State Transition Machines for Lambda-Calculus Expressions

David A. Schmidt *
Computer Science Department
Aarhus University

Abstract

The process of compiler generation from formal definitions of the lambda-calculus is studied. The compiling schemes developed utilize as an object language the set of state transition machines (STMs): automata-like transition sets using first order arguments. An intermediate definition form, the STM-interpreter, is defined and treated as central to the formulation of state transition code. Three compiling schemes are produced: one derived directly from an STM-interpreter; one formulated from a version of Landin's SECD-machine; and one defined through meaning preserving transformations upon a denotational definition of the lambda-calculus itself. The results are compared and some tentative conclusions are made regarding the utility of compiler generation with the use of the STM forms.

* Permanent Address: Computer Science Department, Kansas State University, Manhattan, Kansas 66506, USA.

Introduction

The work in this paper stems from the conjecture that once one has defined a programming language via formal means, a class of natural compilers for the language is implicitly described as well. Selection of a target (object) language produces a compiler from this class. The diverse levels of formal definitions and object languages make it difficult to formalize the actions taken to develop these compilers. Consequently, we explore compiler development from those formal definitions transformable to a primitive operational form, the STM-interpreter. The STM-interpreter utilizes transition rules into an object language of state transition machines (STMs). The lambda-calculus is used as the example source language for the definitions as it is a well known universal language. Three compiling schemes are developed: one derived directly from an STM-interpreter; one formulated from a version of Landin's SECD-machine [7]; and one defined through meaning preserving transformations upon a denotational definition of the lambda-calculus itself [13]. The different starting points provide insight into the techniques of compiler generation via the use of the intermediate form. Finally conclusions are drawn as to the utility of a compiler generation methodology based upon use of the STM forms.

The Object Language and the STM-interpreter

The automata-like language of state transition machines (STMs) is used as the object language for the compiling schemes. Intuitively an STM is a finite state automaton, each state possessing a finite number of first order (non-functional) arguments. The actions upon the arguments are limited to a set of "machine primitive" operations (e.g., addition, concatenation) and are performed when a transition from one machine state to another occurs. The STM is specified by a set of transition rules, each rule stating the possible state, argument pairs traversible from a current control state. Typically decision predicates are allowed so that a state may have more than one possible successor. The STM format provides a structure which is low level but not yet tied to any machine architecture (although sequential machines are natural targets). A transformation analogous to assembly could be applied to obtain concrete object code.

Formally, an STM is a $<\mathcal{S},\mathcal{E}>$ pair, where

$\mathcal{S}$ = $\{s_1, \ldots, s_m\}$, a finite set of state names including the <u>entry</u> <u>state</u> s_1;

$\mathcal{E}$ = $\{s_1 x = sex_1, \ldots, s_j x = sex_j\}$, $1 \leq j \leq m$, a finite set of equations where x is a variable name and sex_i of the form

i) $s_j ex$, $1 \leq j \leq m$, an explicit state transition;

ii) $ex_1\ ex_2$, a computed state transition;

iii) $ex \rightarrow sex_1, sex_2$, a conditional state transition;

where ex is an expression composed of first order constants ('0', '1', '<u>nil</u>', ...), primitive operators defined upon first order objects ('+', '*', $<>$, $\downarrow i, \ldots$), state names, and x.

The right hand sides of the transition equations are to be well formed in the sense that all variable names used in arithmetic expressions are bound to names on the corresponding left hand sides. Typically the formal definition of an STM can be deduced from an informal presentation; we use the informal version in most cases. Also, when a left hand side variable x represents a tuple object $<y_1, \ldots, y_n>$ we sometimes use the tupled form in place of x.

An example of an STM to compute the factorial function (using the primitive operators equality, subtraction, and multiplication) is

$$\mathcal{S}: \quad \{s0,\ s1,\ s2\}$$

$$\begin{array}{llll} \mathcal{E}: & \underline{s0}\ x & \Rightarrow & \underline{s1}\langle x, 1\rangle \\ & \underline{s1}\langle x, y\rangle & \Rightarrow & x = 0 \rightarrow \underline{s2}\ y, \\ & & & \qquad\quad \underline{s1}\langle x-1,\ x * y\rangle. \end{array}$$

A <u>computation history</u> for the above with <u>initial configuration</u> $\underline{s0}\ 2$ is the sequence of configurations

$$\underline{s0}\ 2 \Rightarrow \underline{s1}\langle 2, 1\rangle \Rightarrow \underline{s1}\langle 1, 2\rangle \Rightarrow \underline{s1}\langle 0, 2\rangle \Rightarrow \underline{s2}\ 2.$$

Now the designation of STM-interpreter can be made. We describe a formal definition of a language to be an STM-interpreter if

i) it is an operational interpreter with finite arguments and discrete transition rules;

ii) the interpreter's transition rules are motivated by the syntax of the source language;

iii) for all inputs the computational history of the definition resembles exactly the history produced by a hypothetical state transition machine.

The intention is that if the definitional interpreter's transition rules define STM computation sequences, we can use these same rules to assemble a set of sequences which may be traversed during computation for some specific input program. The implicit assumption is that interpreters are distinguished from compilers in that an interpreter computes one particular computation sequence for some input program whereas a compiler outputs a set of potential computation sequences (represented in finite form) of which one is selected and traversed at "run time". The objective is to use the interpreter's transition rules to produce a compiling scheme which outputs these potential sequences.

The Source Language

The lambda-calculus [2] is used as the source language for the compilers. Following convention, the set of lambda-calculus expressions Exp is the smallest set formed from a set of variables $x, y, z, \ldots$ and symbols $\lambda, (,)$ such that

i) a variable x is a member of Exp, i.e., $x \in \underline{Var} \subseteq \underline{Exp}$;

ii) if $x \in \underline{Var}$ and $B \in \underline{Exp}$ then $(\lambda\, xB)$ is a member of Exp, i.e., $(\lambda\, xB) \in \underline{Abs} \subseteq \underline{Exp}$;

iii) if $M, N \in \underline{Exp}$ then (MN) is a member of Exp, i.e., $(MN) \in \underline{Comb} \subseteq \underline{Exp}$.

Abs is the set of abstractions in the language and Comb is the set of combinations. We abbreviate expressions of the form $(\lambda\, x(B))$ to $(\lambda\, x.\, B)$ and $((MN)P)$ to (MNP). Outermost parentheses will be dropped in many cases.

