Generation of Efficient Compilers
by Application of
Single-Threading, Control Binding and Lambda-Lifting Techniques

by

Kok Hui Chong

B.A., Coe College, Cedar Rapids, IA, 1987

A MASTER'S THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved By:

Dr. David A. Schmidt

Acknowledgements

I would like to express my gratitute and appreciation to Professor David A. Schmidt for his guidance, assistance, and advice.

I wish to thank Professor Elizabeth A. Unger and Professor Massaki Mizuno for being on my examination committee and teaching me interesting courses during my stay at the Kansas State University.

Finally, I would like to thank my wife Bee Choo and my parents for their patience and encouragement during my graduate study.

Table of Contents

Chapter 1

Introduction

As the practical value of **denotational** semantics becomes better understood, it has become obvious that the implementation of a language can be guided by its semantic definition [5]. In other words, it is feasible to derive a compiler from the semantics of a language. If one makes a comparison between a conventional handwritten compiler and a compiler generated from a semantic definition, one finds that it is easier to produce an **error-free** compiler using the semantic definition, although it first requires writing the semantic definition of the language. A drawback with the early work in this area is that the derived compilers ran slower than the handwritten ones. This is because the existing compiler generating systems that process lambda-calculus-style denotational semantics [8] are hindered by the slow processors that generate inefficient target code.

One possible way around this problem is to develop new machine architectures that are better suited to the implementation of functional languages. However, the dark side of following this approach is that it is not economically feasible; the existing machines use Von Neumann architecture must be wastefully discarded and replaced with the new ones. Fortunately, software solutions come to the rescue. Clues presented by the domains and valuation functions in the semantic definitions open new avenues for researchers to transform a denotational definition of a programming

language into an easily implementable form to generate more efficient compilers [4]. As the search for solutions continues in this direction, several promising techniques have been formulated to improve the efficiency of the generated target language. Among these techniques, we considered single-threading, control binding and lambda-lifting [4,6,7,8]. Although these techniques seem to improve the efficiency of the generated target language, no efforts have been conducted to tie together these techniques to maximize their effectiveness.

The motivation of this research is to fill in the gap. Since the order in which these techniques are employed affects their performances, we need a system which enables us to intermix them in various orders and capture their best ordering. Following this idea, we designed and implemented these techniques in separate modules and tested them with individual lambda calculus expressions. After these independent modules have been successfully built, their interfaces were properly defined so that they can be glued together in any order and executed. The order which we eventually pick will be the order which yields the smallest and best result. As the next step of the research, the augmented system is applied to a set-of-equations semantic definition. The output from the augmented system together with a compile-time evaluator form a compiler. With these efforts, we hope we can open a new dimension of automatically generated compilers that look like and are run as efficiently as hand-written ones.

The compiler generator system is primary used to generate efficient compilers from imperative language definitions. This is due to the fact that the techniques which are incorporated in the system are useful for processing semantics of imperative languages that use storage. Besides that, the system can also be applied to some kinds of functional programs where parameters are passed in a sequential fashion.

There are two obvious limitations to the system. First, it requires language definitions which use eager evaluation. Second, it cannot handle conversion of functional program parameters to variables in many cases.

Contents of Thesis

In the next chapter we give a review of the typed lambda calculus and rewriting rule schemes. The concepts of Single-Threading, Control Binding and Lambda-Lifting are presented in Chapter 3. Chapter 4 and 5 discuss work that was done in the research. These chapters show the structure of the system and how the individual components were implemented. Results are given in Chapter 6. Chapter 7 talks about the compile-time and run-time evaluators. Finally, Chapter 8 contains conclusions. The source code for the compiler generator system is contained in appendix A.

Chapter 2

Typed Lambda Calculus and Rewriting Rule Schemes

2.1. Typed Lambda Calculus.

In the denotational semantics framework, the denotation of a program is usually a mathematical value, such as a number of a function [5]. Denotations are expressed in a simple language called the lambda calculus. The lambda calculus has only a few syntactic constructs and a simple semantics. Despite it simplicity, it is sufficient to express the meaning of most all programming languages (*e.g. PASCAL, LISP, SMALLTALK*). Since our compiler generator system processes semantics definitions encoded in the lambda calculus, understanding the system will require some knowledge of the lambda calculus. FIGURE 2-1 shows the concrete syntax of the **typed lambda calculus**. The domain (type) calculus includes first order domains (*e.g., nat, bool, iden, store, cmd, expr and numeral*), and function space domains (*e.g., (nat->nat), ((iden->store)->nat)*). Our choices of constants is arbitrary. Constants can be built-in functions (*e.g., update, access and plus*), natural numbers (*e.g., zero, one and two*) or booleans (*e.g., true and false*).

Based on the concrete syntax above, we give three samples of typed lambda calculus expressions:

(a). ((times ((plus one) two)) four)

```
E : Expression
D : Domain
T : First-order-domain
i : identifier
c : constant

D :: = T | ( D1 -> D2 )
E :: = i | c | lam i : T . E mal | ( E1 E2 )
```

FIGURE 2-1

(b). lam i : iden . lam s : store . ((access i) s) mal mal

(c). lam f : (nat -> nat) . (f two) mal.

(For practical reasons, all function applications in the typed lambda calculus are written in prefix form.) A lambda expression is itself a kind of "program" that can be "computed" by rewriting it into a normal form. The rewriting is called a reduction, and the rewriting is done by rewriting rule schemes.

2.2. Rewriting Rule Schemes.

A **rewriting rule** is a form L => R. An expression E is rewritten by the rule when a subexpression of E matches L. The matched subexpression in E is replaced by R. Before we give examples, we need some definitions.

2.2.1. Definition.

(i) A **lambda abstraction** is an expression of the form lam i:T.E mal. Expression (b) in section 1.1 is an example.

(ii) A lambda expression is **closed** if every identifier 'i' within it appears within a lambda abstraction lam i.E mal. Expression (b) is closed because the only identifiers in it are s and i, and they reside within the abstraction (lam i. lam

s. B). An expression that is not closed is **open.** As we will see in section 2.3, implementing an open expression is a nuisance and we normally try to avoid it.

(iii) An **innermost lambda abstraction** is a lambda abstraction which contains no other proper lambda abstractions. The lambda abstraction (lam s . ((access i) s) mal) in expression (b) in section 1.1. is an example.

(iv) A **redex** is an expression whose structure matches the left hand side of a rewriting rule [5].

(iv) A **normal form** is an expression which contains no redexes [5].

2.2.2. The rewriting rule schemes.

(i) **Eta rule.** The Eta rule eliminates redundant lambda abstraction.

Definition: (lam x.E x) => E if x is not free in E.

(ii) **Alpha rule.** The alpha rule is a name changing rule. It enables us to change the name of the formal parameter of a lambda abstraction, as long as it is done consistently. Let E[e/x] represent the expression constructed by substituting all free occurrences of x in E by e.

Definition: (lam x.E) => (lam y.E[y/x]) if y is not free in E.

(iii) **Beta rule.** The beta rule enables us to apply a lambda abstraction to an argument by making a new instance of the body of the abstraction and substituting the argument for free occurrences of the formal parameter.

Definition: (lam x.E) e => E[e/x].

(iv) **Delta rule.** The delta rule is a form of rewrite rule for built-in functions. The functionality of this rule is very similar to that of the beta rule.

Definition: f e1 ... en => [en/xn] ... [e1/x1] E

where f is defined as f x1 ... xn => E.

From the implementation point of view, the 'execution' of a lambda calculus expression is by rewriting. A reduction proceeds by repeatedly selecting a **redex** and rewriting it [4]. Following the convention, we use the symbol '=>' to denote that one-step reduction has been performed. In the expression (a) above, there is one redex, namely ((plus one) two) as it matches the left hand side of the delta-rule plus $a\ b$ => $a+b$ If the delta rule is applied, the expression is reduced to

=> ((times three) four).

Notice that the action created a new redex which can further be reduced by the delta-rule times $a\ b$ => $a*b$ to a normal form

=> twelve.

2.3. Rewriting Strategies.

There are different strategies for rewriting an expression. Two of the strategies we consider here are **call-by-value** and **call-by-name.** The rewriting strategy that we use is call-by-value.

2.3.1. Call-by-value.

In call-by-value, arguments to the beta and alpha rules are evaluated at the point of call. For this reason, it is sometimes called an eager rewriting strategy. The evaluated arguments are used to initialize the formal parameters of the rules. Since the evaluated arguments are usually smaller, using this rewriting strategy can minimize run-time memory usage. For example:

```
    (lam x. plus x x) (plus 3 4)
=>  (lam x. plus x x) 7
=>  plus 7 7
=>  14.
```

2.3.2. Call-by-name.

In call-by-name, arguments to the beta and alpha rules are not evaluated at the point of call. Consequently, it is sometimes called a lazy rewriting strategy. Each occurrence of the formal parameter is replaced textually by the unevaluated actual parameter. Since some arguments are not used in the body of an abstraction, using this rewriting strategy can save effort of evaluating unused arguments. For example:

```
   (lam x. if true (time 2 3) x) (fac 100)
=> if true (times 2 3) (fac 100)
=> times 2 3
=> 6.
```

We have just finished a tutorial session about the typed lambda calculus and the rewiring rule schemes. Our next step is to denote the meaning of a typical imperative language using the typed lambda calculus. This also leads us to the discussion of the partial evaluation techniques.

Chapter 3

Single-Threading, Control Binding and Lambda-Lifting

Since the concepts of Single-Threading, Control Binding and Lambda-Lifting form the main ingredients for our research, we devote this chapter to do a cursory inspection of these topics.

3.1. Single-Threading.

As presented in detail in Schmidt [5,6,8], single-threading is the sequential processing property of a programming language's semantic definition. A semantic definition is said to be **single-threaded** if its store argument can be replaced by access rights to a single global variable while preserving the operational properties of the semantic definition. We believe that by exploiting this property in the definition, better and more efficient implementation can be generated from it. Statically checkable, syntactic criteria [8] for verifying that an expression is single-threaded in its use of a store argument are summarized in the following paragraphs.

Definition.

An expression is:

(i) **trivial** if it is an identifier.

(ii) **active** if it is not properly contained in an abstraction.

The Syntactic Criteria.

In this section, the letter *S* denotes a store-typed domain while the letters *D, D1 and D2* denote any domains, for example, *store, nat, bool* and *etc.* We write *e:D* to state that expression *e* belongs to domain *D*.

An expression *E* is single-threaded in its domain *S* if:

(i) *E* is *i:D* or *c:D*.

(ii) *E* is *(lam i:D1. E1):D1->D2, E1* is single-threaded, and

if *D1 = S*, then all active S-typed identifiers in *E1* are *i:S*;

if *D1 <> S*, then *E1* has no active S-typed expressions.

(iii) *E* is *(E1 E2):D2*, *E1* and *E2* are single-threaded, and

if *D2 = S*, then if both *E1* and *E2* contain one or more active S-typed expressions, then all of the active S-typed expressions in *E* must be occurrences of the same identifier *i:S*;

if *D2 <> S*, then all occurrences of active S-typed expressions in *E* must be occurrences of the same identifier *i:S*.

In order to understand the above criteria better, it is best to study some examples.

(a) (lam s1. lam s2. s1)

This expression fails to satisfy condition (ii). The problem arises when an expression outside the lambda abstraction lam s2. s1 updates the s-typed identifier 's1'. The s-typed identifier 's1' in the abstraction will not be able to see the change and thus generates unexpected results when it is used.

(b) (lam i. access i s0)

This expression fails to satisfy condition (ii). The reason is similar to the one

mentioned in (a).

(c) (update [[A]] (access [[A]] s0) (update [[A]] zero s0))

This expression fails to satisfy condition (iii). The active s-typed subexpressions, namely s0 and (update [[A]] zero s0) clash if the expression is evaluated from right to left. After the subexpression (update [[A]] zero s0) is evaluated, a new s-typed value is created, say s1. The presence of 's0' in (access [[A]] s0) violates the sequential processing property of the expression.

(d) access [[A]] (update [[A]] zero s0)

This expression does not satisfy condition (iii) although the expression standalone is single-threaded. This is because the expression can appear within a larger expression and cause a problem, for example, the expression (update [[A]] (access [[A]] (update [[A]] one s0)) s0). The subexpression (update [[A]] one s0) creates a local s-typed value which will disappear right after the operator 'access' has used it.

(e) lam i . lam n . lam s . (((update i) n) s) mal mal mal

This expression is single-threaded because it satisfies all three conditions.

(f) (update [[B]] two (update [[A]] one s0))

This expression is single-threaded because it satisfies all three conditions.

Single-Threaded Language Definition. The abstract syntax of a simple while-loop language is given in FIGURE 3-1. To study the meaning of the while-loop language, we map its syntactic structures to its mathematical entities through a **denotational semantics** for the language. These entities are defined by the **semantic algebras** shown in FIGURE 3-2. In FIGURE 3-3, the semantic algebras are used to give meaning to the syntax via **valuation functions**.

