COMPILER GENERATION FROM DENOTATIONAL SEMANTICS[T]

Neil D. Jones [1, 2]
David A. Schmidt [2, 3]

Abstract

A methodology is described for generating provably correct compilers from denotational definitions of programming languages. An application is given to produce compilers into STM code (an STM or state transition machine is a flow-chart-like program, low-level enough to be translated into efficient code on conventional computers). First, a compiler φ: LAMC → STM from a lambda calculus dialect is defined. Any denotational definition Δ of language L defines a map $\overline{\Delta}$: L → LAMC, so $\overline{\Delta} \circ \varphi$ compiles L into STM code. Correctness follows from the correctness of φ.

The algebraic framework of Morris, ADJ, etc. is used. The set of STMs is given an algebraic structure so any $\overline{\Delta} \circ \varphi$ may be specified by giving a derived operator on STM for each syntax rule of L.

This approach yields quite redundant object programs, so the paper ends by describing two flow analytic optimization methods. The first analyzes an already-produced STM to obtain information about its runtime behaviour which is used to optimize the STM. The second analyzes the generated compiling scheme to determine runtime properties of object programs in general which a compiler can use to produce less redundant STMs.

1. University of Kansas, Lawrence,Kansas, USA
2. University of Aarhus, Aarhus Denmark
3. Kansas State University, Manhattan, Kansas, USA

This publication contains material which may be used in this author's forthcoming doctoral dissertation.

ǂ Steven S. Muchnick was also involved in the earlier stages of this research.

INTRODUCTION

Recent advances in the formal definition of programming languages [STO77] and the verification of translators constructed to formal specifications ([MOR73], [MIS76], [ADJ79]) have motivated attempts to generate provably correct translators automatically from language specifications ([GAN79], [MOS79], [RAS79]). This paper describes one solution to this problem: a method which, when given a language definition in the style of denotational semantics ([STO77], [GOR79]), will produce a correct translator into a specific target language.

In part I, we establish the existence of universal compilers and compiler generators. First, some definitions are given concerning terminology and the nature of compilation. The target language (a flowchart-like language called State Transition Machines, or STMs) is described, followed by descriptions of the compiling and compiler generation methods. This leads to the definition of a compiling scheme: a formalism for associating with each correctly parsed source program a corresponding object program. Next, a specific scheme is presented which translates lambda-expressions into STM form. The existence of the latter scheme allows us to show that for every denotational definition there is a corresponding scheme which translates source programs into STM code.

Since the object programs produced by this method are inefficient, part II briefly describes optimization techniques. These use the concepts of mixed computation (or partial evaluation) [ERS78] and abstract interpretation (or flow analysis) [COU77]. First, a method is given to optimize a fixed STM, transforming it into an STM in which every state transition performs an action whose effects cannot be predicted at compile time. This method can be applied to the output of a universal compiler. Second, methods are described to perform a flow analysis on an STM scheme, determining at compiler generation time those computations performable at compile time (e.g., symbol table or environment lookups) and those executable at run time. The effect is to split the STM scheme into two parts - a compile time executable portion and a portion generating run time transition rules. This makes possible the automatic generation of compilers which produce more efficient code.

PART I EXISTENCE OF COMPILER GENERATORS

Compilers and Interpreters

The compiler generation process to be described is concerned entirely with semantic issues. The parsing problem is well understood [AhU74], so we assume that the source program is presented in the form of a parse tree π. The set Parsetrees of all parse trees for a specific programming language $\mathcal{L}$ will be structured as an abstract syntax algebra (e.g. [McC63]), so each production is viewed as a tree-construction operator. Assuming that each program π denotes a function from a set of Inputs to a set of Outputs, a semantics maps each program into its denoted Input-Output function. An interpreter realizes this function directly, while a compiler produces an object program whose denotation is the same as the denotation of π.

semantics: Parsetrees → (Inputs → Outputs)

interpreter: Parsetrees × Inputs → Outputs
compiler: Parsetrees → Targetprograms
target semantics: Targetprograms → (Inputs → Outputs)

We will use an object language which is first-order and closer to machine codes, namely the set of state transition machines, or STMs for short.

In a sense one could define a compiler from an interpreter:

$$\text{compiler}(\pi) = \lambda\ i \in \text{Inputs}.\ \text{interpreter}(\pi, i)$$

by freezing the first argument of the interpreter. This approach requires the entire language implementation machinery to be present in the object program, including parts for constructions which may not be present in the program π. Further, the object program compiler(π) is not

specialized to π - for example all while loops in π would be processed by the same part of "interpreter". This contrasts sharply with conventional compiled code, in which distinct parts of the parse tree give rise to distinct parts of the object program. The compilers which we generate will generate code which is specialized in this sense.

Our goals naturally lead to consideration of the language definition itself as a parameter. A denotational semantics of a language $\mathcal{L}$ associates with each syntactic form $A \rightarrow A_1 \ldots A_n$ and each related semantic function $\mathcal{C}$ $\mathcal{C}$: A-trees $\rightarrow$ denotations, a corresponding definition clause

$$A \rightarrow A_1 \ldots A_n : \mathcal{C}[\![A]\!] = \ldots \mathcal{C}[\![A_1]\!] \ldots \mathcal{C}[\![A_n]\!] \ldots$$

This defines $\mathcal{C}[\![\pi_A]\!]$ for a tree π_A of sort A in terms of the denotations of its subtrees. The right side of this equation is an expression, usually in some extension of the lambda-calculus. We will use the term LAMC to describe a suitable extension of the lambda-calculus.