Using the standard meanings of the terms free variable, bound variable, and closed term [2], $[M/x]B$ denotes the syntactic substitution of expression M for all free occurrences of x in B (with the renaming of bound variables in B to avoid name clashes with free variables in M). This allows description of the rules of conversion:

$$\alpha : \lambda x.\, B > \lambda y.\, [y/x]B$$
$$\beta : (\lambda x.\, B)M > [M/x]B.$$

The α-rule performs renaming of bound variables, and the β-rule performs the binding of an argument to an abstraction form. The utility of the two rules is augmented by the addition of substitution rules:

$$M > M,$$

$$\frac{M > M'}{(\lambda xM) > (\lambda xM')},$$

$$\frac{M > M', \quad N > N'}{(MN) > (M'N')}.$$

The expression $M >^* N$ denotes the application of zero or more of the above rules to M to obtain N.

A lambda-expression is said to be in <u>normal form</u> if it contains no sub-expression (redex) of the form $(\lambda xB)(M)$. A term is said to have a <u>head redex</u> [3] if it is of the form $(\lambda xB)N_1 \ldots N_n$ where $N_1 \ldots N_n$, $n \geq 1$, are themselves lambda expressions. (Note that the term head redex is used differently from that in [15].) A notion we find useful is <u>weak normal form</u>: a lambda expression is in weak normal form when it contains no head redex. For expressions M and N, the expression $M \equiv N$ means that M and N are syntactically identical with the exception of bound variable names, that is, a finite number of applications of the α-rule to M yields N. The following result is well known:

<u>Theorem</u> [3]: If $M >^* N_1$, $M >^* N_2$, and both N_1 and N_2 are in normal form, then $N_1 \equiv N_2$.

The analogue does not hold for expressions in weak normal form - for example, $(\lambda x. x)(y)((\lambda x. x)(y)) >^* y((\lambda x. x)(y))$ and $(\lambda x. x)(y)((\lambda x. x)(y)) >^* yy$. A given expression may reduce to many weak normal forms. However, if we restrict all uses of the β-rule to head redexes only, not allowing use of the substitution rules, the resulting determinism produces a unique weak normal form (if one exists). The expression $M \overset{*}{\rightarrow} N$ denotes zero or more uses of the restricted β-reduction. We study $\overset{*}{\rightarrow}$ because it describes the evaluation of a program by sequential machine - instructions are executed one at a time from first to last; procedure bodies are evaluated only when actual parameters are bound to formal ones. If a lambda-calculus "program" is reducible to ground type (an atomic form such as Var), then $\overset{*}{\rightarrow}$ is exactly leftmost reduction, and the standardization theorem (that leftmost reduction is adequate for discovering normal forms) [3] guarantees that computation using $\overset{*}{\rightarrow}$ will produce the expected result.

A Lambda-Calculus Machine

To obtain insight into the structure of STM-interpreters we develop one to interpret lambda-calculus expressions. The development is based upon the tenets that

i) a lambda-expression represents a state of computation;

ii) application of the β-rule causes a state transition.

The interpreter will use as its set of automaton state names the set of subexpressions of the input lambda-expression (which can be represented by label names, if desired). Binding of variables in β-reduction will be accounted for by the inclusion of an environment argument. An operand stack for handling nested applications is also needed. We introduce the following notation. Given objects $a_i \in D_i$, let $<a_1, a_2, \ldots, a_n>$ denote a member of $D_1 \times D_2 \times \ldots \times D_n$, and define $<a_1, a_2, \ldots, a_n> \downarrow i = a_i$ when $1 \leq i \leq n$. Appending tuple p to list L is given as p:L. The empty list is denoted by $<>$.

We can describe the "type" of a list by using regular equations (see below). To access a list m let $m[x] = <y_1, y_2, \ldots, y_m>$ iff $<x, y_1, y_2, \ldots, y_m>$ is the first tuple in m with first member equal to x (treated as a symbol, not the specific occurrence). Finally, for lambda expression M, let $\mathcal{L}(M)$ define the set of subexpression occurrences ("labels") of M.

We now define the interpreter. The state of the interpreter is given by the triple $\underline{s}$, e, c where

$s \in \mathcal{L}(M)$, the automaton state

$e \in ENV = (\underline{Var} \times \mathcal{L}(M) \times ENV)^*$, the environment

$c \in CONT = (\mathcal{L}(M) \times ENV)^*$, the operand stack.

The transition function $\Rightarrow$ for the interpreter takes interpreter states into interpreter states and is defined as

1.1	$\underline{x}, e, c$	$\Rightarrow \underline{e[x] \downarrow 1}, e[x] \downarrow 2, c$	$\underline{\text{if}}\ x \in \underline{Var}$
1.2	$\underline{(\lambda x B)}, e, \langle a, e' \rangle : c$	$\Rightarrow \underline{B}, \langle x, a, e' \rangle : e, c$	$\underline{\text{if}}\ (\lambda x B) \in \underline{Abs}$
1.3	$\underline{(MN)}, e, c$	$\Rightarrow \underline{M}, e, \langle N, e \rangle : c$	$\underline{\text{if}}\ (MN) \in \underline{Comb}$.

Figure 1.

We underline the automaton state for clarity. The initial state of the machine is given as $\underline{init}(M) = \underline{M}, \langle\rangle, \langle\rangle$; the machine reaches a final state when none of 1.1–1.3 apply. The lambda-expression denoted by an interpreter state is given by the function $\underline{Unload}$:

$$\underline{Unload}(\underline{s}, e, c_1 : c_2 : \ \ldots \ : c_n) = \underline{Real}\ (\underline{s}, e)\underline{Real}(c_1)\underline{Real}(c_2)\ldots\underline{Real}(c_n)$$

where

$$\underline{Real}(s, e) = [\underline{Real}(e[x_1])/x_1][\underline{Real}(e[x_2])/x_2]\ldots[\underline{Real}(e[x_m])/x_m]s$$

and $\{x_1, \ldots, x_n\}$ is the set of free variables in s.

An example of the interpreter's evaluation of an expression is given in figure 2.

We find the following notation useful. Given interpreter states a and b, let $a \Rightarrow b$ denote an application of a transition rule to state a yielding b. Similarly $a \overset{m}{\Rightarrow} b$ denotes m transitions from a to obtain b. Let $Eval_m(M)$ describe $\underline{Unload}(\underline{s}, e, c)$ where $\underline{M}, \langle\rangle, \langle\rangle \overset{m}{\Rightarrow} \underline{s}, e, c$. An execution of $\underline{M}, \langle\rangle, \langle\rangle$ to an $\underline{Unload}$ed final state is denoted by $Eval(M)$.