It is important to be able to recognize that the definition of FIGURE 3-3 is indeed a definition of a sequential, imperative language because the semantic store argument is treated in a sequential fashion when passed as a parameter. That is, any program is translated by the definition into a single-threaded lambda expression. To make the point clearer, let us study the result of translating program P[[A:=0; B:=A+1.]] using the definition in FIGURE 3-3.

P[[A:=0; B:=A+1.]]
= C[[A:=0; B:=A+1]]
= lam s. C[[B:=A+1]] (C[[A:=0]]s)
= lam s. C[[B:=A+1]] (lam s. update [[A]] (lam s. zero)s s) s
= lam s. (lam s. update [[B]] (lam s. access [[A]] s) s plus (lam s. one) s s) (lam s .update [[A]] (lam s. zero)s s) s

The resultant expression is single-threaded because it satisfies the criteria above. This suggests that the individual instances of the store argument can be replaced by access rights to a single global variable. A semantic definition whose store argument can be replaced by access rights to a single global variable while preserving operational properties is said to be single-threaded (in its store). The criteria defined by Schmidt [8] are sufficient conditions for the single-threading property to hold for a denotation of a program.

After we have detected that a semantic definition is single-threaded in its store argument, we can transform the semantic definition into one which uses a global store variable. The technique defined in [7] goes as follows:

(i) For the Store algebra, replace domain *s:Store=D* by the variable declaration var *s:Store=D* and transform:

destruction operations *c:A1*,...,*An*Store->E, E<>Store*. defined as *(c a1,...,an,s)=e* to *c:A1*,...,*An*Unit->E*, defined as *(c a1.....an,())=e1. Any occurrences of s* in *e* are replaced by *()*.

```
Abstract syntax:
  P: Program
  C: Command
  E: Expression
  B: Boolean-expr
  I: Identifier
  N: Numeral

  P::= C.
  C::= C1; C2| I:=E| if B then C1 else C2| while B do C
  E::= E1+E2| I| N
```

FIGURE 3-1

```
Semantic algebras:
  I.  Truth values
      Domain t: Tr = B
      Operations
       true: Tr
       false: Tr
       not: Tr -> Tr

  II. Natural numbers
      Domain n: Nat = N
      Operations
       zero, one, ...:Nat
       plus: Nat * Nat -> Nat
       equals: Nat * Nat -> Tr

  III. Store
       Domain s: Store = Identifier -> Nat
       Operations
        newstore: Store
         newstore = lam i. zero
        access: Identifier-> Store -> Nat
         access i s = s(i)
        update: Identifier -> Nat -> Store -> Store
         update i n s = lam j. j equalid i -> n [] s(j)
```

FIGURE 3-2

```
P: Program -> Store -> Store
   P[[C.]] = C[[C]]

C: Command -> Store -> Store
   C[[C1; C2]] = lam s. C[[C2]](C[[C1]]s)
   C[[I:=E]] = lam s. update [[I]] (E[[E]]s) s
   C[[if B then C1 else C2]] = lam s. B[[B]]s ->
                                  C[[C1]]s [] C[[C2]]s
   C[[while B do C]] wh
      where wh = lam s. B[[B]]s -> wh(C[[C]]s) [] s

E: Expression -> Store -> Nat
   E[[E1+E2]] = lam s. E[[E1]]s plus E[[E2]]s
   E[[I]] = lam s. access [[I]] s
   E[[N]] = lam s. N[[N]]

B: Boolean-expr -> Store -> Tr  (omitted)

N: Numeral -> Nat (omitted)
```

FIGURE 3-3

construction operations *c:A1*,...*An*Store->Store*, defined as *(c a1,...,an,s)=e* to *c:A1*,...,*An*Unit->Unit*, defined as *(c a1,...,an,())=(s:=e)*. The result is the value *()*.

(ii) Replace all occurrences of Store-typed identifiers *s* that appear in the semantic equations and operations by *()*.

The transformed language of FIGURE 3-3 is represented in FIGURE 3-4.

The store argument is no longer copied into an expression during reductions. Instead, ()-values are used. We assume that the ()-values are the **control markers**, that is, they give permission to subexpressions to evaluate.

3.2. Control Binding.

If we translate the program P[[A:=0; B:=A+1.]] using the definition in FIGURE 3-4, we get:

```
Store module
 var s: Store = New + Upd
       where New = Upd = { () }
    Operations

    newstore: Unit
     newstore = (s:= inNew())

    access: Identifier*Unit -> Nat
     (access i ()) = eval i s

    update: Identifier*Nat*Unit -> Unit
     (update i n ()) = (s:=inUpd(i, n, s))

Valuation functions:

P: Program -> Unit -> Unit
  P[[C.]]= C[[C]]

C: Command -> Unit -> Unit
  C[[C1; C2]] = lam ().C[[C2]](C[[C1]]())
  C[[if B then C1 else C2]] =
        lam (). B[[B]]() -> C[[C1]]() [] C[[C2]]()
  C[[while B do C]] = lam (). wh()
    where wh=lam (). B[[B]]() -> wh(C[[C]]()) [] ()
  C[[I:=E]] = lam (). update [[I]] (E[[E]]()) ()

E: Expression -> Unit -> Nat
  E[[E1+E2]] = lam (). E[[E1]]() plus E[[E2]]()
  E[[I]] = lam ().access [[I]] ()
  E[[N]] = lam (). N[[N]]
```

FIGURE 3-4

= lam (). (lam (). update [[B]] (lam (). access [[A]] ()) () plus (lam (). one) () ())
(lam () .update [[A]] (lam (). zero)() ()) ()

Notice that the definition in FIGURE 3-4 produces program denotations that contain a large number of combinations of the form (lam().M)() (expressions that manipulate the global variable). These combinations can be optimized out of the denotation before run-time. That is, we want to remove occurrences of lam () and () from the definition. The program will be translated to lambda expression without all the (lam

(). E)() forms. The technique of **Control Binding** defined in [7] is used to serve this purpose. The technique used on a language definition goes as follows:

For a valuation function *A* such that each equation for *A* has the form *A[[Ai]] = lam ().Ei*, replace all occurrences of

lam ().Ei with *Ei*.

A[[A]]() in *Ei* with *A[[A]]*.

A[[A]] not in combination with *()* by *(lam ().A[[A]])*.

FIGURE 3-5 gives the definition of FIGURE 3-4 after control binding. (Note: Control binding is also performed on the Store algebra.) Notice that almost all of the lam () and () values have disappeared.

```
P[[C.]] = C[[C]]

C[[C1; C2]] = C[[C1]];C[[C2]]
C[[I:=E]] = update [[I]] E[[E]]
C[[if B then C1 else C2]] = B[[B]] -> C[[C1]] [] C[[C2]]
C[[while B do C]] = wh
  where wh = B[[B]] -> C[[C]];wh [] ()
C[[skip]] = ()

E[[E1+E2]] = E[[E1]] plus E[[E2]]
E[[I]] = access [[I]]
E[[N]] = N[[N]]

( the expression E1; E2 abbreviates (lam ().E2)E1 )
```

FIGURE 3-5

The resultant language definition in FIGURE 3-5 is very useful because it can be used to derive a code generator. As one will see, the task can be easily accomplished. Figure 3-6 illustrates how we can apply the semantic notation in Figure 3-5 to translate a program to its denotation.

```
  P[[A:=0; B:=A+1.]]
= C[[A:=0; B:=A+1]]
= C[[A:=0]]; C[[B:=A+1]]
= update [[A]] E[[0]]; C[[B:=A+1]]
= update [[A]] N[[0]]; update [[B]] E[[A]] plus N[[1]]
= update [[A]] zero; update [[B]] access [[A] plus one
```

FIGURE 3-6

The result in Figure 3-6 is almost machine code. Without much effort, we can transform the resultant expression in Figure 3-6 into its postfix form and obtain:

```
zero
[[A]]
update
[[A]]
access
one
plus
[[B]]
update
```

Notice that there is a striking resemblance between the postfix expression defined above and the hypothetical stack machine code given below:

```
pushconst zero
pushid [[A]]
do update
pushid [[A]]
do access
pushconst one
do plus
pushid [[B]]
do update
```

Notice that the stack code does not carry any store arguments but lets the "do access" and "do update" manipulate the store instead. (The store is a fixed machine component.) In reality, this is exactly what a conventional stack code which runs on a Von Neumann architecture would do.

As mentioned above, the conversion of a lambda calculus expression from one form into another is the fundamental operation of our implementation. Obviously, the efficiency of this implementation cannot be ignored. In fact, this important aspect has been taken very seriously and a technique called lambda-lifting has been designed to serve the purpose.

3.3. Lambda-Lifting.

Lambda-lifting transforms a program into an equivalent form that uses supercombinators [6].

A supercombinator is a closed lambda-abstraction such that all lambda-abstractions within it are also closed. For example,

```
(lam x : nat. plus ((lam y : nat. y) 1) x)
```

is a supercombinator, but

```
(lam x : nat. plus y x)
```

is not because y is free in the expression. In the implementation, handling free variables is a nuisance because a symbol table must be maintained to remember the values of the free variables. Furthermore, each such expression must have its own symbol table. The process of transforming the supercombinators into names and easy-to-implement rewrite rules to supercombinators are called **lambda lifting.** For example, the supercombinator above can be named $0, and the rules:

```
$1 y => y
$0 x => plus ($1 1) x
```

are generated.

The algorithm from [4] which does the conversion is summarized below:

While there are more lambda abstractions do

BEGIN

1) *Choose any lambda abstraction which has no inner lambda abstractions in its body.*

2) *Take out all its free variables as extra parameters.*

3) *Give an arbitrary name to the lambda abstraction. Following the convention, we use $0, $1, $2 and so on as names for supercombinators.*

4) *Replace the occurrence of the lambda abstraction by name applied to the free variables.*

5) *Compile the lambda abstraction and associate the name with the compiled code.*

END

Using this algorithm (Chapter 4 puts this algorithm to work), we can convert the semantic equations for C and E in Figure 3-3 so that the right hand side of the equations consist only of supercombinators and their arguments. See figure 3-7. The rewriting rules for the supercombinators is given in Figure 3-8.

```
P[[C.]] = C[[C]]

C[[C1; C2]] = $0 C[[C1]] C[[C2]]
C[[I:=E]] = $1 [[I]] E[[E]]
C[[if B then C1 else C2]] = $2 B[[B]] C[[C1]] C[[C2]]
C[[while B do C]] = fix ($3 B[[B]]  C[[C]])
   where fix g = g (fix g)

E[[E1+E2]] = $4 E[[E1]] E[[E2]]
E[[E1-E2]] = $5 E[[E1]] E[[E2]]
E[[I]] = $6 [[I]]
E[[N]] = $7 N[[N]]
```

FIGURE 3-7

```
$1 i e s => $upd i (e s) s
$6 i s => (s i)
$upd i n s j => j equalid i -> n [] s(j)
$0 c1 c2 s => c2 (c1 s)
$2 b c1 c2 s => (b s) -> (c1 s) [] (c2 s)
$3 b c f s => (b s) -> f (c s) [] s
$4 e1 e2 s => (e1 s) plus (e2 s)
$5 e1 e2 s => (e1 s) minus (e2 s)
$7 n s = n
```

FIGURE 3-8

In FIGURE 3-8, the frequent occurrences of the store argument (s) make the implementation of the lambda-lifted definition more inefficient than we like. For example, if the program C[[A:=0; B:=A+1]] is translated using the definition in FIGURE 3-7 and FIGURE 3-8, the resultant expression looks as follow:

```
    P[[A:=0; B:=A+1.]]
=   C[[A:=0; B:=A+1]]
=   $0 C[[A:=0]] C[[B:=A+1]]
=   $0 $1 [[A]] E[[0]] $1 [[B]] $4 E[[A]] E[[1]]
=   $0 $1 [[A]] $7 N[[0]] $1 [[B]] $4 $6 [[A]] $7 N[[1]]
=   ($1 [[B]] $4 $6 [[A]] $7 N[[1]]) ($1 [[A]] $7 N[[0]] s)
=   ($upd [[B]] (($6 [[A]] s) plus ($7 N[[1]] s) s) ($upd [[A]] ($7 N[[0]] s) s)
=   ($upd [[B]] (($6 [[A]] s) plus one) s) ($upd [[A]] zero s)
```

We have seen three optimization techniques, each does something different. We want to put them together to see if we can get all their advantages in one system. But the order in which they should be applied is not known. Consequently, our goal in the Chapters that follow is to conquer this unknown.

Chapter 4

Implementation of Compiler Generator System

Phase I

The compiler generator system is implemented entirely in **Standard ML**, a language developed at University of Edinburgh [3]. Standard ML is a *functional* and *interactive* programming language. We used a version that runs on a VAX 11/780 operating under Berkeley 4.3 UNIX and on a SUN 3/60 operating under SunOS 4.0.

This and the next chapter describe how the techniques described in Chapter 3 are implemented. The order in which they are presented will correspond to the one presented in Chapter 3. We also point out ways to handle problems that arouse during the work of automating these techniques. The discussion will not go into details about the source code.

