 78b]
Steele, G.L. Jr. and Sussman, G.J. "The Art of the Interpreter or, the Modularity Complex (Parts Zero, One and Two)," Mass. Inst. of Tech. Artif. Intell. Memo No. 453, Cambridge, MA (May, 1978).

[Wand 80b]
Wand, M. "Different Advice on Structuring Compilers and Proving Them Correct," Indiana University Computer Science Department Technical Report No.95 (September, 1980).

[Wand 82b]
Wand, M. "Semantics-Directed Machine Architecture" *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.* (1982), 234–241.

[Wand 82c]
Wand, M. "Deriving Target Code as a Representation of Continuation Semantics" *ACM Trans. on Prog. Lang. and Systems 4*, 3 (July, 1982) 496–517.

[Wand 83]
Wand, M. "Loops in Combinator-Based Compilers," to appear, *Conf. Rec. 10th ACM Symp. on Principles of Prog. Lang.* (1983).