For $M = ((\lambda\, y((\lambda\, y(yy)))y))(\lambda\, xx))$ let the following numeric labels denote the subexpressions of M:

$$((\lambda\, y((\lambda\, y(y^5y^6)^4)^3y^7)^2)^1(\lambda\, xx^9)^8)^0$$

$$\begin{array}{lcll}
\underline{\text{init}}(M) & = & \underline{0},<>,<> & \\
 & = & \underline{1},<>,<8,<>> & \\
 & = & \underline{2},<y,8,<>>,<> & \text{! let } <y,8,<>> = e1 \\
 & = & \underline{3},e1,<7,e1> & \\
 & = & \underline{4},<y,7,e1>:e1,<> & \text{! let } <y,7,e1>:e1 = e2 \\
 & = & \underline{5},e2,<6,e2> & \\
 & = & \underline{7},e1,<6,e2> & \\
 & = & \underline{8},<>,<6,e2> & \\
 & = & \underline{9},<x,6,e2>,<> & \\
 & = & \underline{6},e2,<> & \\
 & = & \underline{7},e2,<> & \\
 & = & \underline{8},<>,<> & \\
\end{array}$$

$\underline{\text{Unload}}(\underline{8},<>,<>) = \underline{\text{Real}}(8,<>) = (\lambda\, xx)$

Figure 2.

Since the interpreter is to model leftmost β-reduction to weak normal form, its operation is to be consistent with the rule

$$\text{I.} \qquad (\lambda\, xB)MN_1\ldots N_n \underset{I}{\rightarrow} [M/x]BN_1\ldots N_n, \qquad n \geq 0.$$

In the results which follow $M \overset{n}{\underset{I}{\rightarrow}} N$ denotes n applications of rule I to M obtaining N as a result.

The consistency of the machine is guaranteed by the following lemma:

Lemma 1: $\forall n.\ n \geq 0$: if $M \overset{n}{\underset{I}{\rightarrow}} N$ then $\exists m.\ m \geq 0$: $\text{Eval}_m(M) \equiv N$.

The converse also holds.

Lemma 2: $\forall m.\ m \geq 0$: if $\text{Eval}_m(M) \equiv N$ then $\exists n.\ n \geq 0$: $M \overset{n}{\underset{I}{\rightarrow}} N$.

Together the two lemmas yield the main result:

<u>Theorem 1</u>: Eval(M) $\equiv$ N <u>iff</u> M $\overset{*}{\underset{l}{\rightarrow}}$ N and N is in weak normal form.

As the proofs of these properties are somewhat long and tedious, they will not be presented; they can be found in [11].

<u>A Compiling Scheme</u>

The translation rules specified for the interpreter can be used to produce a syntax directed translation scheme (SDTS) for the lambda-calculus. The scheme is defined as an attribute grammar (see [16]), and it generates a set of transitions for a lambda-expression. It is described using the following rules:

3.1 $<\mathrm{Exp} \downarrow m \uparrow n0> \rightarrow x,$ $\quad$ <u>where</u> $n0 = \{\underline{m}, e, c \Rightarrow \underline{e[x] \downarrow 1}, e[x] \downarrow 2, c\}$

3.2 $<\mathrm{Exp} \downarrow m \uparrow n1 \cup p1> \rightarrow$
$\quad \lambda x <\mathrm{Exp} \downarrow m{:}1 \uparrow p1>$ $\quad n1 = \{\underline{m}, e, <a, e'>:c \Rightarrow \underline{m{:}1}, <x, a, e'>:e, c\}$

3.3 $<\mathrm{Exp} \downarrow m \uparrow n2 \cup p2 \cup q2> \rightarrow$
$\quad <\mathrm{Exp} \downarrow m{:}1 \uparrow p2><\mathrm{Exp} \downarrow m{:}2 \uparrow q2>,$ $n2 = \{\underline{m}, e, c \Rightarrow \underline{m{:}1}, e, <m{:}2, e>:c\}$

<u>Figure 3.</u>

The SDTS generates output code by unioning the code for an expression's sons with the transition rule corresponding to the expression itself. Note that unique labels are assigned to the rules through the use of an inherited attribute m, specified in the style of Watt and Madsen [16]. Also the label $\underline{e[x] \downarrow 1}$ denotes a run-time evaluation which will determine a label to insert in that position.

The compilation of the expression of figure 2 is shown in figure 4. The execution of the transition set is exactly that of figure 2. We can call the anonymous control over the traversal of the STM a function NEXT. NEXT receives as its arguments the initial configuration $\underline{M}$,<>,<> and the STM for M. This allows statement of

$\underline{0}$,	e,	c	$\Rightarrow$	$\underline{1}$,	e,	$<8,e>:$ c
$\underline{1}$,	e,	$<a,e'>:$ c	$\Rightarrow$	$\underline{2}$,	$<y,a,e'>:e$,	c
$\underline{2}$,	e,	c	$\Rightarrow$	$\underline{3}$,	e,	$<7,e>:$ c
$\underline{3}$,	e,	$<a,e'>:$ c	$\Rightarrow$	$\underline{4}$,	$<y,a,e'>:e$,	c
$\underline{4}$,	e,	c	$\Rightarrow$	$\underline{5}$,	e,	$<6,e>:$ c
$\underline{5}$,	e,	c	$\Rightarrow$	$\underline{e[y]\downarrow 1}$,	$e[y]\downarrow 2$,	c
$\underline{6}$,	e,	c	$\Rightarrow$	$\underline{e[y]\downarrow 1}$,	$e[y]\downarrow 2$,	c
$\underline{7}$,	e,	c	$\Rightarrow$	$\underline{e[y]\downarrow 1}$,	$e[y]\downarrow 2$,	c
$\underline{8}$,	e,	$<a,e'>:$ c	$\Rightarrow$	$\underline{9}$,	$<x,a,e'>:e$,	c
$\underline{9}$,	e,	c	$\Rightarrow$	$\underline{e[x]\downarrow 1}$,	$e[x]\downarrow 2$,	c

(Note: assume label numbering to be the same as in figure 2.)

Figure 4.

Theorem 2: Eval(M) ≡ NEXT ($\underline{M}$,<>,<>,STM(M)).