4.1. Organization of Compiler Generator System.

Conceptually, the compiler generator system operates in phases. Each phase transforms a source language definition from one form into another. The initial design of the compiler generator system is sketched in FIGURE 4-1. Since the ordering of single-threading, control binding and lambda-lifting has not been decided, double-headed arrows are used. The order which we use for our discussion here is applying single-threading first, control binding second and lambda-lifting last.

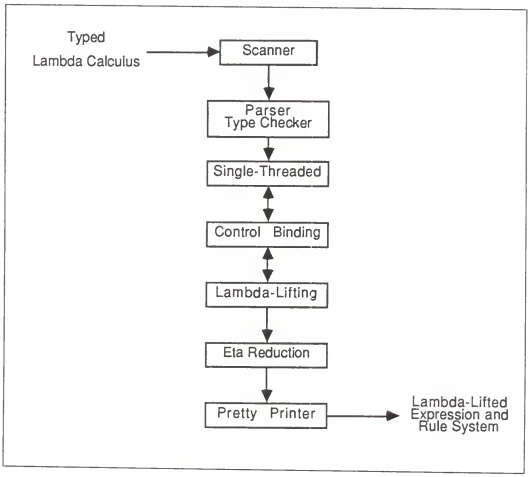


FIGURE 4-1

Notice that the input to this system is not a language definition as one might expect. Instead, a typed lambda calculus expression is used. We do this for couple of reasons.

First, the system is easier to build and understand if it is to process a single typed lambda calculus expression. Second, the existing system will be extended in Chapter 5 to process a set-of-equations semantic definition. For these reasons, we choose to use the expression

lam i : iden . lam s : store . ((access i) s) mal mal

as our example input throughout this chapter.

4.2. Scanner, Parser and Type-Checker.

These three modules perform a service found in most all of compilers: they break up the input into its constituent pieces and create a derivation tree from them. The derivation tree has the typing information attached to each of its nodes. This version of the derivation tree is sufficiently informative for the implementation of the rest of the system.

4.2.1. Scanner.

The scanner module is the simplest one. It processes a string of characters one at a time and produces a list of strings as its output. An example should make the point clear. If the input to this module is a string that looks as follows:

"lam i : iden . lam s : store . ((access i) s) mal mal "

then the output would look like:

["lam","i",":","iden",".","lam","s",":","store",".","(","(","access",
"i",")","s",")","mal",mal"].

Although it seems hard to read, this list of strings is useful input for the parser.

4.2.2. Parser and Type Checker.

These two modules are implemented as one because it improves efficiency. While the parser is building the tree, the type checker performs its task at the same time.

The parser is coded from the concrete syntax for the typed lambda calculus. The parser reads a list of strings from the scanner and determines whether the input program the list represents is syntactically well-formed. While the parser is doing its job, the tree the parser builds is being type checked. Any ill-formed syntax or typing will be reported. (No error recovery is included.) The parse tree output corresponding to the input list seen above is depicted in FIGURE 4-2.

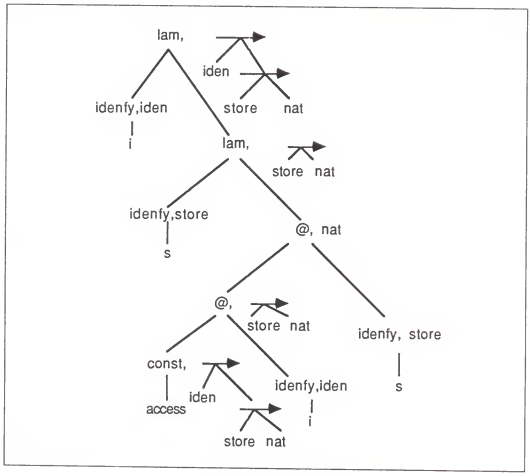


FIGURE 4-2

The parse tree may be viewed as a graphical representation of the derivation [1]. Each interior node of the parse tree is labeled either by a 'lam' or '@' and each leaf of the parse tree is labeled either by an 'idenfy' or 'const'. Following the convention, the interior nodes are sometimes called nonterminals while the leaves are sometimes called terminals. Unlike the regular terminals and nonterminals, they are tagged with typing information. The typing information are represented by trees as well. All constants or operators must have their types defined in a predefined environment if they are not explicitly defined in the input expression. Hence, the type for 'access' will be retrieved from the predefined environment when this module is involked.

4.3. Single-Threading.

We systematically automated the single-threading criteria and global variable transformation technique presented in the previous chapter. The first stage verifies that the lambda calculus expression is single-threaded and the second stage transforms the single-threaded expression. The implementation of the first stage progresses in a bottom-up fashion, that is, the leaves of the tree will first be verified before their roots. For example, the identifiers 's' and 'i', and the operator 'access' will be the first to be verified. If there are no offending leaves, their roots will be verified next. This similar process continues until the root of the whole tree is encountered and verified. The tree in FIGURE 4-2 is indeed single-threaded in its 'store' argument. The tree is unaltered after the first stage is completed. Following the transformation technique in section 3.1, stage two transforms the single-threaded tree in one traversal. FIGURE 4-3 shows the result.

4.4. Control Binding.

The technique of control binding we described earlier handles a-set-of equations

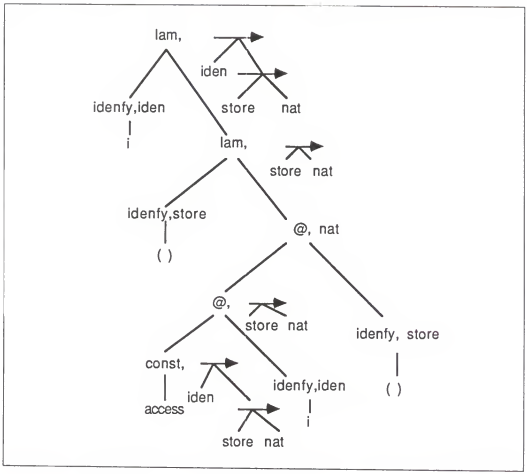


FIGURE 4-3

semantic definition. To do control binding on a single lambda calculus expression, some adjustments must be made. (Note: In Chapter 5, control binding on a language definition will be presented.) The newly adjusted technique is as follow,

Step 1. Rewrite all occurrences of *(lam (). E) ()* to *E*.

Step 2. If all uses of operator *c* in *E* has the form *(c E1,...En ())*, rewrite each use of *c* to *(c E1,...,En)*.

FIGURE 4-4 shows the resultant tree after the technique is enforced on the tree in FIGURE 4-3. Notice that the corresponding type tags are altered accordingly. (Although it seems wasteful to do that since the typing information is no longer

needed, it maybe useful in the future expansion of this project.)

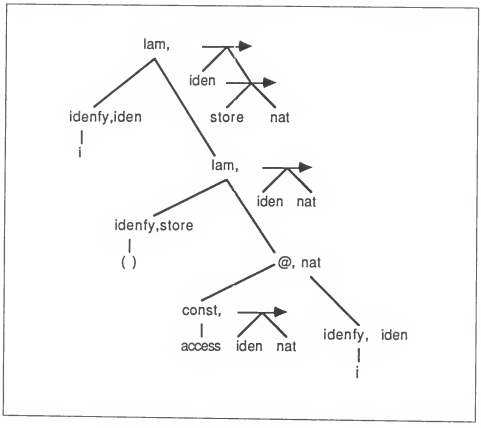


FIGURE 4-4

4.5. Lambda-Lifting.

Up till now, the tree still has the 'lam' operators it started with. As we pointed out earlier, the presence of the 'lam' is undesirable and we must make them disappear. The presentation that follows describes the solution, lambda-lifting, one linearlized trees. In section 4.7, we talk about the module which does linearization.

After the tree in FIGURE 4-2 is linearized, it looks as follows:

lam i. lam s. ((access i)s)

Although the type tags are not shown, they are still properly maintained in the implementation. Let us now put the lambda-lifting algorithm to work in a stepwise

fashion. For easy reference, all the steps are numbered so that they correspond to the ones in the algorithm.

After the tree in FIGURE 4-4 is fed through the linearlization module, it enters the first iteration of the lambda-lifting loop. The following steps occur:

Step 1) choose lam s. ((access i)s)

Step 2) construct (lam i. lam s. ((access i)s))i

Step 3) let $0 represent lam i. lam s. ((access i)s)

Step 4) construct lam i. ($0 i)

Step 5) define (($0 i) s) = ((access i)s)

After the first iteration of the loop is completed, we have:

(($0 i) s) = ((access i)s)

lam i. ($0 i)

As there remains one more lambda abstraction, the second iteration the following:

Step 1) choose lam i. ($0 i)

Step 2) construct lam i .($0 i) (no change)

Step 3) let $1 represents lam i. ($0 i)

Step 4) construct $1

Step 5) define ($1 i) = ($0 i)

The lifting terminates and we end up with the following expression and rule system:

(($0 i) s) = ((access i)s)

($1 i) = ($0 i)

$1

Clearly, the rule ($1 i) = ($0 i) is redundant. We can apply the eta-rule defined section 2.2 to simplify it to

$1 = $0

Having done so, $1 itself is redundant, and $1 can be replaced wherever it occurs by $0, giving:

(($0 i) s) = ((access i)s)

$0.

We name (($0 i) s) = ((access i)s) as a rewriting rule and the standalone $0 as an expression to be evaluated. As we will discover later, it is a little too general to call this rule a rewriting rule. A more specific term is essential to distinguish it from yet another set of rewriting rules. For this reason, we call this rule a **compile-time rule.** On the other hand, the rule for the 'access' is called a **run-time rule.** A more complete coverage of these rules can be found in Chapter 5.

4.6. Eta-Reduction.

Through the example given the previous section, we see the need to incorporate eta-reduction in our compiler generator system. Eta-reduction was implemented as an independent module. The input to this module is a list of rewriting rules. For instance, if the list

[(($0 i) s),((access i)s)), (($1 i),($0 i))]

is fed through this module, the output would look as follows:

[(($0 i) s),((access i)s))].

The redundant rewriting rule which does nothing has been optimized out of the list by eta-reduction.

4.7. Pretty Printer

The pretty printer analyses the rewriting rules and prints them in such a way that the structure of the rules become clearly visible. It is more of a debugging tool rather

than a necessary component to the system. The module processes the tree in FIGURE 4-2 and produces a linearized tree similar to the one presented in section 4.6. Without this module, it will be hard to make the presentation in section 4.6. as clear as is.

The discussion in the next chapter covers the process of augmenting the existing system to one which processes a set-of-equations semantic definition. The order in which the modules are presented is similar to the one you see in this chapter.

Chapter 5

Implementation of Compiler Generator System.

Phase II

The system as described so far processes only a single typed lambda calculus expression. In this chapter, attention will be focused on augmenting the existing modules so that, together, they can process an arbitrary set-of-equations semantic definition. Since an equation lhs = rhs can be viewed as a pair of expression, set-of-equations semantic definition is just a list of pairs of lambda calculus expressions. One can subsequently select a pair and break it into its two constituent parts (expressions) before they are processed. As a result, the existing system processes equations as pairs of lambda calculus expressions. Although there are some modification necessary, they are minor.

5.1. Augmented System.

The design of the augmented system in FIGURE 5-1 looks similar to the one presented in FIGURE 4-1. The key difference is this system takes the run-time and compile-time rules as its input rather than a single typed lambda calculus expression. Since the rules play a key role in helping us understanding the system, the discussion in section 5.2 is devoted to them.

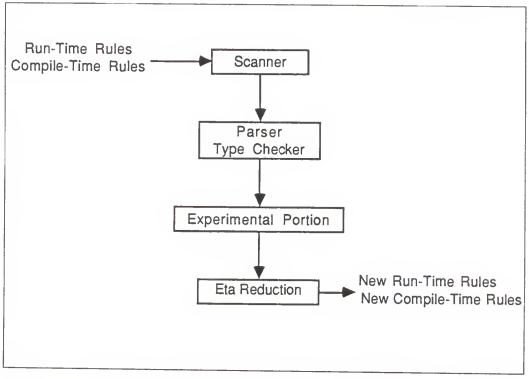


FIGURE 5-1

5.2. Run-Time Rules and Compile-Time Rules.

The run-time and compile-time rules are indistinguishable. However, their underlying operational behaviors are quite distinct. In the denotational semantics framework, the terms "semantic algebra" and "valuation functions" are used for the run-time and compile-time rules respectively. As one might expect, these rules will look very similar to the ones defined in FIGURE 4-2 and FIGURE 4-3, only this time they are defined in an easily implementable form. The corresponding rules are defined in FIGURE 5-2 and FIGURE 5-3 respectively.