Now let DDs be the set of denotational definitions of programming languages in terms of LAMC. The compiler-interpreter distinctions above naturally generalize as follows:

universal interpreter: UI = DDs x Parsetrees x Inputs $\rightarrow$ Outputs
universal compiler : UC = DDs x Parsetrees $\rightarrow$ Targetprograms
compiler generator : CG = DDs $\rightarrow$ Compilers
compiler semantics : Compilers $\rightarrow$ (Parsetrees $\rightarrow$ Targetprograms)

Note: The term "universal" refers to the fact that the denotational definition is nearly arbitrary, the only restrictions being expressibility in LAMC, and that inputs and outputs be first-order objects (this restriction comes from the fact that our object program language of STMs involves only first-order data sets).

A denotational definition $\mathcal{C}[\![ ]\!]$ may be viewed as a single lambda-expression Δ which denotes a function: Parsetrees $\rightarrow$ (Inputs $\rightarrow$ Outputs). For any parse tree π, the lambda-expression $\Delta(\pi)$ (Δ applied to π) denotes the meaning of π;

this meaning may be applied computationally to an input i by forming $(\Delta(\pi))i$ and β-reducing the expression as far as possible. Actually, many of these reductions will be independent of i. The semantics implementation system SIS of [MOS75] contains a set of rules which β-reduce $\Delta(\pi)$ to normal form in the absence of i. The resulting reduced λ-expression may be considered as an object program for π, so SIS may be regarded as a universal compiler.

From a practical view this approach has two drawbacks: first, the object programs are in the λ-calculus (although some researchers favor higher-order machine languages, e.g. [BER76] and [BAC78]); and second, the compiler may enter an infinite loop if the semantics is badly defined.

Compiling Schemes

We propose the following solution for the above restrictions. Let $\mathfrak{L}_1$, $\mathfrak{L}_2$ be source and target languages. Following [MOR73] and [ADJ79] a compiler com: $\mathfrak{L}_1 \to \mathfrak{L}_2$ may be described by putting appropriate algebraic structures on $\mathfrak{L}_1$ and $\mathfrak{L}_2$ (typically $\mathfrak{L}_1$ becomes a free "syntax" algebra) so that a compiler "com" becomes a homomorphism. A syntax algebra $\mathfrak{L}_1$ is finitely generated, so com may be finitely specified by a compiling scheme which associates with each abstract syntax production $p: A \to A_1 \ldots A_n$ a corresponding operator $\hat{p}$ in the target language algebra.

For generality we want to use the same target language for many source languages. This is easily done by representing $\hat{p}$ as a derived operator, described by an expression involving the natural operators of $\mathfrak{L}_2$.

As an example, any denotational definition Δ may be viewed as a compiling scheme $\overline{\Delta}$: $\mathfrak{L} \to$ LAMC. To do this we put an algebraic structure on LAMC with sort "lambda expression" and operators "apply", "abstraction" etc. Clearly a semantic rule $C[\![A]\!] = \ldots C[\![A_1]\!] \ldots C[\![A_n]\!] \ldots$ associates with each syntax operator (i.e. production) $p: A \to A_1 \ldots A_n$ of $\mathfrak{L}$, a corresponding LAMC derived operator $\hat{p}(a_1, \ldots, a_n) = \ldots a_1 \ldots a_n \ldots$. Given a parse tree π, the LAMC expression $\overline{\Delta}(\pi)$ may be computed by a syntax-directed transduction involving only syntactic substitution.

Compiler Generation

Given a compiler φ: LAMC → STM, where LAMC and STM are the source and target algebras, $\varphi \circ \overline{\Delta}$ defines a compiler from $\mathfrak{L}$ to STM – a parse tree π is compiled by first constructing the LAMC expression $\overline{\Delta}\pi$ and then applying φ to the result. The action of computing $\varphi(\overline{\Delta}\pi)$ constitutes universal compilation. Further, the function cg which takes a denotational definition Δ into $cg(\Delta) = \varphi \circ \overline{\Delta}$ is a natural compiler generator, as the result has functionality $\mathfrak{L}$-Parsetrees → STM.

Formally, since $\overline{\Delta}$: $\mathfrak{L}$ → LAMC is a homomorphism into the derived theory of LAMC, and φ: LAMC → STM is also a homomorphic map, the STM algebra can be extended to an $\mathfrak{L}$-algebra, and φ may also be extended. In this fashion $\varphi \circ \overline{\Delta} : \mathfrak{L} \rightarrow$ STM becomes an $\mathfrak{L}$-homomorphism and thus an $\mathfrak{L}$-compiler. Pragmatically, a compiling scheme which maps $\mathfrak{L}$-terms directly into STM-terms is constructed by treating both $\overline{\Delta}$ and φ as syntax directed transductions. Given production p: $A \rightarrow A_1 \ldots A_n$ of $\mathfrak{L}$ and its corresponding derived operator $\hat{p}(a_1, \ldots, a_n)$ expressed in LAMC, apply φ directly to $\hat{p}$, expanding the LAMC-term into an STM-term with free variables $a_1, \ldots, a_n$. The result is a derived operator for $\varphi \circ \overline{\Delta}$ which can be used for a direct translation from $\mathfrak{L}$ to STM, and which is easily realized as a syntax-directed transduction.