The benefits obtained when separating the compiled STM from the context of the interpreter include those typically attributed to compilation. Most importantly, the compilation of a lambda-abstraction less its arguments will produce a state transition machine which performs leftmost evaluation once an enabled initial configuration is supplied. Thus compilation provides the potential evaluation sequences mentioned earlier whereas interpretation does not. The STM is sufficiently low level for easy translation to object code for various computer architectures. The structure of the rules encourages substantial optimization upon the set before execution. This optimization may take the form of traversal of run-time invariant transitions (mixed computation [4]) or elimination of redundant argument structures by alternative descriptions. In addition, if the STM is viewed as a set of general recursive function-like equations, with the state names of the transition rules corresponding to the names of functions, the evaluation of the STM can be mathematically described as the least fixed point [6] of the equation system. We explore later a complementary approach to development when we take a functionally defined semantics and convert it into state transition form. In the sections to follow we will refer to the interpreter developed here as the WNF-machine.

Non-Leftmost Reductions

The simplicity of the WNF-machine depends upon the utilization of the leftmost reduction strategy. Unfortunately, many realistic computation schemes use a non-leftmost strategy - in particular, a combination may be defined such that the operand portion is reduced prior to the application of operator to operand. Such a situation is known by the term of call by value. An example of a call by value function is addition, which requires that the meanings of both its operands be available before application. We let the expression (λ VALxB) denote the call-by-value abstraction whose binding variable is x and body is B. The set of expressions Exp is augmented by the rule for domain ValAbs:

$$\underline{\text{if}}\ x \in \underline{\text{Var}} \text{ and } B \in \underline{\text{Exp}}\ \underline{\text{then}}\ \lambda\ \text{VAL}xB \in \underline{\text{ValAbs}} \subseteq \underline{\text{Exp}}.$$

An appropriate reduction rule for combinations involving ValAbs is

$$(\lambda\ \text{VAL}xB)M > [M/x]B\ \underline{\text{if}}\ M \text{ is in a normal form.}$$

(However, the Church-Rosser property may be violated; see [8] for an example.) In order to draw the WNF-machine closer to existing devices we consider only closed expressions and introduce a set of base constants $\underline{C}$. The principal problem in adjusting for the new abstraction construct is that the strict leftmost evaluation sequencing is violated - evaluation of a combination in which the ValAbs abstraction occupies the operator position requires the evaluation of the operand before the combination itself can be reduced. This action is simulated by introducing a set of marked labels, one for each ValAbs object in the input lambda-expression. Let the marked counterpart of label s be $\bar{s}$. In anticipation of the introduction of base functions to the language, base constants will be handled differently from variables. A new label, val, will be used as a universal label for each base constant in the lambda-expression. The environment argument of val will contain only the base constant value which val represents. This version supports the introduction of δ-rules [3] to the system.

The complete transition set of the augmented WNF-machine is given in Figure 5. The rules for handling ValAbs are 5.5 and 5.6. The s and $\bar{s}$ labels are treated as distinct - for example, rule 5.6 applies only when the first argument of c is of the form $<\bar{u}, e>$. Note the replacement of the e argument on the right hand side of rule 5.4. Base functions are not included in the example, but the reader should have no problem with their insertion.

5.1	$\underline{x}, e, c$	$\Rightarrow$	$e[x]\downarrow 1,\ e[x]\downarrow 2,\ c$	$\underline{\text{if}}\ x \in \underline{\text{Var}}$
5.2	$\underline{(\lambda x B)}, e, <a, e'>$	$\Rightarrow$	$\underline{B},\ <x, a, e'>:e,\ c$	$\underline{\text{if}}\ (\lambda x B) \in \underline{\text{Abs}}$
5.3	$\underline{MN}, e, c$	$\Rightarrow$	$\underline{M}, e, <N, e>:c$	$\underline{\text{if}}\ (MN) \in \underline{\text{Comb}}$
5.4	$\underline{w}, e, c$	$\Rightarrow$	$\underline{\text{val}}, w, c$	$\underline{\text{if}}\ w \in \underline{C}$
5.5	$\underline{(\lambda \text{VAL}\, x B)}, e, <a, e'>$	$\Rightarrow$	$\underline{a}, e', <\overline{\lambda\ \text{VAL} x B}, e>:c$	$\underline{\text{if}}\ (\lambda\ \text{VAL} x B) \in \underline{\text{ValAbs}}$
5.6	$\underline{s}, e, <\bar{u}, e'>:c$	$\Rightarrow$	$\underline{B}, <x, s, e>:e', c$	

$\underline{\text{if}}\ u = (\lambda\ \text{VAL} x B)$ and $(s = \underline{\text{val}}$ or $s \in \underline{\text{Abs}}$ or $\underline{\text{ValAbs}})$

Figure 5.

We also define analogous functions UnloadV and RealV for the new interpreter.

```
UnloadV(s, e, c)  =  Reorder(<> ,<s, e>:c)
where Reorder(i, c)  =  case c of
                              <>  →  RealV(i)
                        <s, e>:c  →  Reorder(i:<s, e>, c)
                        <s̄, e>:c  →  Reorder(<s, e>:i, c)
                        esac.
```

and

$\underline{\text{RealV}}(<s_1, e_1>: \ldots :<s_n, e_n>) = \underline{\text{ExV}}(s_1, e_1)\ldots\text{ExV}(s_n, e_n)$,

$\underline{\text{ExV}}(s, e) = \underline{\text{if}}\ s = \underline{\text{val}}\ \underline{\text{then}}\ e\ \underline{\text{else}}\ \underline{\text{Real}}(s, e)$.

The purpose of the Reorder function is to extract the augmented labels from the c argument, recover the original label, and replace it in its original position. ExV is necessitated by the val label.

An example of the new machine at work is given in figure 6.

For M = $(((\lambda\ \text{VAL}\ x(\lambda\ \text{VAL} y(xy)))(\lambda\ xx))1)$ let the following numeric labels denote the subexpression of M:

$$(((\lambda\ \text{VAL}\ x\ (\lambda\ \text{VAL}\ y\ (x^6 y^7)^5)^4)^2\ (\lambda\ x x^8)^3)^1\ 1^9)^0$$

$\underline{0}$, <>, <> $\Rightarrow$ $\underline{1}$, <> , < 9,<>>

$\underline{2}$, <> , < 3,<>> : < 9,<>>

$\underline{3}$, <> , < $\bar{2}$,<>> : < 9,<>>

$\underline{4}$, < x, 3,<>> , < 9,<>> ! let <x, 3,<>> = e1

$\underline{9}$, <> , < $\bar{4}$, e1>

$\underline{\text{val}}$, 1 , < $\bar{4}$, e1>

$\underline{5}$, < y, $\underline{\text{val}}$, 1>: e1, <> ! let < y, $\underline{\text{val}}$, 1>: e1 = e2

$\underline{6}$, e2, < 7, e2>

$\underline{3}$, <> , < 7, e2>

$\underline{8}$, < x, 7, e2> , <>

$\underline{7}$, e2, <>

$\underline{\text{val}}$, 1, <>

$\underline{\text{Unload}}$(< $\underline{\text{val}}$, 1,<>>) = $\underline{\text{Reorder}}$(< $\underline{\text{val}}$, 1>)
= $\underline{\text{RealV}}$(< $\underline{\text{val}}$, 1>)
= 1.