The basic idea of our implementation is to subsequently extract and process each individual rule from a set of rules. Inside each module, these rules are subsequently broken down into a left-hand and right-hand expressions. These expressions are then processed in turn. Consequently, the modules are unware of the fact that they are

```
Run-Time Rules:
  ((plus m) n) = m+n
  ((times m) n) = m*n
  (pred n) = n-1
  (eq0 n) = n=0
  (((if true) f) g) = f
  (((if false) f) g) = g
  empty = s:=]am i . zero
  ((access i) s) = s(i)
  (((update i) n) s) = s:=[i|->n]s
```

FIGURE 5-2

```
Compile-Time Rules:
  ($C ((; c1) c2)) = lam s . (($C c2) (($C c1) s))
  ($C ((:= i) e)) = lam s . (((update i) (($E e) s)) s)
  ($C ((+ e1) e2)) = lam s . ((plus (($E e1) s))
                              (($E e2) s))
  ($E (# n)) = lam s . ($N n)
  ($E (@ i)) = lam s . ((access i) s)
  ($N 0) = zero
  ($N 1) = one
  ($N 2) = two
  ($N 3) = three
  ($N 4) = four
  ($N 5) = five
```

FIGURE 5-3

processing a set of rules.

5.3. Scanner, Parser and Type Checker.

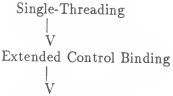
Although the ideas presented in section 4.2 can still be applied, some minor modifications are required. It is best explained by an example. Consider the following rule:

```
($E (@ i)) mal = lam s : store . ((access i ) s) mal.
```

The operators '$E', '@' and 'access' are treated as built-in functions. Their corresponding types are *(expr -> store -> nat), (iden -> expr)* and *(iden -> store -> nat)*. A predefined environment is used to record the definitions of these built-in functions. However, the presence of the undefined identifier 'i' in the right-hand side expression causes trouble. In order to successfully process the entire rule, the right-hand expression must be informed of the type of the identifier 'i'. For this reason, we installed a temporary environment in the type checker. The environment serves as a communication channel between the left-hand and right-hand expressions. In this case, the information it sends is the type for the identifier 'i'. After the entire rule has been processed, the extra parameter 'i' is removed because it is no longer needed. Through this module and the pretty printer module, the linearized trees shown in FIGURE 5-2 and FIGURE 5-3 are generated.

5.4. Experimental Portion.

Recall that the goal of our research is to study the interaction of single-threading, lambda-lifting and control binding. Since we have three techniques to consider, we have six models to study, each of which consists of a unique combination of the three techniques. Since all models are executed in a similar manner, our plan is to study one of them here in detail. In Chapter 6, we give a more comprehensive examination of the results from the experiment. The model we present here is:



5.4.1. Single-Threading.

We begin with detecting single-threadingness on the rules on FIGURE 5-2 and FIGURE 5-3. Not surprisingly, the single-threading module defined in section 4.3 can be applied directly without any alteration. The detections of single-threadedness of the left-hand and right-hand expressions can be performed independently. Any expression that fails to satisfy the single-threading criteria will be reported to the user. If the rules satisfy the conditions for single-threading, they are transformed to ones which use the control markers. FIGURE 5-4 and FIGURE 5-5 show the resultant rules. Notice that all occurrences of the single-threaded store arguments are replaced by the ()-values.

```
Run-Time Rules:
 ((plus m) n) = m+n
 ((times m) n) = m*n
 (pred n) = n-1
 (eq0 n) = n=0
 (((if true) f) g) = f
 (((if false) f) g) = g
 empty = s:=lam i . zero
 ((access i) ()) = s(i)
 (((update i) n) ()) = s:=[i|->n]s
```

FIGURE 5-4

5.4.2. Extended Control Binding.

The single-threaded rules contain a large number of control markers. Naturally, our goal in this section is to optimize them out of the rules. Based on our current implementation of control binding, applying the technique defined in section 3.2 directly means a large portion of the code would have to be modified. In order to minimize the changes required on the module, we simply extend the technique.

```
Compile-Time Rules:
  ($C ((; c1) c2)) = lam () . (($C c2) (($C c1) ()))
  ($C ((:= i) e)) = lam () . (((update i) (($E e) ())) ())
  ($C ((+ e1) e2)) = lam () . ((plus (($E e1) ()))
                               (($E e2) ()))
  ($E (# n)) = lam () . ($N n)
  ($E (@ i)) = lam () . ((access i) ())
  ($N 0) = zero
  ($N 1) = one
  ($N 2) = two
  ($N 3) = three
  ($N 4) = four
  ($N 5) = five
```

FIGURE 5-5

The extended control binding technique consist of two parts. The first part is intended to rearrange the rules. They are rearranged so that the second part can used them to exploit the maximum power of the new control binding technique. The extended technique is summarized below:

Part I.

Every operator *op* whose rules have

op a1...an => lam x1...lam xn. lam (). E

transform the rules to

op a1...an, x1...xn () => E

Part II

For all rules of the form

op a1...an, x1...xn () => E

from **Part I**,

1) transform the rules to

op a1...an, x1...xn => E

2) for all occurrences of *op* in *E* do:

   a) if the occurrence is applied to an argument *()*, eliminate *()*.

   b) if the occurrence is not applied to an argument *()*, enclose it by the new combinator *%1*, whose rewriting rule is:

%1 f () => f

If the rules in FIGURE 5-4 and FIGURE 5-5 are run through **Part I** of the module, they are transformed into ones that look like in FIGURE 5-6 and FIGURE 5-7. Note that all outermost lambda abstractions have disappeared.

```
Run-Time Rules:
  ((plus m) n) = m+n
  ((times m) n) = m*n
  (pred n) = n-1
  (eq0 n) = n=0
  (((if true) f) g) = f
  (((if false) f) g) = g
  empty = s:=lam i . zero
  ((access i) ()) = s(i)
  (((update i) n) ()) = s:=[i|->n]s
```

FIGURE 5-6

```
Compile-Time Rules:
  (($C ((; c1) c2) ()) = (($C c2) (($C c1) ()))
  (($C ((:= i) e) ()) = (((update i) (($E e) ())) ())
  (($C ((+ e1) e2) ()) = ((plus (($E e1) ())) (($E e2) ()))
  (($E (# n) ()) = ($N n)
  (($E (@ i) ()) = ((access i) ())
  ($N 0) = zero
  ($N 1) = one
  ($N 2) = two
  ($N 3) = three
  ($N 4) = four
  ($N 5) = five
```

FIGURE 5-7

Part II of the module is used to eliminate the frequent occurrences of the ()-value. The results are shown in FIGURE 5-8 and FIGURE 5-9.

Control binding on the expression

```
(($C c2) (($C c1) ()))
```

```
Run-Time Rules:
  ((plus m) n) = m+n
  ((times m) n) = m*n
  (pred n) = n-1
  (eq0 n) = n=0
  (((if true) f) g) = f
  (((if false) f) g) = g
  empty = s:=lam i . zero
  (access i) = s(i)
  ((update i) n) = s:=[i|->n]s
  ((%1 c ) ()) => c
```

FIGURE 5-8

```
Compile-Time Rules:
  ($C ((; c1) c2)) = ((%1 ($C c2)) ($C c1))
  ($C ((:= i) e)) = ((update i) ($E e))
  ($C ((+ e1) e2)) = ((plus ($E e1)) ($E e2))
  ($E (# n)) = ($N n)
  ($E (@ i)) = (access i)
  ($N 0) = zero
  ($N 1) = one
  ($N 2) = two
  ($N 3) = three
  ($N 4) = four
  ($N 5) = five
```

FIGURE 5-9

is of special importance. Since the argument (($C c1) ()) to the leftmost $C is not a ()-value, we are forced to use rule 3 in part II. As a result, the $C is enclosed with a new run-time combinator. The rule for the new combinator is grouped together with the run-time rules. Note that not all ()-values have disappeared. The ()-value in ((%1 c) ()) => c is needed as it gives permission for c to gain control of the global store variable. (We will have more to say about this in section 6.2.)

5.4.3. Lambda-Lifting.

Doing lambda-lifting on sets of rules is simple. Like the single-threading module, the lambda-lifting module can be applied directly without any changes. Lambda-lifting the left-hand and right-hand expressions are independent processes. Recall that doing lambda-lifting eliminates lambda abstraction and generates new rules. The rule systems in FIGURE 5-8 and FIGURE 5-9 do not contain any lambda abstraction. No new rules are created.

5.5. Eta-Reduction.

For the same reason we saw in Chapter 3, the eta reduction module is also installed in the augmented system. After the reduction is performed on the rules in FIGURE 5-8 and FIGURE 5-9, they remain unaltered because no redundant rules were found.

We have just completed a tour of the various phases of the compiler generator system. The system is capable of doing partial evaluation on a set-of-equations semantic definition. The resultant rule systems are smaller and run more efficiently. Nevertheless, to complete the process of generating a compiler, a compile-time evaluator must be built. The evaluator is generic because it is capable of evaluating any given set of rules written in the typed lambda calculus. One can execute the output from the evaluator in various ways. Two of the possible ways are discussed in Chapter 7.

Chapter 6

Results

As mentioned in the previous chapter, there are six ways to order the single-threading, control binding and lambda-lifting modules. In order to justify which ordering works the best, each of the orderings were tested with the same set of test data. The output from these tests are then compared. The best ordering will be the one which yields the smallest result. In the previous chapter, we studied how one of these tests was conducted. The remaining five tests can be carried out in the similar manner. The results are posted in sections 6.1 through 6.5. Section 6.6 gives a summary.

6.1. Single-Threading, Lambda-Lifting and Control Binding.

By comparison, the results in FIGURE 6-1 and 6-2 seem much larger than the ones in FIGURE 5-8 and 5-9. This is simply because some of the rules in FIGURE 6-2 possess the ()-value. Moreover, new rules were created in FIGURE 6-2 through the process.

6.2. Control Binding, Single-Threading and Lambda-Lifting.

As shown in FIGURE 6-3 and FIGURE 6-4, the results contain a large number of ()-values. The reason is simple: without the presence of the ()-value, it is useless to do the control binding. Since the single-threading module generates the ()-value, it must always be executed before the control binding module.

```
Run-Time Rules
  ((plus m) n) => m+n
  ((times m) n) => m*n
  (eq0 n) => n=0
  (pred n) => n-1
  (((if true) f) g) => f
  (((if true) f) g) => g
  empty => s:=lami.zero
  (access i) => s(i)
  ((update i) n) => s:=[i|->n]s
```

FIGURE 6-1

```
Compile-Time Rules
  ($C ((; c1) c2)) => (($0 c2) c1)
  (($0 c2) c1) => (($C c2) (($C c1) ()))
  ($C ((:= i) e)) => (($1 i) e)
  (($1 i) e) => ((update i) (($E e) ()))
  ($E ((+ e1) e2)) => (($2 e1) e2)
  (($2 e1) e2) => ((plus (($E e1) ())) (($E e2) ()))
  ($E (# n)) => ($3 n)
  ($E (@ i)) => ($4 i)
  ($4 i) => (access i)
  ($3 0) => zero
  ($3 1) => one
  ($3 2) => two
  ($3 3) => three
  ($3 4) => four
  ($3 5) => five
```

FIGURE 6-2

6.3. Control Binding, Lambda-Lifting and Single-Threading.

The results are same as the ones in section 6.2.

6.4. Lambda-Lifting, Single-Threading and Control Binding.

Although the results in FIGURE 6-5 and 6-6 are closest to the ones in FIGURE 5-8 and 5-9, they are not smaller, however. This is simply because the resultant rules con-

```
Run-Time Rules
  ((plus m) n) => m+n
  ((times m) n) => m*n
  (eq0 n) => n=0
  (pred n) => n-1
  (((if true) f) g) => f
  (((if true) f) g) => g
  empty => s:=lami.zero
  ((access i) ()) => s(i)
  (((update i) n) ()) => s:=[i|->n]s
```

FIGURE 6-3

```
Compile-Time Rules
  (($C ((; c1) c2)) ()) => (($C c2) (($C c1) ()))
  (($C ((:= i) e)) ()) => (((update i) (($E e) ())) ())
  (($E ((+ e1) e2)) ()) => ((plus (($E e1) ())) (($E e2) ()))
  (($E (# n)) ()) => ($N n)
  (($E (@ i)) ()) => ((access i) ())
  ($N 0) => zero
  ($N 1) => one
  ($N 2) => two
  ($N 3) => three
  ($N 4) => four
  ($N 5) => five
```

FIGURE 6-4

tain a few more rules for new combinators.

6.5. Lambda-Lifting, Control Binding and Single-Threading.

By comparison, the results in FIGURE 6-7 and 6-8 appear to be the biggest. Ordering the modules in this manner clearly produced the worst results.