State Transition Machines

We now define the target language previously mentioned. The set of STMs provides a useful target language because an STM at the same time is close to conventional flow charts and has a semantics closely related to the λ-calculus.

An STM is a system of equations which defines a function from one first order data set to another. (A data set is <u>first order</u> if it can be defined by a finite set of possibly recursive set equations involving predefined base sets, +, and x, e.g. ATOM = N + T, LIST = ATOM + LIST x LIST).

An STM possesses a finite number of control states s; with each is associated a local memory state x, ranging over some first order data set. Each equation defines a rule for transition from one state to another (or to a final answer). An STM has a strong resemblance to an automaton or a flow chart, one difference from the latter being that the memory state is not global but is attached to each control state. Another is that control state names can be treated as data, allowing simulation of "computed gotos" and function call/return linkages.

Intuitively, application of an STM to a data value proceeds by a series of state transitions $sv \to s'v' \to s''v'' \to \ldots$, where each transition involves only application of base functions and testing of conditions. An STM is easily translated into efficient code on conventional architectures, since all memory, data binding and control flow activities are explicitly specified (see [KIT80]).

Definition An STM $\mathbb{m}$ is a sequence of transition rules $s_0x = sex_0, \ldots, s_nx = sex_n$ where x is a variable name, $s_0, \ldots, s_n$ are distinct control state names (s_0 is the entry state), and each sex_i has one of the forms

i)	ex	a halt transition, producing a final answer;
ii)	t ex	an explicit transition to control state t; (not necessarily one of $s_0, \ldots, s_n$);
iii)	$ex \to sex', sex''$	a conditional transition; or
iv)	$[ex_1]ex_2$	a pop transition.

In this each ex is an expression built from x and primitive operators such as "+", "=", <...>, ↓i (tupling and subscripting).

□

Aside from the pop transitions, an STM is merely a flow chart represented in the form of a system of equations. An STM to compute n! iteratively might have transition rules

$$s_0x = s_1\langle x, 1\rangle$$
$$s_1x = (x{\downarrow}1 = x{\downarrow}2 \to 1,\ s_1\langle x{\downarrow}1 - 1,\ x{\downarrow}1 * x{\downarrow}2\rangle)$$

The second rule can be syntactically sugared:

$$s_1\langle n, acc\rangle = (n=0 \rightarrow acc,\ s_2\langle n-1,\ n * acc\rangle)$$

An operational semantics for an STM $\mathbb{m}$ is now described. For each initial data object a, $\mathbb{m}$ will have a computation history $s_0 a = t_1 a_1,\ t_2 a_2,\ \ldots$ where the t_i's and a_i's are control states and memory states. The history is built in this manner: For any $i \geq 1$, suppose $\mathbb{m}$ has the transition rule $t_i x = sex_i$. Then $t_{i+1}\ a_{i+1} = next(sex_i, a_i)$ where the "next" function is defined as follows.

Let eval(ex, a) be the value of expression "ex", given that variable x is bound to value a. Then

$$next(sex, a) = \begin{cases} eval(ex, a) & \text{if } sex = ex \\ t\ eval(ex, a) & \text{if } sex = t\ ex \\ next(sex', a) & \text{if } sex = ex \rightarrow sex',\ sex'' \\ & \quad \text{and } eval(ex, a) = \underline{true} \\ next(sex'', a) & \text{if } sex = ex \rightarrow sex',\ sex'' \\ & \quad \text{and } eval(ex, a) = \underline{false} \\ t\langle c, d\rangle & \text{if } sex = [ex_1]\ ex_2, \\ & \quad eval(ex_1, a) = \langle t, c\rangle \text{ and} \\ & \quad eval(ex_2, a) = d \end{cases}$$

Closures and pop transitions are used to naturally model call by name, upward FUNARGs etc. A closure is a tuple $\langle s, v_1, \ldots, v_m\rangle$ whose first component is a control state name. Typically it is used to represent a function $\lambda\ x_{m+1} \ldots x_n.$ ex with free variables, where s is the entry state of an STM to compute "ex", and $v_1, \ldots, v_n$ are the values of the free variables of ex. A pop transition is an expression $[c]\ ex_{m+1} \ldots ex_n$ in a transition rule. Computationally the effect is this: suppose the value of c is the closure $c = \langle s, v_1, \ldots, v_m\rangle$ and v_i is the value of ex_i $(m+1 \leq i \leq n)$; then control is transferred to the state

$$s\langle v_1, \ldots, v_m,\ v_{m+1}, \ldots, v_n\rangle$$

Figure 1 illustrates the ease with which this mechanism allows translation of a recursive definition into STM form, via continuation semantics.