Figure 6.

In order to show the consistency of the new machine with respect to call by by value reductions, we define rule set II specifying this reduction strategy.

II. 1 $(\lambda\ \text{VAL}\ x B)\ M_1 M_2 \ldots M_n \xrightarrow[\text{II.1}]{} [M_1/x] M_2 \ldots M_n$ if M_1 is in weak normal for

II. 2 $(\lambda\ \text{VAL}\ x B)\ M_1 M_2 \ldots M_n \xrightarrow[\text{II.2}]{} (\lambda\ \text{VAL}\ x B)\ M' M_2 \ldots M_n$ if $M_1 \xrightarrow[\text{II}]{} M'$

As before, $M \xrightarrow[\text{II}]{} N$ denotes an application of either rule, and $M \xrightarrow[\text{II}]{n} N$ denotes n applications of rules from the set. Along with rule I, these axioms constitute the entire β-reduction scheme. We use the terms $\underline{\text{EvalV}}_m$ and $\underline{\text{EvalV}}$ to define evaluation sequences similar to those given earlier but using the augmented rule set with UnloadV. Analogous to the earlier results, the following can be shown:

Theorem 3: EvalV(M) $\equiv$ N iff $M \xrightarrow[II]{*} N$ and N is in weak normal form.

The result is that the new scheme is faithful to rule set II. Theorems with respect to the complete system and combined rule sets I and II are not given but can be easily formulated.

As seen before, the rule set of figure 5 gives an SDTS for the extended lambda-language. Again assuming the existence of an implicit control function NEXT, we state without proof:

Theorem 4: EvalV(M) $\equiv$ NEXT(< $\underline{M}$,<> ,<>>, STM(M)).

Relationship to the SECD Machine

The archetypical lambda-calculus machine is Landin's SECD-machine [7]. We present a brief explanation of its operation and compare it to the WNF-machine. The definition used is derived from one presented by Plotkin [8].

First we define Exp = Val $\cup$ ValAbs $\cup$ Comb $\cup$ $\underline{C}$; the set of environments EN = (Var $\times$ CL)*; and the set of closures CL = (Exp $\times$ EN)*. The state of the SECD machine is a four-tuple S, E, C, D, where

- S $\in$ CL*, a stack of closure values;
- E $\in$ EN , the current active environment;
- C $\in$ (Exp $\cup$ {apply})*, the control string;
- D $\in$ (S$\times$E$\times$C$\times$D)* , the stack of activation records (dump).

The transition function $\Rightarrow$ takes states into states and is given in figure 7. The start state for the machine for closed lambda-expression M is <> ,<> ,M,<> . The machine's final state is of the form Cl,<> ,<> ,<> where Cl $\in$ CL. A function analogous to Unload can be defined to extract the expression denoted by Cl.

We note three major differences between the WNF-machine and the SECD construction:

i) the SECD-machine processes lambda-expressions from right to left while the WNF-machine processes in a left to right order;

ii) the control in the SECD-machine is embodied in the stack argument C, whereas the WNF-machine is driven by its automaton state argument;

iii) the SECD-machine uses a dump D to maintain scopes of free variables while the WNF-machine replicates the scope information locally in the environment argument e associated with each automaton state.

7.1	Cl:S,E,<>,<S',E',C',D'>	⇒	Cl:S',E',C',D'	where Cl ∈ CL
7.2	S,E,x:C,D	⇒	e[x]:S,E,C,D	if x ∈ Var
7.3	S,E,a:C,D	⇒	<a,E>:S,E,C,D	if x ∈ C
7.4	S,E,(λ VAL xB):C,D	⇒	<λVALxB,E>:S,E,C,D	if (λ VAL xB)∈ValAbs
7.5	<λ VAL xB,E'>:Cl:S,E,apply:C,D	⇒	<>,<x,Cl>:E',B,<S,E,C,D>	
7.6	S,E,(MN):C,D	⇒	S,E,N:M:apply:C,D	if (MN) ∈ Comb

Figure 7.

Constructing an SDTS from SECD

Earlier we contrived an interpreter which led to a compiling scheme. Now we use this experience to transform the SECD-machine into a form which gives a suitable SDTS also. The approach is to analyze the structure of the input lambda-expression M and encode into automaton state names all possible sequences of C-values which may arise during M's evaluation. The idea is feasible because the size of C is bounded and C's contents are dependent only upon the form of M. The existence of an STM-interpreter resulting from the approach is assumed, and we move directly to the compiling scheme. The scheme's grammar is augmented with an attribute c, the current contents of the control string. As c is an inherited attribute, we denote it as ↓c and enclose it in the nonterminal symbol; we also add a label attribute m. The attribute is assigned value by the translation portion of the scheme. The SDTS is given in figure 8. The informalities used in earlier schemes appear here also. Note that the application tokens $\underline{ap}$ have been specialized to their point of creation s to give $\underline{aps}$. Additionally, abstractions generate an extra $\underline{pop}$ transition which is used to recover the dump's values after the processing of an abstraction body. The token $\underline{popb}$ is used as the recovery state after evaluating body b. The grammar also generates a pop transition denoted by $\underline{pop0}$ to obtain the final value. Expression M's initial configuration is $\underline{0:pop0}, <>, <>, <>$. An example of compiled code and evaluation is seen in figure 9.