6.6. Summary.

After briefly examining the results from all six test cases, we learned a few important things. First, the single-threading module must come before the control binding

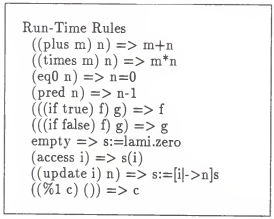


FIGURE 6-5

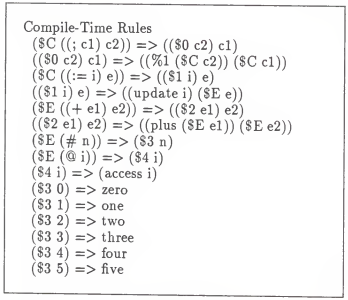


FIGURE 6-6

module. This is because the success of the control binding module is totally dependent upon the ()-values produced by the single-threading module. Second, the control binding module should come before the lambda-lifting module. The reason is that the control binding module eliminates all outermost lambda abstractions in the right-hand expression of the rules without introducing any new combinators. But, lambda-lifting eliminates abstractions by introducing new combinators. Based on these factors, we

```
Run-Time Rules
  ((plus m) n) => m+n
  ((times m) n) => m*n
  (eq0 n) => n=0
  (pred n) => n-1
  (((if true) f) g) => f
  (((if false) f) g) => g
  empty => s:=lami.zero
  ((access i) ()) => s(i)
  (((update i) n) ()) => s:=[i|->n]s
```

FIGURE 6-7

```
Compile-Time Rules
  ($C ((; c1) c2)) => (($0 c2) c1)
  ((($0 c2) c1) ()) => (($C c2) (($C c1) ()))
  ($C ((:= i) e)) => (($1 i) e)
  ((($1 i) e) ()) => (((update i) (($E e) ())) ())
  ($E ((+ e1) e2)) => (($2 e1) e2)
  ((($2 e1) e2) ()) => ((plus (($E e1) ())) (($E e2) ()))
  ($E (# n)) => ($3 n)
  (($3 n) ()) => ($N n)
  ($E (@ i)) => ($4 i)
  (($4 i) ()) => ((access i) ())
  ($N 0) => zero
  ($N 1) => one
  ($N 2) => two
  ($N 3) => three
  ($N 4) => four
  ($N 5) => five
```

FIGURE 6-8

conclude that it is best to order the single-threading module first, control binding second and lambda-lifting last.

Chapter 7

Evaluators

In this chapter, the concepts of the **compile-time** and **run-time evaluators** are introduced. Conceptually, the purpose of an evaluator is to apply rewriting rules to its argument until a normal formal is reached. FIGURE 7-1 shows the data flow of the compile-time and run-time evaluators.

7.1. Compiled-Time Evaluator.

The purpose of a compile-time evaluator is to perform **compile-time computations** that are encoded in a set-of-equations language definition. Examples of compile-time computations are translation to intermediate code, symbol table building, type checking, and constant folding. The input argument to the evaluator is the program to be compiled. The compile-time evaluator uses the "compile-time rules". (see FIGURE 5-9.) The rewriting rules and the expression to be evaluated are represented by trees. An easy way to convert the expression to one that uses tree structure is to run it through the parser. The central idea of our implementation of the evaluator is to play a tree matching game. Each subexpression in the expression is matched against the left-hand expression of the rule. If a match is found, the subexpression is replaced by the right-hand expression of the rule. Using these techniques repeatedly, each subexpression is simplied as far as possible until a normal form is formed. As an example, suppose the

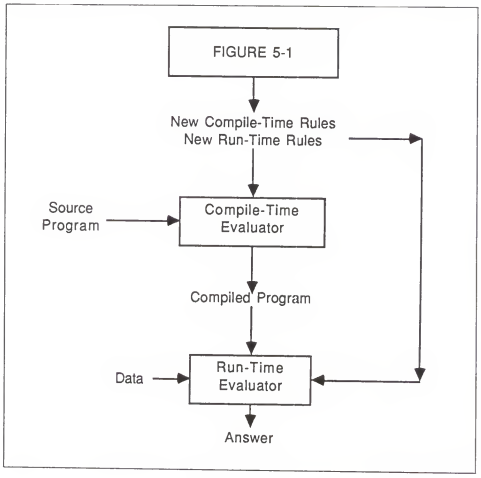


FIGURE 7-1

evaluator is to evaluate the expression:

$C ((;((:= A) #0)) ((:= B) @A)).

Using the compile-time rules in FIGURE 5-9, the evaluator makes the following reductions:

```
=> %1 ($C ((:= B) @A)) ($C (:= A #0))
=> %1 (update B ($E @A)) ($C (:= A #0))
=> %1 (update B (access A)) ($C (:= A #0))
=> %1 (update B (access A)) (update A ($E #0))
=> %1 (update B (access A)) (update A ($N 0))
=> %1 (update B (access A)) (update A zero)
```

The reductions proceed from the left to right; the left subtree is reduced first before

the right subtree. The resultant normal form expression contains a few run-time operators; they are *%1*, *update* and *access*. They cannot be simplied any further because they are run-time-dependent store algebras. The operator *%1* is special and section 7.2 clarifies it.

If one uses the compile-time rules in FIGURE 5-3, the translation of the program above would be the expression

lam s. (lam s. update B (lam s. access A s) s) (lam s. update A (lam s. zero) s)

Notice that the resultant expression contains a large number of trivial bindings of the form (lam s.E)s. Thus, we have justified that inefficient code would be generated if an unevaluated compiled-time rules like the one in FIGURE 5-3 is used.

7.2. Run-Time Evaluator.

The purpose of the run-time evaluator is to perform *run-time computations*. The input argument to the evaluator is the output from the compile-time evaluator. The run-time evaluator uses the "run-time rules". (see FIGURE 5-8.) The run-time evaluator has not been implemented. A general notion of how it works is provided here. Let us consider the expression

%1 (update B (access A)) (update A zero).

The operator *%1* in it is sometimes called a "control structure" for it distributes control to its arguments. The subexpressions (update B (access A)) and (update A zero) are arguments to this operator. The operator *%1* first grants the control to its rightmost argument. The leftmost argument gains the control after the rightmost argument has released it. The run-time evaluator uses the following simplification strategy when evaluating the expression above using the run-time rules in FIGURE 5-8.

```
  %1 (update B (access A)) (update A zero)    <>
=> %1 (update B (access A)) ()                <(A, zero)>
```

```
=> (update B (access A))          <(A, zero)>
=> (update B zero)                <(A, zero)>
=> ()                             <(B, zero),(A, zero)>
```

The values () are the simplified results from the *update* operations. The values <> and <(B, zero),(A, zero)> are the initial and final state of the global store variable respectively.

The are two ways to implement the run-time evaluator. Using software to simulate the global store variable is one of the possible ways. Under this approach, the store variable is implemented as a list of identifier-number pairs. The *update* operation concatenates a new pair to the list. The *access* operation lookups the value for a given identifier from the list. It is not an ideal approach, although it is less expensive to implement the evaluator this way.

A more practical approach would be to design a machine which treats the global store variable as its primary storage. Using this approach, the operators *update* and *access* can be encoded as primitive machine instructions. As a result, we gain a faster implementation this way. Is it an ongoing research topic to design a machine that matches semantic definitions.

Chapter 8

Conclusions

An automated tool for compiler generation has been developed. Most of the work was devoted to making it capable of performing **partial evaluation**. Partial evaluation in a form of compile-time simplification make use of the techniques of single-threading, control binding and lambda-lifting. Through this research, we discovered that one can get the best results by applying **single-threading first**, **control binding second**, and **lambda-lifting last**. Another desirable feature which is also included in the system is the ability to perform type checking and parsing. Thus, a language designer who uses the system need not check by hand the well-definedness of a language.

Virtually any denotational definition can be implemented by the system. Besides that, the generated compilers are small and the compiled programs run faster. But most important is the fact that it is an automated system that produces correct compilers from a language's formal specifications.

Although users of the generated compilers are forced to deal with programs written in the typed lambda calculus, there are ways to avoid this. Peyton Jones proposed algorithms which allow one to translate a high-level functional program into one which uses the lambda calculus [4]. By doing this, the lambda calculus is viewed as an intermediate language between the high level language program and the concrete implementation. In our framework, the concrete implementation can be treated as our

compiler generator system. Consequently, the users do not have to deal with the lambda calculus.

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers, Principle, Techniques and Tools. Addison-Wesley, Reading, Mass., 1987.

[2] Ghezzi Carlo and Mehdi Jazayeri. Programming Language Concepts. John Wiley & Sons, Inc., New York, 1982.

[3] Harper, Robert. Introduction to Standard ML. Technical report, Laboratory for Foundation of Computer Science, Department of Computer Science, University of Edinburgh, 1986.

[4] Peyton Jones, S.L. The Implementation of Functional Programming Languages. Prentice-Hall International, Englewood Cliffs, NJ, 1987.

[5] Schmidt, D.A. Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Boston, 1986.

[6] Schmidt, D.A. Detecting Global Variables in Denotational Specifications. ACM Trans. Prog. Lang. Syst., 7 (1985) 299-310.

[7] Schmidt, D.A. An Implementation from A Direct Semantics Definition. In Programs as Data Objects, Lecture Notes in Computer Science 217, Springer, Berling, (1985) 222-235.

[8] Schmidt, D.A. Detecting Stack-Based Environments in Denotational Definitions. Science of Computer Programming, (1988) 107-131.

[9] Wikstrom Ake. Functional Programming Using Standard ML. Prentice Hall, Englewood Cliffs, NJ, 1987.

Appendix A

This appendix contains listings of the source files for the compiler generator system.