STM RULES

$s_0 x$ = loop <x, <halt>>
loop <x, c> = (x = 0 → [c]1, loop <x-1, <exit x c >>)
exit <x, c, y> = [c] g(x, y)
halt y = y

COMPUTATION FOR x = 2

$s_0 2$ = loop <2, <halt>>
= loop <1, <exit, 2, <halt>>>
= loop <0, <exit, 1, <exit, 2, <halt>>>>
= exit <1, <exit, <halt>>, 1>
= halt g(2, g(1, 1))
= g(2, g(1, 1))

Figure 1. STM for f(x) = if x = 0 then 1 else g(x, f(x-1))

STM Compiling Schemes

The techniques described earlier can be applied to STMs provided we give them a suitable algebraic structure. There will be one carrier, namely the set STM of all state transition machines and one operator for each n = 0, 1, 2, ... used to combine n STMs into a single STM.

To define the combination operator, identify certain control state names $s^1, s^2, \ldots$ as linkage states. Given STMs $\mathbb{m}_1, \ldots, \mathbb{m}_n$ we define

combine $(\mathbb{m}_1, \ldots, \mathbb{m}_n) = \mathbb{m}_1' \ldots \mathbb{m}_n'$ (concatenated equation sequence)

where $\mathbb{m}_1', \ldots, \mathbb{m}_n'$ are obtained as follows:

a) rename states as necessary so no two equations have the same left-hand-side control state

b) identify linkage states $s^1, s^2, \ldots, s^n$ with the entry states of $\mathbb{m}_1, \ldots, \mathbb{m}_n$, respectively.

A compiling scheme $\mathfrak{L} \rightarrow$ STM will associate with each syntax operator $A \rightarrow A_1 \ldots A_n$ of $\mathfrak{L}$ a derived STM operator defined by a term involving "combine". Figure 3 is an example of a compiling scheme φ: SAL $\rightarrow$ STM from a simple assignment language SAL into STM code (a continuation semantics for the same language is given in Figure 2) – note its similarity to the continuation semantics. For example, Figure 3 specifies that

$$\varphi[\![\text{stmt}_1 ; \text{stmt}_2]\!] =$$

$$\text{combine} (\{ s_0 \langle \rho, \sigma, c \rangle = s_0 (\text{stmt}_1) \langle \rho, \sigma, \langle s_1, \sigma, c \rangle \rangle,$$
$$s_1 \langle \rho, \sigma, c \rangle = s_0 (\text{stmt}_2) \langle \rho, \sigma, c \rangle \},$$
$$\varphi[\![\text{stmt}_1]\!], \varphi[\![\text{stmt}_2]\!]$$

where $s_0(\text{stmt}_1)$ and $s_0(\text{stmt}_2)$ are linkage states, identified with the entry states of $\varphi[\![\text{stmt}_1]\!]$ and $\varphi[\![\text{stmt}_2]\!]$.

Note that φ may be easily realized as a syntax-directed transduction.

SYNTACTIC DOMAINS

prog → stmt
stmt → id := id | new id; stmt | stmt; stmt
id → identifier

SEMANTIC FUNCTIONS

run : PROG → N → N (input and output in variable X at location 0)
exec : STMT → ENV → S → C → N

SEMANTIC DOMAINS

loc : LOC = N (location)
ρ : ENV = N × (ID → LOC) (an environment is a pair <max loc allocated, allocation function>)
c : C = S → N (statement continuations)

BASE DOMAINS AND FUNCTIONS (undefined here)

σ : S (store)
inits : N → S (initial store – n in loc 0, 0 elsewhere)
fetch : S × LOC → N (load from store)
update : S × LOC × N → S (store into store)

SEMANTIC EQUATIONS

1. prog → stmt

$$run[\![prog]\!]n = exec[\![stmt]\!]\rho\ inits(n)\ (\lambda\sigma.fetch(\sigma,0))$$

where $\rho = <0, \lambda id.\ id = X \rightarrow 0, \perp>$

2. $stmt \rightarrow stmt_1;\ stmt_2$

$$exec[\![stmt]\!]\rho\ \sigma\ c = exec[\![stmt_1]\!]\rho\ \sigma(\lambda\sigma.exec[\![stmt_2]\!]\rho\ \sigma\ c)$$

3. $stmt \rightarrow$ new id; $stmt_1$

$$exec[\![stmt]\!]\rho\ \sigma\ c = exec[\![stmt_1]\!]\rho'\ \sigma\ c$$

where $\rho' = <\rho\downarrow 1+1, \rho\downarrow 2 + [id' \mapsto \rho\downarrow 1 + 1]>$

4. stmt → id := id'

$$exec[\![stmt]\!]\rho\ \sigma\ c = c(update(\sigma, (\rho\downarrow 2)id', fetch(\sigma, (\rho\downarrow 2)id)))$$