8.1 $<M \uparrow n0 \cup p0> \rightarrow$ $\quad$ $\underline{\text{where}}$ $n0 = \{\underline{pop0}, CI:S, E, D \Rightarrow Unload(CI)\}$
$\quad <Exp \downarrow \underline{pop0} \downarrow 0 \uparrow p0>,$

8.2 $<Exp \downarrow c \downarrow m \uparrow n1> \rightarrow x,$ $\quad$ $n1 = \{\underline{m:c}, S, E, D \Rightarrow \underline{c}, E[x]:S, E, D\}$

8.3 $<Exp \downarrow c \downarrow m \uparrow n2> \rightarrow a,$ $\quad$ $n2 = \{\underline{m:c}, S, E, D \Rightarrow \underline{c}, <\underline{val}, a>:S, E, D\}$

8.4 $<Exp \downarrow c \downarrow m \uparrow n3 \cup p3> \rightarrow$ $\quad$ $n3 = \{\underline{m:c}, S, E, D \Rightarrow \underline{c}, <m:1:popm, x, E>:S, E, D\}$
$\quad \lambda\, Val\ x <Exp \downarrow \underline{popm} \downarrow m:1 \uparrow p3>,$ $\quad$ $\cup \{\underline{popm}, CI:S, E, <t, S', E', D'> \Rightarrow$
$\quad\quad \underline{t}, CI:S', E', D'\}$

8.5 $<Exp \downarrow c \downarrow m \uparrow n4 \cup p4 \cup q4> \rightarrow$ $\quad$ $n4 = \{\underline{m:c}, S, E, D \Rightarrow \underline{m:2:m:1:apm:c}, S, E, D\}$
$\quad <Exp \downarrow \underline{apm:c} \downarrow m:1 \uparrow p4>$ $\quad$ $\cup \{\underline{apm:c}, CI1:CI2:S, E, D \Rightarrow$
$\quad <Exp \downarrow \underline{m:1:apm:c} \downarrow m:2 \uparrow q4>$ $\quad$ $\underline{CI1\downarrow 1}, <>, <CI1\downarrow 2, CI2>:CI1\downarrow 3,$
$\quad\quad <c, S, E, D>\}$

Figure 8.

$$M = ((\lambda\,\mathrm{VAL}\ z.\,z^{2})^{1}\,1^{3})^{0}$$

(Note: use the above label scheme to represent m's values.)

Compilation:

$\underline{\text{pop0}}$, CI:S, E.D ⇒ $\underline{\text{Unload}}$(CI)
$\underline{\text{0:pop0}}$, S, E, D ⇒ $\underline{\text{3:l:ap0:pop0}}$, S, E, D
$\underline{\text{ap0:pop0}}$, CI1:CI2:S, E, D ⇒ CI1↓1, <>, <CI1↓2, CI2>:CI1↓3, <pop0, S, E, D>
$\underline{\text{1:ap0:pop0}}$, S, E, D ⇒ $\underline{\text{ap0:pop0}}$, < 2:pop1, z, E>:S, E, D
$\underline{\text{2:pop1}}$, S, E, D ⇒ $\underline{\text{pop1}}$, E[z]:S, E, D
$\underline{\text{pop1}}$, CI:S, E, <t, S', E', D'> ⇒ $\underline{\text{t}}$, CI:S', E', D'
$\underline{\text{3:l:ap0:pop0}}$, S, E, D ⇒ $\underline{\text{l:ap0:pop0}}$, <val, 1>:S, E, D

Evaluation:

0:pop0, <>, <>, <> ⇒ $\underline{\text{3:l:ap0:pop0}}$, <>, <>, <>
$\underline{\text{l:ap0:pop0}}$, <$\underline{\text{val}}$, 1>, <>, <>
$\underline{\text{ap0:pop0}}$, < 2:pop1, z, <>>:<$\underline{\text{val}}$, 1>, <>, <>
$\underline{\text{2:pop1}}$, <>, < z, <$\underline{\text{val}}$, 1>>, <$\underline{\text{pop0}}$, <>, <>, <>>
$\underline{\text{pop1}}$, < val, 1>, < z, <$\underline{\text{val}}$, 1>>, <$\underline{\text{pop0}}$, <>, <>, <>>
$\underline{\text{pop0}}$, <$\underline{\text{val}}$, 1>, <>, <>
$\underline{\text{Unload}}$(< val, 1>)
1

Figure 9.

The modified SECD scheme produces STMs larger than those seen so far. An optimization step becomes a necessity. Although the SECD interpreter did not directly yield the compiling scheme (and thus is not an STM-interpreter), the results of figure 9 suggest that a somewhat booader definition of the STM-interpreter might be considered, dependent upon the class of transformations allowable. The study of meaning preserving transformations is continued in the next section.

Compilation from Denotational Definitions

We further extend the compiler generation methods by developing a compiling scheme based upon a higher order definition of the lambda-calculus, as given with the denotational definition method of Scott and Strachey [13]. Production of a compiling scheme from a higher order semantic definition presents difficulties not encountered with the low level machines we have dealt with so far. In particular, arguments to the semantic definitions may be function objects and the order of evaluation of the definition may be non sequential in nature.

We examine the standard semantics of the lambda-calculus as given in Stoy [13]. The reader is advised to refer there for notational conventions. The semantic definition uses an evaluation function $\mathcal{E}$, analogous to Eval, and an environment argument e, which is also a function. Unlike the machines examined earlier, the definition maps the input lambda-expression into an abstract object, a denotation. The domain of denotations is called D; this domain is satisfiable in the Scott-models of denotational semantics [12]. In the definitions, we deal with syntactic domain $\underline{\text{Exp}}=\underline{\text{Var}} \cup \underline{\text{Abs}} \cup \underline{\text{Comb}}$. The domains and equations are:

$\mathcal{E} = \underline{\text{Exp}} \to E \to D$	! the evaluation function
$e \in E = \underline{\text{Var}} \to D$	! the environment
$a \in D = D \to D$	! the lambda-calculus model of denotations;
	! note that every object is treatable as a
	! function

10.1	$\mathcal{E}[\![x]\!]\, e =$	$e[\![x]\!]$	! $x \in \underline{\text{Var}}$
10.2	$\mathcal{E}[\![\lambda\, xB]\!]\, e =$	$\lambda a.\ \mathcal{E}[\![B]\!]\, e[a/x]$	! $x \in \underline{\text{Var}}$, $B \in \underline{\text{Exp}}$
10.3	$\mathcal{E}[\![MN]\!]\, e =$	$\mathcal{E}[\![M]\!]\, e\, (\mathcal{E}[\![N]\!]\, e)$	! $M, N \in \underline{\text{Exp}}$

Figure 10.

The right hand sides of 10.1-10.3 are expressions in Scott's LAMBDA language [12]. Thus, the meaning of 'λ ' and 'e[a/x]' in 10.2 are not syntactic lambda-calculus expressions, but LAMBDA notation used to describe abstract denotations. Whatever STM compiling scheme we develop from the above will therefore compile into object code which computes LAMBDA representations of denotations and not syntactic lambda-calculus terms.