Type.sml

Pretty_Print.sml

Scan_Parse_Type.sml

St_Trans.sml

CbPartI.sml

CbPartII.sml

Lam_Lifting.sml

Eta.sml

Main.sml

Evaluator.sml

Lang_Def

```
(* File name: Type.sml

   Date completed: 4-1-89

   Purpose: This file contains user-defined data types.

   Input: None

   Output: None *)

=============================================================
Declaration                    of                  Data        Types
=============================================================

datatype data_type = nat|
                     booll|
                     store|
                     iden|
                     cmd|
                     numeral|
                     expr|
                     func of data_type * data_type;

datatype tree = lam of tree * tree * data_type|
                apply of tree * tree * data_type|
                idenfy of string * data_type|
                const of string * data_type;

datatype constant = a_const| not_const|unused;

datatype enviroment = typelist of (string * constant * data_type) list;

datatype free_id_list = ids of (string * data_type) list;

datatype rewrite_rules = rule of (tree * tree) list;

datatype lifted_table = ttable of int * rewrite_rules;

System.Control.Print.printDepth :=50;
```

```
(* File name : Pretty_Print.sml

   Date completed : 4-1-89

   Purpose: To analyse the rewriting rules and prints them in
            such a way that the structure of the rules become
            clearly visible.

   Input : A pair of lists. The first list corresponds to the
           Run-Time Rules and the second list corresponds to the
           Compiler-Time Rules.

   Output : The pretty printed Run-Time Rules and Compiler-Time Rules.
            The Run-Time Rules and Compile-Time Rules still remain
            a pair of lists. *)

=========================================================
pretty                                                printer
=========================================================

fun pretty_print ttree =
   case ttree of
     lam(ttree1,ttree2,ddatatype) =>
       let val str1 = pretty_print(ttree1) in
           let val str2 = pretty_print(ttree2) in
             "lam "^str1^" . "^str2
           end
       end|

     apply(ttree1,ttree2,ddatatype) =>
       let val str1 = pretty_print(ttree1) in
            let val str2 = pretty_print(ttree2) in
              "("^str1^" "^str2^")"
            end
       end|

     const(str,ddatatype) => str|

     idenfy(str,ddatatype) => str;

=========================================================

fun doprint(rn_rules,tr_rules) =
     let fun print_rw [] = output std_out "n"|
         print_rw((lhs,rhs)::rest) =
         let val str_lhs = pretty_print(lhs) in
         let val str_rhs = pretty_print(rhs) in
         let val dummy = output std_out (str_lhs^" => "^str_rhs^"n") in
            print_rw rest
         end
```

```
    end
    end
in
    let val dummy = output std_out ("nrun-time rules"^"n----------n") in
        let val dummy = print_rw rn_rules in
            let val dummy = output std_out ("ncompile-time rules"^
                        "n----------n") in print_rw tr_rules
            end
        end
    end
end;
```

```
(* File name      : Scan_Parse_Type.sml

   Date completed : 4-1-89

   Purpose: This file handles the scanning, parsing and
            type-checking.

   Input : A pair of lists of strings. The first list corresponds
           to the Run-Time Rules and the second list corresponds
           to the Compiler-Time Rules.

   Output : The Run-Time Rules and Compiler-Time Rules. Each rule is
            represented by a pair of parse trees.*)

=============================================================
Print                                 Error                 Message
=============================================================

exception error;

fun found_error message =
   let val dummy = output std_out ("\n---- "^message^" ----\n") in
      raise error
   end;

=============================================================
Scanner
=============================================================

fun reverse word ans =
   if word = nil then
      ans
   else
      let val str = hd word in
         reverse (tl word) (str::ans)
      end;

fun gather_word strlst word =
   if strlst = nil then
      nil
   else let val token = hd strlst in
        if token = " " then
          (implode (reverse word nil))::gather_word (tl strlst) nil
        else
          gather_word (tl strlst) (token::word)
        end;

fun scan str =
   let val strlst= explode str in
      gather_word strlst nil
```

```
end;

=============================================================
Loading                    Predefined                Environment
=============================================================

fun load_env empty_env =
  let val typelist(empty_list) = empty_env in

    ("update", a_const, func(iden,func(nat,func(store,store))))::
    ("access", a_const, func(iden,func(store,nat)))::
    ("empty", a_const, store)::
    ("$C", a_const, func(cmd,func(store,store)))::
    ("$E", a_const, func(expr,func(store,nat)))::
    ("$N", a_const, func(numeral,nat))::
    ("$I", a_const, func(expr,iden))::
    ("@", a_const, func(iden,expr))::
    ("#", a_const, func(numeral,expr))::
    (";", a_const, func(cmd,func(cmd,cmd)))::
    (":=", a_const, func(iden,func(expr,cmd)))::
    ("+", a_const, func(expr,func(expr,expr)))::

    ("plus", a_const, func(nat,func(nat,nat)))::
    ("times", a_const, func(nat,func(nat,nat)))::
    ("eq0",a_const, func(nat,bool))::
    ("pred",a_const, func(nat,nat))::
    ("if",a_const, func(bool,func(func(nat,nat),func(func(nat,nat),
    func(nat,nat)))))::
    ("Yop",a_const,func(func(nat,nat),func(nat,nat)))::

    ("A", a_const, iden)::("B", a_const, iden)::("C", a_const, iden)::
    ("X", a_const, iden)::("Y", a_const, iden)::("Z", a_const, iden)::

    ("0", a_const, numeral)::("1", a_const, numeral)::
    ("2", a_const, numeral)::("3", a_const, numeral)::
    ("4", a_const, numeral)::("5", a_const, numeral)::
    ("6", a_const, numeral)::("7", a_const, numeral)::
    ("8", a_const, numeral)::("9", a_const, numeral)::

    ("zero", a_const, nat)  :: ("one", a_const, nat)   ::
    ("two", a_const, nat)   :: ("three", a_const, nat) ::
    ("four", a_const, nat)  :: ("five", a_const, nat)  ::
    ("six", a_const, nat)   :: ("seven", a_const, nat) ::
    ("eight", a_const, nat) :: ("nine", a_const, nat)  ::
    ("true", a_const, bool):: ("false", a_const,bool)::

    (*The following constants won't be here if the run time evaluator is implemented*)

    ("s(i)", a_const,nat)  :: ("s:=[i]->n]s", a_const,store) ::
    ("m+n", a_const,nat)   :: ("m*n", a_const,nat)           ::
```

```
    ("n=0", a_const,booll) :: ("n-1", a_const,nat)        ::
    ("s:=lami.zero", a_const,store) ::
    empty_list
  end;
```

```
=============================================================
Update Environment Try to define the same identifier with more than one data type is forbited
=============================================================
```

```
fun update_env(list,id,typed) =
  if list = nil then
    (id, not_const, typed)::nil
  else
    let val (str, const_or_id, data_type) =  hd list in
      if (id = str) then
        if (typed = data_type) then
          list
        else
          found_error ("identifier "^id^" has already been"^" defined ...")
      else
        hd list::update_env(tl list,id,typed)
  end;
```

```
=============================================================
Access Environment Try to access the data type for an undefined identifier is forbited
=============================================================
```

```
fun access_env(list,id) =
  if list = nil then
    found_error ("identifier "^id^" is undefined ...")
  else
    let val (str, const_or_id, data_type) =  hd list in
      if id = str then
        (const_or_id, data_type)
      else
        access_env(tl list,id)
  end;
```

```
=============================================================
Converting          String          To          Datatype
=============================================================
```

```
fun str_to_datatype rest =
  let val typed = hd rest in
    let val rest = tl rest in
      if typed = "nat" then (nat, rest)
      else if typed = "booll" then (booll, rest)
      else if typed = "store" then (store, rest)
      else if typed = "iden" then (iden, rest)
      else if typed = "cmd" then (cmd, rest)
```

```
        else if typed = "numeral" then (numeral, rest)
        else if typed = "expr" then (expr, rest)
        else if typed = "(" then
        (* a function type *)
            let val (type1, rest1) = str_to_datatype rest in
              if hd rest1 = "->" then
                let val (type2, rest2) = str_to_datatype (tl rest1) in
                  if hd rest2 = ")" then
                    (func(type1, type2), tl rest2)
                  else
                    found_error "syntax_err...  missing ')'"
                end
              else
                found_error "syntax_err...  missing '->'"
            end
        else found_error ("type '"^typed^"' is undefined ...")
    end
  end;

=============================================================
Parse                    and                    Type                    Check
=============================================================

fun parse_type [] type_list = found_error "no input"|
    parse_type (word::rest) type_list =
    if word = "lam" then        let val id = (hd rest) in
          if (hd(tl rest)) = ":" then
            let val (typed, rest) = str_to_datatype(tl(tl rest)) in
              let val type_list = update_env(type_list,id,typed) in
                if hd rest = "." then
                  let val (sub_tree,tree_type,rest,type_list) =
                    parse_type (tl rest) type_list in
                      if hd rest = "mal" then
                        (lam(idenfy(id, typed),sub_tree,func(typed,tree_type)).
                        func(typed,tree_type),tl rest,type_list)
                      else found_error ("syntax_err... "^" missing 'mal'")
                  end
                else found_error "syntax_err ... missing '.'"
              end
            end
          else found_error "syntax_err ... missing ':'"
        end
    else if word = "(" then
          let val (sub_tree1,tree_type1,rest1, type_list) = parse_type rest type_list in
            let val (sub_tree2,tree_type2,rest2, type_list) =
              parse_type rest1 type_list in
              let val func(arg_type,result_type) = tree_type1 in
                if arg_type = tree_type2 then
                  if hd rest2 = ")" then
                    (apply(sub_tree1,sub_tree2,result_type),
```

```
                    result_type, tl rest2, type_list)
                  else
                    found_error "syntax_err ... missing ')'"
                else
                  found_error "type error ..."
              end
            end
          end
    else
        let val (const_or_id, tree_type) = access_env(type_list, word) in
          if const_or_id = not_const then
            (idenfy(word, tree_type), tree_type, rest, type_list)
          else
            (const(word, tree_type), tree_type, rest, type_list)
        end
```

```
=============================================================
Main          Entry          to          Load_Env          and          Parse_Type
=============================================================
```

```
fun spt(str_list,type_list) =
   let val type_list = load_env (typelist type_list) in
      let val (typed_tree, tree_type, not_used, type_list) =
         parse_type (scan(str_list)) type_list  in
         (typed_tree,tree_type,type_list)
      end
   end;
```

```
(* File name      : St_Trans.sml

   Date completed : 4-1-89

   Purpose: Implement the Single-Threading Criteria and
            the Transformation Algorithm.

   Input : A pair of lists. The first list corresponds to the
           Run-Time Rules and the second list corresponds to the
           Compiler-Time Rules.

   Output : The Run-Time Rules and Compiler-Time Rules. Each rule is
            represented by a pair of parse trees. If the rules are
            single-threaded, they transformed to ones that use
            the global store variable. *)

=============================================================
Implementation          of          The          Single-Threading          Criteria
=============================================================

fun sgl_thrd (lam(ttree1, ttree2, ddata_type)) =
      let val (st, const_or_id, stid, stexpr) = sgl_thrd (ttree2) in
        if st then
          let val func(datatype1, datatype2) = ddata_type in
            if datatype1 = store then
              if stid <> "no_active" then
                if const_or_id = not_const then
                  let val idenfy(id, t) = ttree1 in
                    if id = stid then
                      (true, unused, "no_active", "no_active")
                    else
                      (false, unused, "unused", "unused")
                  end
                else
                  (true, const_or_id, stid, stexpr)
              else
                (true, const_or_id, stid , stexpr)
            else
              if stexpr = "no_active" then
                (true, const_or_id, stid, stexpr)
              else (false, unused, "unused", "unused")
          end
        else (false, unused, "unused", "unused")
      end|

  sgl_thrd (apply(ttree1, ttree2, ddata_type)) =
    let val (st1, const_or_id1, stid1, stexpr1) = sgl_thrd (ttree1) in
      let val (st2, const_or_id2, stid2, stexpr2) = sgl_thrd(ttree2) in
        if st1 andalso st2 then
          if ddata_type = store then
```

```
                if (stexpr1 <> "no_active") andalso
                  (stexpr2 <> "no_active") then
                    if (stexpr1 = stexpr2) andalso
                      (const_or_id1 = not_const) then
                      (true, const_or_id1, stexpr1, "apply")
                    else
                      (false, unused, "unused", "unused")
                else
                    (true, const_or_id2, stid2, "apply")
            else
                if (stexpr1 <> "no_active") andalso
                  (stexpr2 <> "no_active") then
                    if (stexpr1 = stexpr2) andalso
                      (const_or_id1 = not_const) then

                      (true, const_or_id1, stexpr1, stexpr1)
                    else
                      (false, unused, "unused", "unused")
                else if (stexpr1 = "no_active") andalso
                      (stexpr2 <> "no_active") then
                        if stexpr2 = "apply" then
                          (false, unused, "unused", "unused")
                        else
                          if const_or_id2 = not_const then
                            (true, const_or_id2, stexpr2, stexpr2)
                          else
                            (false, unused, "unused",
                            "unused")
                else (true, unused, "no_active", "no_active")
          else (false, unused, "unused", "unused")
      end
    end|

  sgl_thrd (idenfy(s, ddata_type)) =
      if ddata_type = store then
        (true, not_const, s, s)
      else (true, not_const, "no_active", "no_active")|

  sgl_thrd (const(s, ddata_type)) =
      if ddata_type = store then
        (true, a_const, s , s)
      else (true, a_const, "no_active", "no_active");

==========================================================
Perform          Global          Variable          Transformation
==========================================================

fun transform (lam(ttree1, ttree2, ddata_type)) =
      let val idenfy(id, t) = ttree1 in
        if t = store then
```

```
            (lam(const ("()", t), transform (ttree2), ddata_type))
          else
            (lam(ttree1, transform(ttree2), ddata_type))
      end|

  transform (apply(ttree1, ttree2, ddata_type)) =
      (apply(transform(ttree1), transform(ttree2), ddata_type))|

  transform (idenfy(s, ddata_type)) =
      if ddata_type = store then
        const ("()", ddata_type)
      else
        idenfy(s, ddata_type)|

  transform everything_else = everything_else;
```

```
=============================================================
Transform                    Single-Threaded                 Rules
=============================================================
```

```
fun trans_rw [] = []|
  trans_rw((lhs,rhs)::rest) =
  let val lhs = transform(lhs) in
    let val rhs = transform(rhs) in
      (lhs,rhs)::trans_rw(rest)
    end
  end;
```

```
=============================================================
Main      Entry      to      The      Single-Threading      Check
=============================================================
```

```
fun st_trans(rn_rules,tr_rules) =

  let
    fun st_rw_rule [] = true
      |st_rw_rule((lhs,rhs)::rest) =
      let val (st,un1,un2,un3) = sgl_thrd(rhs) in
        if st then
          st_rw_rule(rest)
        else
          found_error ((pretty_print rhs)^" is not single-threaded")
      end
  in
    let val ok = (st_rw_rule rn_rules) andalso (st_rw_rule tr_rules) in
      (trans_rw rn_rules, trans_rw tr_rules)
    end
  end;
```

```
(* File name      : CbPartI.sml

   Date completed : 4-1-89

   Purpose: Implement Part I of the extended control
            binding technique.

   Input : A pair of lists. The first list corresponds to the
           Run-Time Rules and the second list corresponds to the
           Compiler-Time Rules.

   Output : The Run-Time Rules and Compiler-Time Rules. Each rule is
            represented by a pair of parse trees. The rules are transformed. *)

=============================================================
Special                    Lambda-Lifting                Technique
=============================================================

fun find_bind_id rhs =
    case rhs of
      lam(ttree1,ttree2,ddatatype) => ttree1::find_bind_id ttree2
    |everything_else => [];

=============================================================

fun find_rhs rhs =
    case rhs of
      lam(ttree1,ttree2,ddatatype) => find_rhs ttree2
    |found_rhs => found_rhs;

=============================================================

fun apply_lhs(lhs,[]) = lhs
  |apply_lhs(lhs,rhs::rest) =
   let val (apply(ttree1,ttree2,ddatatype)) = lhs in
       let val func(angs,ans) = ddatatype in
           apply_lhs(apply(lhs,rhs,ans),rest)
       end
   end;

=============================================================
Main          Entry          to          Special          Lambda-Lifting          Technique
=============================================================

(* Convert Rule E1 => LAM (). E2 TO  E1 () => E2.*)

fun nlifting(rn_rules,tr_rules) = let
    fun new_lift [] = []
      |new_lift ((lhs,rhs)::rest) =
       let val iden_list = find_bind_id rhs in
```

```
    let val new_rhs = find_rhs rhs in
        (apply_lhs(lhs,iden_list),new_rhs)::new_lift rest
    end
end in
(new_lift rn_rules, new_lift tr_rules) end;
```

```
(* File name     : CbPartII.sml

   Date completed : 4-1-89

   Purpose: Implement part II of the extended control binding
            technique.

   Input : A pair of lists. The first list corresponds to the
           Run-Time Rules and the second list corresponds to the
           Compiler-Time Rules.

   Output : The Run-Time Rules and Compiler-Time Rules. Each rule is
            represented by a pair of smaller parse trees. *)

=========================================================
Step      No.     1     of     Extended     Control     Binding     Technique
=========================================================

(* Rewrite (LAM () .E) () to E *)

fun step1(lam(ttree1, ttree2, ddatatype)) =
      let val (dummy1, dummy2, ttree2 ) = step1(ttree2) in
          (ttree1, ttree2, lam(ttree1, ttree2, ddatatype))
      end|

    step1(apply(ttree1, ttree2, ddatatype)) =
       let val (ttree11, ttree12, ttree1) = step1(ttree1) in
       let val (dummy1, dummy2, ttree2) = step1(ttree2) in
       let val ttree = apply(ttree1,ttree2,ddatatype) in
           case ttree11 of
               const(oprt,oprt_type) => if (oprt = "()") andalso (ttree11 = ttree2) then
                                            (ttree12,ttree12,ttree12)
                                        else (ttree,ttree,ttree)|
               everything_else       => (ttree,ttree,ttree)
       end end end|

    step1(ttree) = (ttree, ttree, ttree);

=========================================================

fun apply_step1 [] = []
  | apply_step1 ((lhs,rhs)::rest) =
    let val (unused1,unused2,lhs) = step1 lhs in
        let val (unused1,unused2,rhs) = step1 rhs in
            ((lhs,rhs)::apply_step1 rest)
        end
    end;

=========================================================