Figure 2. Continuation Semantics of Simple Assignment Language

prog → stmt

1. $s_0\ n = s_0(\text{stmt}) \langle 0, \langle s_1 \gg \text{inits}(n) \langle s_2 \rangle$
2. $s_1\ \text{id}\ c = (\text{id} = X \rightarrow [c]0, \perp)$ {initial environment}
3. $s_2\ \sigma = \text{fetch}(\sigma, 0)$ {final answer in loc 0}

stmt → stmt_1; stmt_2

4. $s_0\ \rho\ \sigma\ c = s_0(\text{stmt}_1)\rho\ \sigma \langle s_1\ \rho\ c \rangle$ {do stmt_1}
5. $s_1\ \rho\ c\ \sigma = s_0(\text{stmt}_2)\rho\ \sigma\ c$ {then do stmt_2}

stmt → <u>new</u> id; stmt_1

6. $s_0\ \rho\ \sigma\ c = s_0(\text{stmt}_1) \langle \rho\downarrow 1+1, \langle s_1 \rho \gg \sigma\ c$ {do stmt_1}
7. $s_1\ \rho\ \text{id}\ c = (\text{id} = \text{id}' \rightarrow [c]\ \rho\downarrow 1+1, [\rho\downarrow 2]\ \text{id}\ c)$ {new env lookup function}

stmt → id := id'

8. $s_0\ \rho\ \sigma\ c = [\rho\downarrow 2]\ \text{id}' \langle s_1\ \rho\ \sigma\ c \rangle$ {find id loc}
9. $s_1\ \rho\ \sigma\ c\ \text{loc} = [\rho\downarrow 2]\ \text{id} \langle s_2\ \text{loc}\ \sigma\ c \rangle$ {find id' loc}
10. $s_2\ \text{loc}\ \sigma\ c\ \text{loc}' = [c]\ \text{update}\ (\sigma, \text{loc}', \text{fetch}(\sigma, \text{loc}))$ {do assignment

<u>Figure 3.</u> Compiling Scheme for Simple Assignment Language

Note: Tuple brackets on state arguments have been omitted, e.g. the functionality of s_0 in line 4 is s_0: ENV x S x C → N. For convenience, inits, fetch, and update are taken as primitive operators upon the memory state σ.

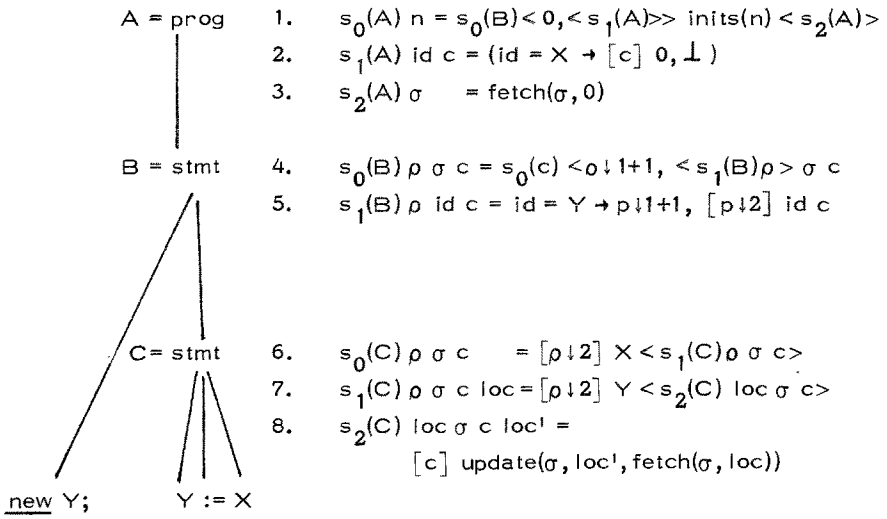


Figure 4. Example of a Compiled Program

A Compiling Scheme from LAMC to STM

Figure 6 contains a scheme for translation of LAMC terms to STM programs. The scheme was developed by applying methods in [REY72] to figure 5 and performing ad-hoc optimization. The conventions and techniques used with figure 3 can also be applied here to give a compiler.

SYNTACTIC DOMAINS

$prog \to ex$

$ex \to con \mid var \mid ex_1\ ex_2 \mid \lambda\ var.\ ex_1 \mid$ base function $\mid$ fix $\lambda\ var.\ ex_1 \mid \ldots$

SEMANTIC DOMAINS

$C = VAL \to A$ continuations

$THUNK = C \to A$ meanings of call-by name operands

$VAL = CON + [THUNK \to THUNK]$

$ENV = VAR \to THUNK$

SEMANTIC FUNCTIONS

$run : CON \to CON$

$ev : EXP \to ENV \to THUNK$

SEMANTIC EQUATIONS

$$run[\![prog]\!]\ v = ev[\![prog]\!](\lambda\ xc.\ \bot)(\lambda\ f.\ f(\lambda\ c.\ cv)(\lambda\ v'.\ v'))$$

$$ev[\![con]\!]\ \rho\ c = c(value[\![con]\!])$$

$$ev[\![var]\!]\ \rho\ c = \rho[\![var]\!]\ c$$

$$ev[\![ex_1 ex_2]\!]\ \rho\ c = ev[\![ex_1]\!]\ \rho(\lambda\ f.\ f(ev[\![ex_2]\!]\ \rho)c)$$

$$ev[\![\lambda\ x.\ ex_1]\!]\ \rho\ c = c(\lambda\ t.\ ev[\![ex_1]\!](\rho + [x \mapsto t]))$$

$$ev[\![basef]\!]\ \rho\ c = c(\lambda\ tc'.\ t(\lambda\ v.\ v \in CON \to c'(basefcn(v)), error))$$

$$ev[\![ex_1 \to ex_2, ex_3]\!]\ \rho\ c = ev[\![ex_1]\!]\ \rho\ (\lambda b.\ b \in CON \to (b \to ev[\![ex_2]\!]\ \rho\ c,\ ev[\![ex_3]\!]\ \rho\ c), error)$$

$$ev[\![\text{fix}\ \lambda\ f.\ ex]\!]\ \rho\ c = ev[\![ex]\!]\ \rho'\ c$$

where $\rho' = \lambda\ x\ c.\ (x = f \to ev[\![ex_1]\!]\ \rho'\ c,\ \rho[\![x]\!]\ c)$