Attempting to convert the above definition to an STM interpreter suggests using $\mathcal{E}[\![x]\!]$, $\mathcal{E}[\![\lambda xB]\!]$, and $\mathcal{E}[\![MN]\!]$ to generate the automaton state names - the syntax constructor $[\![\ ]\!]$ is another form of the labels used to this point. Unfortunately the right hand side definitions do not conform to the STM form: 10.3 contains two state names; 10.2 has an "abstraction" performed after a state name traversal; and the environment function is higher order. We can first organize the state name sequencing by converting the definition to a continuation passing form [14]. The equivalent (congruent [10]) definition of figure 10 is given in figure 11. It is adapted from Reynolds [10].

$$
\begin{aligned}
\hbar &= \underline{Exp} \to E \to C \to D' \\
e \in E &= \underline{Var} \to D' \\
c \in C &= D'' \to D' \\
f \in D'' &= D' \to D' \\
a \in D' &= C \to D'
\end{aligned}
$$

11.1 $\hbar[\![x]\!]\ e\ c \quad = e[\![x]\!]\ c$

11.2 $\hbar[\![\lambda xB]\!]\ e\ c = c(\lambda a.\ \hbar[\![B]\!]\ e[a/x])$

11.3 $\hbar[\![MN]\!]\ e\ c \quad = \hbar[\![M]\!]\ e\ (\lambda f.\ f(\hbar[\![N]\!]\ e)\ c)$

Figure 11.

An extra argument $c \in C$, a continuation, is added. The continuation acts as a sequencing device in that the right hand side of each equation has but one semantic function with all its arguments available. An STM-interpreter derived from such a definition uses this function as the automaton state for the equation's right hand side. The introduction of the continuation has added to the complexity of the semantic domains; the D domain has been fractured into two forms, D'', representing an element of D treated as a function, and D', representing an element of D treated as an argument. Further explanation is found in [10].

Now each equation has the desired state, argument form, but the e and c arguments are higher order objects and still not acceptable. A well known technique for reducing functional objects is the introduction of closures [7]. The closures represent names of functions; once all arguments are supplied, the name-plus-arguments are converted to the function-plus-arguments form. The following transformation should make this concept clear; it is based upon the construction given in [9]. Figure 12 shows the <u>defunctionalization</u> of the e and c arguments using closures. All domain definitions have become nonfunctional. The effect of defunctionalization is that even D' and D'' must be represented as closure objects. The three original semantic equations have expanded to seven. This is because each domain which is simplified to closure objects requires a new auxiliary function which converts the closure-plus-arguments set to the original function applied to the arguments. The new equations are seen in 12.4–12.7. The simple structure of the original definition has led to only one new closure object per defunctionalized domain. This explains why the auxiliary functions are stated so simply. Further examples of the technique are given in [9].

$$
\begin{array}{rcll}
h & = & \underline{\mathrm{Exp}} \times E \times C & \to D' \\
\underline{\mathrm{apC}} & = & C \times D'' & \to D' \\
\underline{\mathrm{apD''}} & = & D'' \times D' & \to D' \\
\underline{\mathrm{apE}} & = & E \times \underline{\mathrm{Var}} \times C & \to D' \\
\underline{\mathrm{apD'}} & = & D' \times C & \to D' \\
e \in E & = & \text{mk-e1} = & \underline{\mathrm{Var}} \times D' \times E \\
c \in C & = & \text{mk-c1} = & \underline{\mathrm{Exp}} \times E \times C \\
f \in D'' & = & \text{mk-v1} = & \underline{\mathrm{Var}} \times \underline{\mathrm{Exp}} \times E \\
a \in D' & = & \text{mk-v2} = & \underline{\mathrm{Exp}} \times E
\end{array}
$$

$$
\begin{array}{lll}
12.1 & h[\![x]\!](e,c) & = \underline{\mathrm{apE}}(e, [\![x]\!], c) \\
12.2 & h[\![\lambda xB]\!](e,c) & = \underline{\mathrm{apC}}(c, \text{mk-v1}<x, [\![B]\!], e>) \\
12.3 & h[\![MN]\!](e,c) & = h[\![M]\!](e, \text{mk-c1}<[\![N]\!], e, c>) \\
12.4 & \underline{\mathrm{apC}}(\text{mk-c1}<[\![N]\!], e, c>, f) & = \underline{\mathrm{apD''}}(f, \text{mk-v2}<[\![N]\!], e>, c) \\
12.5 & \underline{\mathrm{apD''}}(\text{mk-v1}<x, [\![B]\!], e>, a, c) & = h[\![B]\!](\text{mk-e1}<x, a, e>, c) \\
12.6 & \underline{\mathrm{apE}}(\text{mk-e1}<x, a, e>, [\![y]\!], c) & = (x=y) \to \underline{\mathrm{apD'}}(a,c), \\
 & & \qquad \underline{\mathrm{apE}}(e, [\![y]\!], c) \\
12.7 & \underline{\mathrm{apD'}}(\text{mk-v2}<[\![N]\!], e>, c) & = h[\![N]\!](e,c).
\end{array}
$$

<u>Figure 12.</u>

The equations of figure 12 are in STM-form. The right hand side of each equation contains a state name from the set $\{ h[\![M]\!] \mid M \in \underline{Exp} \} \cup \{ \underline{apC}, \underline{apE}, \underline{apD'}, \underline{apD''} \}$ and each argument is nonfunctional. The seven equation sets contain much redundancy - for example the constructor names of the closures are unnecessary because each domain has but one closure type. We eliminate the names. Also since the c argument is of the form $<[\![N]\!], e', c>$, the left hand side of 12.2 can be stated as $h[\![\lambda xB]\!](e, <[\![N]\!], e', c>)$, which allows the reduction of the sequence $12.2 \Rightarrow 12.4 \Rightarrow 12.5$ to one equation. This gives

$$\begin{array}{lll}
12.1' & h[\![x]\!](e,c) & = \underline{apE}\,(e, [\![x]\!], c) \\
12.2' & h[\![\lambda xB]\!](e, <[\![N]\!], e', c>) & = h[\![B]\!]\,(<x, <[\![N]\!], e'>, e>, c) \\
12.3' & h[\![MN]\!]\,(e,c) & = h[\![M]\!]\,(e, <[\![N]\!], e, c>) \\
12.6' & \underline{apE}(<x, a, e>, [\![y]\!], c) & = (x=y) \rightarrow \underline{apD'}\,(a,c),\ \underline{apE}\,(e, [\![y]\!], c) \\
12.7' & \underline{apD'}(<[\![N]\!], e>, c) & = h[\![N]\!]\,(e,c).
\end{array}$$