Step      No.     2     of     Extended     Control     Binding     Technique
```

```
==========================================================

(* If All Uses of 'oprt' in The LHS of Rewrite Rules Have The
   Form (oprt e1 .. en ()) => rhs Then
   1. Alter All Uses of The 'oprt' in The LHS of Rewrite Rules to (oprt e1 .. en) => rhs.
   2. All Uses of () To The 'oprt' in The RHS Also Disappear.
   3. All Uses of The 'oprt' That Are Lacking The () in RHS Are Enclosed
      By A New Combinator. *)

fun insert_list(oprt_new,d,[]) = (oprt_new,d)::[]|

    insert_list(oprt_new,d1,(oprt_old,d2)::rest) =
    if oprt_new = oprt_old then
      ((oprt_old,d2)::rest)
    else
      ((oprt_old,d2)::insert_list(oprt_new,d1,rest));

==========================================================

fun get_info(ttree,list) =

    let fun go_get_info(ttree,list) =
        case ttree of
        lam(ttree1,ttree2,ddatatype) => go_get_info(ttree2,list)
            |apply(ttree1,ttree2,ddatatype) =>
             let val (list1,oprt1,depth1) = go_get_info(ttree1,list) in
                  let val (list2,oprt2,dummy) = go_get_info(ttree2,list1) in
                      if oprt2 = "()" then
                        let val list = insert_list(oprt1,depth1+1,list2) in
                            (list,"dummy",0)
                        end
                      else
                            (list2,oprt1,depth1+1)
                  end
             end
            |const(oprt,ddatatype) => (list,oprt,0)
            |idenfy(oprt,ddatatype) => (list,"dummy",0)
    in
        let val (list,dummy1,dummy2) = go_get_info(ttree,list) in
            list
        end
    end;

==========================================================

fun retrieve_info(oprt_new,[]) = (oprt_new,0)|

    retrieve_info(oprt_new,(oprt_old,d)::rest) =
    if oprt_new = oprt_old then
      (oprt_new,d)
```

```
    else
      retrieve_info(oprt_new,rest);

=========================================================

fun mk_rn_cbnt(apply(ttree1,ttree2,ddatatype),c,cbnt) =
    let
        fun find_type ttree =
            case ttree of
                lam(ttree1,ttree2,dtype) => dtype
              |apply(ttree1,ttree2,dtype) => dtype
              |const(ttree1,dtype) => dtype
              |idenfy(ttree1,dtype) => dtype
    in
        let val type1 = find_type ttree1 in
        let val type2 = find_type ttree2 in
            let val cbnt_type = func(type1,func(type2,type1)) in
            let val c = c + 1 in
            let val cbnt_name = "%"^makestring c in
                (apply(apply(const(cbnt_name,cbnt_type),ttree1,func(type2,type1)),ttree2,type1),
                0,c,(apply(apply(const(cbnt_name,cbnt_type),idenfy("c",type1),
                func(type2,type1)),const("()",type2),type1),idenfy("c",type1))::cbnt)
            end
            end
            end
        end
        end
    end;

=========================================================

fun bind_tree(ttree,list,c,cbnt) = (* c = count , cbnt = runtime combinator *)

    let fun adjust_type(d,ddatatype)=
        if d=2 then
            let val func(ang,ans)=ddatatype in
                ans
            end
        else
            let val func(ang,ans)=ddatatype in
                func(ang,adjust_type(d-1,ans))
            end
    in
        case ttree of
        lam(ttree1,ttree2,ddatatype) =>
          let val (ttree2,d,c,cbnt) = bind_tree(ttree2,list,c,cbnt) in
                (lam(ttree1,ttree2,ddatatype),0,c,cbnt)
          end
        |apply(ttree1,ttree2,ddatatype) =>
          let val (ttree1,d1,c,cbnt) = bind_tree(ttree1,list,c,cbnt) in
```

```
            let val (ttree2,d2,c,cbnt) =
                    bind_tree(ttree2,list,c,cbnt) in
                  if d1 = 0 then
                     (apply(ttree1,ttree2,ddatatype),0,c,cbnt)
                  else
                     if (d1 = 1) then
                        if ttree2 = const("()",store) then
                           (ttree1,0,c,cbnt)
                        else
                           (* mk_cb returns -> tree,0,c,cbnt *)
                           mk_rn_cbnt(apply(ttree1,ttree2,ddatatype),c,cbnt)
                     else
                     (apply(ttree1,ttree2,adjust_type(d1,ddatatype)),d1-1,c,cbnt)
            end
        end

       |const(oprt,ddatatype) =>
            let val (oprt,d) = retrieve_info(oprt,list) in
               if d>0 then
                  (const(oprt,adjust_type(d+1,ddatatype)),d,c,cbnt)
               else
                  (ttree,d,c,cbnt)
            end
       |idenfy(oprt,ddatatype) =>
          (ttree,0,c,cbnt)
  end;

=============================================================

fun get_info_rw([],list) = list
   |get_info_rw((lhs,rhs)::rest,list)=
    let val list = get_info(lhs,list) in
        let val list = get_info(rhs,list) in
            get_info_rw(rest,list)
        end
    end;

=============================================================

fun bind_tree_rw([],list,c,cbnt,rules) = (c,cbnt,rules)|
    bind_tree_rw((lhs,rhs)::rest,list,c,cbnt,rules) =
    let val(lhs,unused1,c,cbnt) = bind_tree(lhs,list,c,cbnt) in
        let val(rhs,unused2,c,cbnt) = bind_tree(rhs,list,c,cbnt) in
            bind_tree_rw(rest,list,c,cbnt,(lhs,rhs)::rules)
        end
    end;

=============================================================

fun get_bind (rn_rules,tr_rules) =
```

```
    let val list = get_info_rw(rn_rules,[]) in
        let val list = get_info_rw(tr_rules,list) in
            list
        end
    end;

=================================================================
Entry       Point       To      The     Extended    Control     Binding     Algorithm
=================================================================

fun ct_bind(rn_rules,tr_rules) =
    let
      fun apply_step2(rn_rules,tr_rules) =
          let fun cat([], rn_rules) = rn_rules
              |cat(hdd::rest,rn_rules) = cat(rest,hdd::rn_rules)
          in
              let val bind = get_bind(rn_rules,tr_rules) in
                  let val (c,cbnt,rn_rules) =
                      bind_tree_rw(rn_rules,bind,0,nil,nil) in
                      let val (c,cbnt,tr_rules) =
                          bind_tree_rw(tr_rules,bind,c,cbnt,nil) in
                          (reverse (cat(cbnt,rn_rules)) nil,reverse tr_rules nil)
                      end
                  end
              end
          end
    in
        let val rn_rules = apply_step1 rn_rules in
            let val tr_rules = apply_step1 tr_rules in
                apply_step2(rn_rules,tr_rules)
            end
        end
    end;
```

```
(* File name      : Lam_Lifting.sml

   Date completed : 4-1-89

   Purpose: Implement the lambda-lifting algorithm.

   Input : A pair of lists. The first list corresponds to the
           Run-Time Rules and the second list corresponds to the
           Compiler-Time Rules.

   Output : The Run-Time Rules and Compiler-Time Rules. Each rule is
            represented by a pair of parse trees. The rules
            have no lambda operators but may be augmented with new rules.*)

=========================================================
Implement                    The                Beta_Abstraction
=========================================================

(* Function build_abst Takes Out All Innermost Lambda Abstractions's
   Free Variables As Extra Parameters. *)

fun build_abst(ttree1, ttree2, ddatatype, ids []) = (ttree2, ids [])|

    build_abst(ttree1, ttree2, ddatatype, ids (head::list)) =
       let val s1 = case ttree1 of
          idenfy(s, ddatatype1) => s|
          const(s, ddatatype1) => s|
          everthing_else => "will_not_happend" in

          let val (s2, ddatatype2) = head in
             if s1 = s2 then
               (* s2 already has a binding identifier *)
               build_abst(ttree1, ttree2 ,ddatatype, ids list)
             else
               (* build new binding identifier for s2 *)
               let val new_type = func(ddatatype2, ddatatype) in
               let val ttree2 = (lam(idenfy(s2, ddatatype2),ttree2, new_type)) in
                  let val (ttree2, list) = build_abst(ttree1, ttree2, new_type, ids list) in
                     (apply(ttree2, idenfy(s2, ddatatype2), ddatatype), list)
                  end
               end
               end
       end
    end;

=========================================================

(* Function search_free Searches Free Variables in The Innermost Lambda Abstraction and
Records Them in A List Called free_ids *)
```

```
fun search_free(ttree,free_ids) =
   case ttree of
      lam(ttree1, ttree2, ddatatype) =>
         let val (ttree2, free_ids) = search_free(ttree2, free_ids) in
            let val ttree2 = (lam(ttree1, ttree2, ddatatype)) in
               build_abst(ttree1, ttree2, ddatatype, free_ids)
            end
         end

    |apply(ttree1, ttree2, ddatatype) =>
         let val (ttree1, free_ids1) = search_free(ttree1, free_ids) in
            let val (ttree2, free_ids2) = search_free(ttree2, free_ids1) in
               (apply(ttree1, ttree2, ddatatype), free_ids2)
            end
         end

    |idenfy(s, ddatatype) =>
         let val ids list = free_ids in
            (idenfy(s, ddatatype), ids((s, ddatatype)::list))
         end

    |const(s, ddatatype) => (const(s, ddatatype), free_ids);

fun beta_abst ttree =
      let val (new_tree, empty_list) = search_free(ttree, ids[]) in
         new_tree
      end;

=============================================================

(* Function set_rule Construct A New Rewrite Rule *)

fun set_rule(lam(ttree1, ttree2, ddatatype), lhs, rhs) =
        let val func(type1,type2) = ddatatype in
           let val new_lhs = apply(lhs, ttree1,type2) in
                 set_rule(ttree2, new_lhs, rhs)
           end
        end|

   set_rule(new_rhs, lhs, rhs) = (lhs,new_rhs);

=============================================================

(* Function lam_lift Gives Supercombinator A Name And Constructs
  New Supercombinator Definition and Tree *)

fun lam_lift(ttree,table) =
   case ttree of
      lam(ttree1, ttree2, ddatatype) =>
         let val ttable(num, rule list) = table in
```

```
        let val func(type1,type2) = ddatatype in
            let val name = "$"^makestring num in
            let val new_lhs = apply(const(name, ddatatype),ttree1,type2) in
            let val (lhs,rhs) = set_rule(ttree2, new_lhs, new_lhs) in
                (const(name, ddatatype),(ttable(num+1, rule((lhs,rhs)::list))))
            end end end
        end end


    |apply(ttree1, ttree2, ddatatype) =>
        let val (ttree1, table) = lam_lift(ttree1, table) in
            let val (ttree2, table) = lam_lift(ttree2, table) in
                (apply(ttree1, ttree2, ddatatype), table)
            end
        end

    |everything_else => (everything_else, table);

=========================================================

(* Function dt_sc Finds The Innermost Lambda Anstraction and Then
   Calls Functions beta_abst and lam_lift *)

fun dt_sc(ttree, table) =
    case ttree of
        lam(ttree1, ttree2, ddata_type) =>
            let val (ttree2, table) = dt_sc(ttree2, table) in
                let val ttree2 = beta_abst(lam(ttree1, ttree2, ddata_type)) in
                    lam_lift(ttree2, table)
                end
            end

    |apply(ttree1, ttree2, ddata_type) =>
        let val (ttree1, table) = dt_sc(ttree1, table) in
            let val (ttree2, table) = dt_sc(ttree2, table) in
                (apply(ttree1, ttree2, ddata_type), table)
            end
        end

    |everything_else => (everything_else, table);

=========================================================
Lambda          Lift          The          Rewrite          Rules
=========================================================

fun dt_sc_rw(num,rw_rule) =
    let fun cons list num rw_rule =
        case list of
            [] => dt_sc_rw(num,rw_rule)
            |(lhs,rhs)::rest => (lhs,rhs)::cons rest num rw_rule
```

```
  in
  case rw_rule of
    [] => []
   |(lhs,rhs)::rest =>
        let val (lhs,ttable(num,rule list)) = dt_sc(lhs,ttable(num, rule [])) in
            let val (rhs,ttable(num,rule list)) = dt_sc(rhs,ttable(num,rule list)) in
                  (lhs,rhs)::cons (reverse list nil) num rest
            end
        end
  end;
```

```
=============================================================
Main          Entry          To          The          Lambda-Lifting          Algorithm
=============================================================
```

```
fun lam_lifting(rn_rules,tr_rules) =
   let val rn_rules = dt_sc_rw(0,rn_rules) in
      let val tr_rules = dt_sc_rw(0,tr_rules) in
            (rn_rules,tr_rules)
      end
   end;
```

```
(* File name      : Eta.sml

   Date completed : 4-1-89

   Purpose: To implement the eta-reduction.

   Input : A pair of lists. The first list corresponds to the
           Run-Time Rules and the second list corresponds to the
           Compiler-Time Rules.

   Output : The Run-Time Rules and Compiler-Time Rules. Each rule is
            represented by a pair of parse trees. All redundant
            rewriting rule has been optimized out of the list if
            there is any. *)

=============================================================
Perform          Eta          Reduction          on          Rewrite-Rule
=============================================================

fun extract ttree =
   case ttree of
      apply(ttree1,ttree2,ddatatype) =>
         let val (opr,ttree1) = extract ttree1 in
            (opr,apply(ttree1,ttree2,ddatatype))
         end|
      const(opr,ddatatype) => (opr,const("dummy",ddatatype))|
      anything_else => ("dummy",anything_else);

=============================================================

fun build(opr1,opr2,ttree) =
   case ttree of
      apply(ttree1,ttree2,ddatatype) =>
         let val ttree1 = build(opr1,opr2,ttree1) in
            apply(ttree1,ttree2,ddatatype)
         end|
      const(opr,ddatatype) =>
         if opr = opr2 then
            const(opr1,ddatatype)
         else
            const(opr,ddatatype)|
      anything_else => anything_else;

=============================================================

fun is_safe(opr1,opr2,ttree) =
   case ttree of
      apply(ttree1,ttree2,ddatatype) => safe(opr1,opr2,ttree1)|
      const(opr,ddatatype) =>
         if opr = opr2 then
```

```
            true
          else
            false|

      anything_else => false;

=============================================================

fun reduce (opr1,opr2,[])=[]|

    reduce (opr1,opr2,(lhs,rhs)::rest) =
    let val new_lhs = build(opr1,opr2,lhs) in
        (new_lhs,rhs)::(reduce(opr1,opr2,rest))
    end;

=============================================================

fun ok_reduce (opr1,opr2,[])=false|

    ok_reduce (opr1,opr2,(lhs,rhs)::rest) =
    let val ok = is_safe(opr1,opr2,lhs) in
        if ok then
            ok
        else
            ok_reduce(opr1,opr2,rest)
    end;

=============================================================
Main              Entry              To              The              Eta              Reduction
=============================================================

fun do_eta [] = []|

    do_eta((lhs,rhs)::rest) =
    let val (opr1,new_lhs) = extract lhs in
        let val (opr2,new_rhs) = extract rhs in
            if (new_lhs = new_rhs) then
                if ok_reduce(opr1,opr2,rest) then
                    do_eta (reduce(opr1,opr2,rest))
                else
                    ((lhs,rhs)::do_eta rest)
            else
                ((lhs,rhs)::do_eta rest)
        end
    end;
```

```
(* File : Main.sml

   Date completed : 4-1-89

   Purpose: Main module to invoke the compiler generator system.

   Input : A pair of lists. The first list corresponds to the
           Run-Time Rules and the second list corresponds to
           the Compiler-Time Rules. The rules are represented
           by a pair of strings.

   Output : The Run-Time Rules and Compiler-Time Rules. Each
            rule is represented by a pair of parse trees. The
            rules are partially evaluated. *)

=============================================================

fun strip_lam lf_tree =
   case lf_tree of
     lam(ttree1,ttree2,ddatatype) => strip_lam ttree2
    |apply(ttree1,ttree2,ddatatype) => (apply(ttree1,ttree2,ddatatype),ddatatype)
    |idenfy(str,ddatatype) => (idenfy(str,ddatatype),ddatatype)
    |const(str,ddatatype) => (const(str,ddatatype),ddatatype);

=============================================================

fun conv [] = []
   |conv((left_str,right_str)::rest) =
    let val (lf_tree,dummy1,type_list) = spt(left_str, []) in
       let val (lf_tree,t_lf) = strip_lam(lf_tree) in
       let val (rt_tree,t_rt,type_list) = spt(right_str,type_list) in
          if t_lf = t_rt then
            (lf_tree,rt_tree)::conv rest
          else
             found_error ((pretty_print lf_tree)^" => "^(pretty_print rt_tree)^
                    "0ype incompatible of lhs and rhs")
       end
       end
    end;

=============================================================
Main      Module      To      Involke      The      Compiler      Generator      System
=============================================================

fun main(rn_rules,tr_rules) =
   let val rn_rules = conv rn_rules in
   let val tr_rules = conv tr_rules in
      let val dummy = output std_out ("0--------0t cb lift0^"----------0) in
         let val (rn_rules,tr_rules) =
            ct_bind(nlifting(lam_lifting(st_trans(rn_rules,tr_rules)))) in
```

```
        doprint(do_eta rn_rules, do_eta tr_rules)
      end
    end
end
end;
```

```
(* File name      : Evaluator.sml

   Date completed : 4-1-89

   Purpose: To implement the compile-time evaluator to
            perform compile-time computations.

   Input : 1) The compile-time rules
           2) Program to be compiled.

   Output : Compiled program. *)

=============================================================

(* Each Subexpression Is Matched Against The Left-Hand Expression
   Of The Rule. The Return Is Either True or False. *)

fun match(lhs,ttree,tenv) =
   case lhs of
      apply(t11,t12,dtype1) =>
         let val (found,tenv) =
            case ttree of
               apply(t21,t22,dtype2) =>
                  let val (found,tenv) = match(t11,t21,tenv) in
                     if found then
                        match(t12,t22,tenv)
                     else
                        (found,tenv)
                  end
               |anything_else => (false,tenv)
         in
            (found,tenv)
         end
     |const(oprt1,dtype1) =>
         let val (found,tenv) =
            case ttree of
               const(oprt2,dtype2) =>
                  if oprt1=oprt2 then
                     (true,tenv)
                  else
                     (false,tenv)
               |anything_else => (false,tenv)
         in
            (found,tenv)
         end
     |identifier => (true,(identifier,ttree)::tenv);

=============================================================

(* If A Match Is Found, The Subexpression Is Replaced By The
```

```
  Right-Hand Expression Of The Rule. *)

fun replace(rhs,tenv) =
   let
      fun do_replace(identifier,[]) = found_error (pretty_print rhs^" is an illegel rhs rule ")
        |do_replace(identifier,(old,new)::rest) =
         if identifier=old then
            new
         else
            do_replace(identifier,rest)
   in
      case rhs of
        apply(ttree1,ttree2,dtype) =>
           let val ttree1 = replace(ttree1,tenv) in
              let val ttree2 = replace(ttree2,tenv) in
                 apply(ttree1,ttree2,dtype)
              end
           end
        |const(oprt,dtype) => const(oprt,dtype)
        |identifier => do_replace(identifier,tenv)
   end;

=============================================================

fun match_rule(nil,ttree) = ttree
  |match_rule((lhs,rhs)::rest,ttree) =
   let val (found,tenv) = match(lhs,ttree,nil) in
      if found then
        replace(rhs,tenv)
      else
        match_rule(rest,ttree)
   end;

=============================================================
Main          Entry          To          The          Compile-Time          Evaluator
=============================================================

fun eval(tr_rule,prog) =
   let fun keep_eval ttree =
      let val ttree = match_rule(tr_rule,ttree) in
          case ttree of
           apply(ttree1,ttree2,dtype) =>
              match_rule(tr_rule,apply(keep_eval ttree1,keep_eval ttree2,dtype))
          |everything_else => everything_else
      end
   in
      let val (ttree,d1,d2) = spt(prog,[]) in
         pretty_print(keep_eval ttree)
      end
   end;
```

```
(* File name: Lang_Def

   Date completed: 4-1-89

   Purpose: A Sample Language Definition

   Input: None

   Output: None *)

=========================================================
```