Figure 5. Continuation Semantics of LAMC

prog → ex

s_0 v = s_0(ex) << >, <<op, <s_1>>, finish>>
s_1 c = [c↓1] <v, c↓2>

ex → con

s_0 <ρ, c> = [c↓1] <value⟦con⟧ c↓2>

ex → x

s_0 <ρ, c> = (x = ρ↓1) → [ρ↓2] c, s_0 <ρ↓3, c>

ex → (ex_1 ex_2)

s_0 <ρ, c> = s_0(ex_1) <ρ, << op, <s_0(ex_2), ρ>>, c>>

ex → λx . ex_1

s_0 <ρ, c> = (c↓1↓1 = op) → s_0(ex_1) <<x, c↓1↓2, ρ>, c↓2>,
err

ex → basef ex_1

s_0 <ρ, c> = s_0(ex_1) <ρ, <<s_1>, c>>
s_1 <i, c> = [c↓1] <basefcn(i), c↓2>

ex → ex_1 → ex_2, ex_3

s_0 <ρ, c> = s_0(ex_1) <ρ, <<s_1, ρ>, c>>
s_1 <ρ, i, c> = i → s_0(ex_2) <ρ, c>, s_0(ex_3) <ρ, c>

ex → fix λ x . ex_1

s_0 <ρ, c> = s_0(ex_1) <<x, <s_0, ρ>, ρ>, c>

Figure 6. LAMC to STM Compiling Scheme

Construction and Correctness of STM-Schemes

Earlier we constructed a compiler by composing two syntax directed translation schemes: one, the denotational definition Δ: L → LAMC, the other, a map φ: LAMC → STM. Now we show that the construction method is universal, and computations using the translated programs are correct. The key step in showing correctness is in describing a close correspondence between computation in STMs versus leftmost β-reduction in LAMC. Now, given a φ, such as the one in figure 6, if it can be shown that computation by a translated LAMC expression faithfully simulates β-reduction, then φ can be considered "correct". This theorem is proved in [Sch80] for one such φ; it is shown there that the STM computation simulates leftmost β-reduction upon head redexes [CUF58]. This allows us to state:

Theorem: For each language $\mathfrak{L}$ whose semantics is described by a denotational definition $\mathbb{C}$, there exists a compiling scheme from $\mathfrak{L}$ into STM code which produces programs correct with respect to $\mathbb{C}$.

Proof By the above, $(\overline{\Delta} \circ \varphi)\pi$ is an STM equivalent to π, for any program $\mathfrak{L}$ in π. Further, $\overline{\Delta} \circ \varphi$ may be described as a compiling scheme as follows: For any production $A \to A_1 \ldots A_n$ and semantic rule $\mathbb{C}[\![A]\!] = \ldots \mathbb{C}[\![A_1]\!] \ldots \mathbb{C}[\![A_n]\!] \ldots$, apply the LAMC-STM compiling scheme to $\ldots \mathbb{C}[\![A_1]\!] \ldots \mathbb{C}[\![A_n]\!] \ldots$, and replace each $\mathbb{C}[\![A_i]\!]$ in the result by $s_0(A_i)$. This derived operator on STM associates with each production and semantic rule a finite set of STM transition rules, and so defines the required compiling scheme.

Remark This construction associates with each production a derived operator on the STM algebra, specified as a term involving "combine". These terms may be "flattened" using properties of "combine" to yield a specification in the style of example 3 or 6. For example if C_0 and C_1 are constant STMs then

$$\text{combine}(C_0, \text{combine}(C_1, X)) = \text{combine}(C_0' C_1', X)$$

where C_0' and C_1' are obtained from C_0 and C_1 by renaming control states.

This establishes the existence of compiler generators from (nearly) arbitrary denotational definitions into STM code. Clearly the same ideas could be used to translate into other object languages, or further translation could be applied to the STM object programs (one such example is found in [KIT80]).

Although the fundamental task has been achieved, a number of pragmatic issues need resolution. Foremost is the improvement of the STMs produced for programs in $\mathfrak{L}$. These tend to contain many compile-time evaluable operations, such as symbol table lookups derived from $\mathfrak{L}$'s semantics and trivial interface transitions used to join together STMs corresponding to subtrees of a source program (both may be seen in figure 3). Part II describes the use of flow analysis and mixed computation to produce more economical STMs.

PART II OPTIMIZATION OF THE OBJECT CODE (Overview)

Assume we are given a state transition machine M which computes a function f: IN → OUT. For a given control state, define its argument to be dynamic (or runtime) if it depends functionally on the input to M, and static otherwise. This notion is also generalized to parts of a data item, e.g. a component of a tuple. A transition rule is said to be dynamic if it cannot be carried out without knowledge of the input value, i.e., if a dynamic object is involved in an essential way, such as in a test, as operand of a base function, or as operand of a pop transition.