Since $12.6' \Rightarrow 12.7'$ we can collapse the two into one equation, and conversion of the nested closures to tuple form gives figure 13.

$$\begin{array}{ll}
13.1\ h[\![x]\!]\,(e,c) & = \underline{apE}\,(e, [\![x]\!], c) \\
13.2\ h[\![\lambda xB]\!]\,(e, <[\![N]\!], e'>:c) & = h[\![B]\!]\,(<x, [\![N]\!], e'>:e, c) \\
13.3\ h[\![MN]\!]\,(e,c) & = h[\![M]\!]\,(e, <[\![N]\!], e>:c) \\
13.4\ \underline{apE}\,(<x, [\![N]\!], e'>:e, [\![y]\!], c) & = (x=y) \rightarrow h[\![N]\!]\,(e', c), \\
 & \quad \underline{apE}\,(e, [\![y]\!], c).
\end{array}$$

Figure 13.

The results look similar to figure 1, but here the environment lookup function is explicitly provided.

Some explanation is required as to what the equations above truly denote. The scheme of figure 1 translates lambda-expressions to lambda-expressions, the action made possible assuming retention of the input source text and the Unload function. In contrast, the denotational definition translates into abstract denotations. However, in the conversion to defunctionalized form, the denotations themselves (D' and D'') are not obtained unless all arguments to the semantic functions are present. But since the denotations are themselves functions, the LAMBDA abstractions representing the objects never appear! The result is that no form is ever fully expanded to an abstract denotation. The disadvantage of using first order arguments in STMs is that no higher order object can ever be computed as a final result - only base objects or closure names can appear. Another side effect of the conversion is that the definition becomes non-homomorphic (the meaning of a phrase in the language is no longer completely determined by the meaning of its subparts - see 13.1 $\Rightarrow$ 13.4). Such a result is inevitable when converting to a sequential form.

Conclusion

We have described the construction of compiling schemes for STMs from three varied semantic definitions: an explicitly contrived interpreter; an existing machine,one of whose central data structures was altered; and a higher order definition upon which a pair of significant transformations were performed. The intention has been to utilize the STM-interpreter format of each definition as a device to expose the underlying structure of each language definition and to promote an easy conversion to an SDTS from that point. With respect to the former, the arguments to the STM-interpreter forms display the data organization of the original definitions. This was most obvious in the SECD-machine - its rigid operational structure was preserved in its STM form. In contrast the WNF-machine contained a simple structure explicitly oriented towards β-reduction. The form produced by the essentially "structureless" denotational definition could have been overtly reorganized into many different argument structures.

Of equal interest are the transformations applied to the definitions to obtain the STM forms, for they elicit the complementary information as to what features of the definition are counter-productive to easy compilation for primitive sequential machines - conversion of the control of the SECD-machine immediately comes to mind. The imposition of operational constructs in the abstract denotational definition implies that great latitude is available to its implementors for both optimization and error.

It is not surprising that operational definitions lead to compiling schemes, but the STM-interpreter form defines a class of definitions which are especially useful. It can be asked whether the STM restriction is too strong - in particular, can the requirement of finite state control be replaced by a weaker notion? On the other hand, the STM format is itself quite general; a straightforward implementation of the examples in this paper would use heap storage management, and some of the "primitive operations" may require many lines of assembly code to perform. These questions are not considered here. We make a final remark in regard to developing an automated compiling methodology. Using the lambda-calculus as a universal defining language for programming languages semantics suggests a direction - the universal process uses the lambda-calculus SDTS to produce an object code scheme for each construct in the defined programming language. This code set is then utilized as a specific scheme for compiling input source programs in the language. An elaboration of the technique is presented in [5].

Acknowledgment: Neil Jones contributed substantially by his critical reading of an earlier draft of this paper. Thanks also go to Karen Møller for her assistance in organizing the material.

References

[1] Aho, A.V., and Ullman, J.D. The Theory of Parsing, Translation, and Compiling, Volume I, Prentice-Hall, Englewood Cliffs, N.J. (1972).

[2] Church, A. The Calculi of Lambda-Conversion, Annals of Mathematical Studies 6, Princeton Univ. Press, Princeton, N.J. (1951).

[3] Curry, H.B., and Feys, R. Combinatory Logic, Volume I, North-Holland, Amsterdam (1958).

[4] Ershov, A.P. On the Essence of Compilation, in Formal Description of Programming Language Concepts, Neuhold, ed., North-Holland, Amsterdam (1976) 391-420.

[5] Jones, N.D., and Schmidt, D.A. Compiler Generation from Denotational Semantics, PB-113, Aarhus University, Aarhus, Denmark (1979).

[6] Kleene, S.C. Introduction to Metamathematics, North-Holland, Amsterdam (1952).

[7] Landin, P.J. The Mechanical Evaluation of Expressions, Computer Journal 6-4 (1964) 308-320.

[8] Plotkin, G.D. Call-by-Name, Call-by-Value and the Lambda-Calculus, Theoretical Computer Science 1 (1975) 125-159.

[9] Reynolds, J.C. Definitional Interpreters for Higher-Order Programming Languages, Proc. of the ACM National Conference, Boston, (1972) 717-740.

[10] Reynolds, J.C. On the Relation between Direct and Continuation Semantics, Proc. of the Second Colloquium on Automata, Languages and Programming, Saarbrücken, Springer-Verlag, Berlin (1974) 141-156.

[11] Schmidt, D.A. Compiler Generation from Lambda-Calculus Definitions of Programming Languages, Ph.D. Thesis, Kansas State University, Manhattan, Kansas, forthcoming.

[12] Scott, D.A. Data Types as Lattices, SIAM Journal of Computing 5 (1976) 522-587.

[13] Stoy, J.E. Denotational Semantics, MIT Press, Cambridge, Mass. (1977).

[14] Strachey, C., and Wadsworth, C.P. Continuations - A Mathematical Semantics for Handling full Jumps, Technical monograph PRG-11, Oxford University (1974).

[15] Wadsworth, C.P. The Relation between Computational and Denotationa Properties for Scott's Models of the Lambda-Calculus, SIAM Journal of Computing 5 (1976) 488-521.

[16] Watt, D.A., and Madsen, O.L. Extended Attribute Grammars, Report no. 10, University of Glasgow (1977).