Run-Time Rules

```
(* Natural Numbers *)

main [(("lam m : nat . lam n : nat . ( ( plus m ) n ) mal mal ", "m+n ") ::

("lam m : nat . lam n : nat . ( ( times m ) n ) mal mal ", "m*n ") ::

("lam n : nat . ( eq0 n ) mal ", "n=0 ") ::

("lam n : nat . ( pred n ) mal ", "n-1 ") ::

("lam f : ( nat -> nat ) . lam g : ( nat -> nat ) . ( ( ( if true ) f ) g ) mal mal ", "f ") ::

("lam f : ( nat -> nat ) . lam g : ( nat -> nat ) . ( ( ( if false ) f ) g ) mal mal ", "g ") ::

("lam f : ( nat -> nat ) . lam n : nat . ( ( Yop f ) n ) mal mal ", "( f ( ( Yop f ) n ) ) ") ::

(* store *)

("empty ", "s:=lami.zero ") ::

("lam i : iden . lam s : store . ( ( access i ) s ) mal mal ","s(i) ")::

("lam i : iden . lam n : nat . lam s : store . ( ( ( update i ) n ) s ) mal mal mal ", "s:=[i|->n]s
"::[]).
```

Compile-Time Rules

```
(* SC: cmd -> store -> store *)

(("lam c1 : cmd . lam c2 : cmd . ( $C ( ( ; c1 ) c2 ) ) mal mal ", "lam s : store . ( ( $C c2 ) ( (
```

```
$C c1 ) s ) ) mal "):: 

("lam i : iden . lam e : expr . ( $C ( ( := i ) e ) ) mal mal ", "lam s : store . ( ( ( update i ) ( ( $E e ) s ) ) s ) mal "):: 

(* $E: expr -> store -> nat *) 

("lam e1 : expr . lam e2 : expr . ( $E ( ( + e1 ) e2 ) ) mal mal ", "lam s : store . ( ( plus ( ( $E e1 ) s ) ) ( ( $E e2 ) s ) ) mal "):: 

("lam n : numeral . ( $E ( # n ) ) mal ", "lam s : store . ( $N n ) mal "):: 

("lam i : iden . ( $E ( @ i ) ) mal ", "lam s : store . ( ( access i ) s ) mal "):: 

(* $N: numeral -> nat *) 

("( $N 0 ) ", "zero "):::("( $N 1 ) ", "one "):::("( $N 2 ) ", "two "):: ("( $N 3 ) ", "three "):::("( $N 4 ) ", "four "):::("( $N 5 ) ", "five "):: ("( $N 6 ) ", "six "):::("( $N 7 ) ", "seven "):::("( $N 8 ) ", "eight "):: ("( $N 9 ) ", "nine "):::[]));
```

Generation of Efficient Compilers

by Application of

Single-Threading, Control Binding and Lambda-Lifting Techniques

by

Kok Hui Chong

B.A., Coe College, Cedar Rapids, IA, 1987

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1989

Abstract

The semantics definition of a language can be used to generate an error-free compiler. A drawback with the early work in automated compiler generation is that the generated compilers ran slower than the handwritten ones. Clues presented by the domains and valuation functions in the semantic definitions can be used to transform a denotational definition of a programming language into a more efficient form. Among the techniques which improve the efficiency of the generated compiler are **Single-Threading**, **Control Binding**, and **Lambda-Lifting**.

The motivation of this research is to tie together these techniques in the right order to maximize their effectiveness. We designed and implemented a compiler generator system which enables us to intermix these techniques in any order. Virtually any denotational definition can be implemented by the system. The output from the system together with a compile-time evaluator form a correct and efficient compiler.

As a result of testing the system with the semantics definition of a typical imperative language, we concluded the best results are obtained by applying **Single-Threading** first, **Control Binding** second and **Lambda-Lifting** last.