Note that in the example of figure 3 transition rules 1, 3, 8 and 10 are dynamic. Rules 4, 6, and 9 are used to pass control to other rules associated with other nodes of the input parse tree; these rules are static. Other examples of static computations are the access and update of environments (symbol tables), e.g. rules 2, 6, and 7.

Optimization after STM Creation

A general analysis and optimization of an arbitrary STM M can be achieved in the following stages

i) analyze M to determine which state arguments or parts thereof are static and which are dynamic. Compute the values of the static arguments;

ii) mark those transition rules which are static;

iii) combine each static transition rule with its successor (which is unique since the transition is static). This is known as chain collapsing.

iv) remove those static arguments whose values were computed in i) as they no longer affect the computation. This is known as argument simplification.

These steps create an STM in which each transition and data item depends upon the input arguments. The optimization resembles Ershov's mixed computation [ERS78] in that, to compile, one executes the program as completely as possible in the absence of input data, and outputs as the object program those residual program parts not executable at compile time.

A straightforward symbolic execution is not adequate, due to states which may be repeatedly entered during execution. The solution is to do a flow analysis, or abstract interpretation in the sense of Cousot [COU77] (certain extensions are needed to handle pop transitions). Briefly, the idea is as follows.

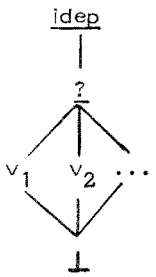
Abstract interpretation is done by associating with each control state s an argument description $\hat{\alpha}(s)$. The values of $\hat{\alpha}(s)$ are elements of a description lattice appropriate to the argument domain of s. Initially, each $\hat{\alpha}(s)$ equals $\perp$ (indicating that nothing is known about the argument of s) except that the entry state s_1 is described by $\hat{\alpha}(s_1)$ = idep (indicating that its argument is input dependent). The program is now abstractly executed in parallel, updating each state descriptor $\hat{\alpha}(s)$ as soon as new information is obtained about the argument of s.

Thus an atomic argument may have one of 4 descriptions: $\perp$ (no information); a specific value, e.g. 17; ?, indicating that the value is not input dependent but is not known at compile time; and idep, input dependent.

We note that the principal domain types of STM states are primitive domains and domain products. These and a third type, the set of closures, are assigned descriptors. Noting that the set of closures is a set of tupled objects, one of whose members is a control state name, the description lattices are as follows.

Argument type T	Description Lattice $\hat{T}$
Primitive domain N	N ∪ {idep} where $\forall x \in N,\ x \sqsubseteq$ idep
Product domain $D_1 x \ldots x D_n$	$\hat{D}_1 x \ldots x \hat{D}_n$
Closure domain $D_1 x \ldots x D_n$ where D_1 is a domain of control state names	2^{D1}, a powerset lattice

In the first case the lattice is of the form



The special treatment given to closure domains stems from their elements' use in pop transitions - it is a control domain. Hence analysis of closure objects analyzes control, viz. the set of possible successor states of a given program state. The abstract interpretation needs to determine a set of possible successors, and so the powerset lattice form is required. To maintain the finite chain property we only record the state; the other components may be recovered from the argument description at the place where the closure was created.

The STM is viewed as defining a continuous fucntion $\mathbb{G}$: AD → AD, where AD is the domain of functions $\lambda s.\hat{\alpha}(s)$. The minimal fixpoint $\underline{fix}\,\mathbb{G}$ is the desired analysis. Figure 7 contains the result of abstract interpretation on the program of figure 4.

Given this information, chain collapsing and argument simplification are straightforward. Drawbacks to this method are:

1. Abstract interpretation is expensive in
   time : $\mathbb{G}$ must be iterated until each αs has converged.
   space : Retention of the memory state descriptions is necessary.
2. Properties invariant with respect to the language definition are rediscovered each time a program is compiled.
3. The algorithm is a post processor rather than a compiler generator.

$$
\begin{aligned}
\alpha\, s_0(A) &= \underline{idep} \\
\alpha\, s_1(A) &= \langle X, \{s_1(C)\}\rangle \\
\alpha\, s_2(A) &= \underline{idep}
\end{aligned}
$$

$$
\begin{aligned}
\alpha\, s_0(B) &= \langle\langle 0, \{s_1(A)\}\rangle, \underline{idep}, \{s_2(A)\}\rangle \\
\alpha\, s_1(B) &= \langle\langle 0, \{s_1(A)\}\rangle, ?, \{s_1(C), s_2(C)\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\alpha\, s_0(C) &= \langle\langle 1, \{s_1(B)\}\rangle, \underline{idep}, \{s_2(A)\}\rangle \\
\alpha\, s_1(C) &= \langle\langle 1, \{s_1(B)\}\rangle, \underline{idep}, \{s_2(A)\}, 0\rangle \\
\alpha\, s_2(C) &= \langle 0, T, \{s_2(A)\}, 1\rangle
\end{aligned}
$$

Figure 7. Result of Abstract Interpretation of Figure 4.

Scheme Analysis and Compiler Generation

Current research centers on the development of methods suited for flow analysis of the compiling schemes themselves. Application of such algorithms to the schemes would determine which elements of the definition are static regardless of the input program being compiled, and hence can be evaluated at compile time. For example, consider the language rule "stmt → new id; $stmt_1$" from the scheme of figure 3. The associated STM rules have two generic states – an entry state $s_0(stmt)\ \rho\ \sigma\ c$, and an environment application state $s_1(stmt)\ \rho$ id c. Analysis of the language scheme would reveal that ρ and c are static in $s_0(stmt)$, but σ is dynamic. For $s_1(stmt)$, all of ρ, id, and c are found to be static in behavior but receive multiple values during compilation (i.e., their flow analytic values

map to '?'). The conclusion is that both transition rules are static (no computation with dynamic arguments occurs). Since s_0(stmt) has a unique successor, it can be collapsed with its successor rule. s_1(stmt) has many predecessors, but only a single successor for each. Thus it may be eliminated in favor of a compile time computation which performs the rule's transition each time it is encountered.

Such an analysis could be used to determine in advance which parts of the scheme must appear in the object program and which parts may be evaluated at compile time. Analysis of figure 3 reveals that all environment and location computations are performable at compile time and that only rules 1, 3, and 10 need appear in object programs.

Further, such analysis could reveal those state arguments which can receive only one value description during abstract interpretation. Using this information and that above, a more efficient compiling algorithm may be visualized which does not explicitly build the full STM, abstractly interpret it, and then reduce it as described above. Instead, the algorithm accomplishes all these effects simultaneously by traversing the parse tree in a way corresponding to the possible flow of control in the STM, keeping in memory only those descriptions of STM states which are needed to do the abstract interpretation. During the traversal, compile time values are computed (e.g. ρ, loc). Whenever a runtime transition rule is encountered, the necessary compile time parameters are inserted (e.g. loc in rule 10) and the residual rule is added to the object program.

This method would appear to have both time and space advantages over the preceding one.

References

[ADJ79] Thatcher, J.W., Wagner, E.G., and Wright, J.B. More Advice on Structuring Compilers and Proving Them Correct, 6th Colloquium, Automata, Languages, and Programming, Graz, Austria, 1979, Springer Lecture Notes in Computer Science 71.

[AhU72] Aho, A.V., and Ullman, J.D. The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Englewood Cliffs, N.J. 1972.

[BAC78] Backus, J. Can Programming Be Liberated from the von Neumann Style? Comm. ACM 21-8, 1978, 613-641.

[BER76] Berkling, K.J. Reduction Languages for Reduction Machines, Rpt. ISF-76-8, Gesellschaft für Mathematik und Datenverarbeitung MbH, Bonn, 1976.

[COU77] Cousot, P., and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Con-Struction or Approximation of Fixpoints, 4th ACM Symposium onPrinciples of Programming Languages, Los Angeles, 1977, 234-252.

[ERS78] Ershov, A.P. On the Essence of Compilation, in Formal Description of Programming Language Concepts, E.J. Neuhold, ed., North-Holland, Amsterdam, 1976, 391-420.

[GAN79] Ganzinger, H. Some Principles for the Development of Compiler Descriptions from Denotational Language Definitions, Tech. Rpt., Technical University of Munich, 1979.

[GOR79] Gordon, M.J.C. The Denotational Description of Programming Languages, Springer-Verlag, Berlin, 1979.

[KIT80] Kitchen, C. Compiling State Transition Machines into Machine Language, M.S. Thesis, University of Kansas, forthcoming.

[McC63] McCarthy, J. Towards a Mathematical Science of Computation, in IFIP 62, C.M. Poppelwell, ed., North-Holland, Amsterdam, 21-28.

[MiS76] Milne, R., and Strachey, C. A Theory of Programming Language Semantics, Chapman and Hall, London, 1976.

[MOR73] Morris, F.L. Advice on Structuring Compilers and Proving Them Correct, 1st ACM Symposium on Principles of Programming Languages, Boston, 1973, 144-152.

[MOS75] Mosses, P.D. Mathematical Semantics and Compiler Generation, Ph.D. Thesis, University of Oxford, 1975.

[MOS79] Mosses, P.D. A Constructive Approach to Compiler Correctness, DAIMI IR-16, University of Aarhus, 1979.

[RAS79] Raskovsky, M., and Turner, R. Compiler Generation and Denotational Semantics, Fundamentals of Computation Theory, 1979.

[REY72] Reynolds, J.C. Definitional Interpreters for Higher-Order Programming Languages, Proc. of the SCM National Conference, Boston, 1972, 717-740.

[REY74] Reynolds, J.C. On the Relation Between Direct and Continuation Semantics, 2nd Colloquium on Automata, Languages, and Programming, Saarbrücken, Springer-Verlag, Berling, 1974, 141-156.

[SCH80] Schmidt, D.A. Compiler Generation from Lambda-Calculus Definitions of Programming Languages, Ph.D. Thesis, Kansas State University, Manhattan, Kansas, forthcoming.

[STO77] Stoy, J.E. Denotational Semantics, MIT Press, Cambridge, Mass., 1977.