8424

Fully Lazy Evaluation of Functional Programs

Masato Takeichi

March 1987

Acknowlegements

I am deeply indebted to Eiiti Wada, my thesis supervisor. Without his constant encouragement this thesis would never have been written.

I am also very grateful to Sigeiti Moriguti for his invaluable advice since I got into computer science.

Kohei Noshita has influenced my research profoundly.

I prepared this thesis when I worked at the University of Electro-Communications (Denkitsuushin Daigaku). The Computer Science department provided a superb academic environment for research and education. I want to thank all the colleagues and students at UEC.

Finally I am very grateful to my wife, Shigeko, and my children, Norito and Fumi.

Table of Contents

Chapter 1

Introduction

One of the most important theme of computer science is to reduce software costs for developing correct programs. Honest programmers will do their best to write both correct and efficient programs in a way based on some programming principle. Research on programming methodology has brought many guiding ideas in conventional programming languages. For example, so-called *structured programming* in the late sixties has seriously affected our traditional style of programming practices. The reduction in software costs has not been successfully achieved, however.

A growing interest in functional programming as a practical approach to software development has been attached to a new departure for a radically different methodology. According to a functional approach, we can write clean initial programs in a powerful and elegant programming language with many sophisticated features. Such programs can be transformed into more efficient ones by applying simple mathematical rules, since purely functional programming avoids any side effects of which presence makes transformation a complex task. Program transformation in general aims for obtaining an efficient program that preserves correctness and requires less computing resources than the original. Its object is how to attain the ideal that only the necessary computation is to be taken and no other computaion be taken at all. Full laziness is a new approach to this ideal.

Section 1.1 demonstrates how functional style increases clarity of programs. Section 1.2 describes the basic idea of full laziness in a general setting and claims its optimality. Section 1.3 gives an outline of the rest of the thesis.

1.1. Functional programming is effective

Functional programming has a long history. Important early work includes two languages, McCarthy's Lisp [McCarthy62] and Landin's ISWIM [Landin66]. Church's lambda calculus [Church41] provides the mathematical background to most of the functional language including Lisp and ISWIM. There has been a good deal of argument, however, on whether Lisp is a functional language or not. For practical reasons, it contains impure features for functional programming as well as the clean mathematical

form of recursion equations. In fact, almost all serious programs written in the dialects of Lisp turn out to be of imperative style as in traditional procedural languages like Algol, Fortran, and Pascal. It is due solely, or at any rate mainly, to the conviction that assignment and goto statements are indispensable to get adequate performance from the machine. There is, however, a serious drawback in the use of side effects caused by these statements. Such referentially opaque features in procedural languages make programs incomprehensible and error-prone.

Recent study on functional programming shows that many sophisticated features such as local recursion, lazy evaluation, and partial parametrization open up a new programming style that has not been observed in traditional programming methodology. Moreover, purely functional programming becomes practical by the efficient compiler and by the high power of recent computers. We shall discuss such novel features in Chapter 2. This section gives a brief overview of functional programming that brings us the conviction of its effectiveness.

Programming at a higher level

We note first that functional programming allows one to specify problems in a highly abstract way. We can write programs with the whole picture in mind in functional programming, while every computational step has to be specified in procedural, or imperative, programming.

Consider the problem of finding Ramanujan's numbers as a non-trivial example; to find the numbers that are equal to two different sums of two natural numbers raised to the third power. A Pascal program to find the least Ramanujan's number is developed in [Wirth73] by the method of stepwise refinement. Along with control structures and data structures, housekeeping variables for recording the state of computation are introduced in the course of refinement. The final program developed is shown in Figure 1.1.1.

A functional solution to this problem looks like Figure 1.1.2.[1] This simple program will print pairs of two numbers indefinitely, or rather until it runs out of space, producing the output

[1] Strictly speaking the problem requires the number 1729, not the pair ((9,10),(1,12)), for example. Section 2.1 illustrates how this program is obtained from the specification of the problem. The language used here will be explained in Chapter 8.

```
var i, il, ih, min, a, b, k : integer;
  j, p, S : array [1..12] of integer;
begin i := 1; il := 1; ih := 2;
  j[1] := 1; p[1] := 1; S[1] := 2; j[2] := 1; p[2] := 8; S[2] := 9;
  repeat min := S[i]; a := i; b := j[i];
    if j[i] = i then il := il+1 else
      begin ih := ih+1; p[ih] := ih*ih*ih;
        j[ih] := 1; S[ih] := p[ih]+1
      end;
    i := il; k := i;
    while k < ih do
      begin k := k+1;
        if S[k] < S[i] then i := k
      end
  until S[i] = min;
  writeln( min, a, b, i, j[i] )
end.
```

Figure 1.1.1. A Pascal program for finding the least Ramanujan's number

```
ram (sort_r 1)
  whererec {
      ram (x:(y:z)) =
        if sumcubes x == sumcubes y then (x,y) : ram(y:z) else ram(y:z)
    and
      sort_r k = (k,k) : merge_r [(k,b) | b <-[k+1..] ] (sort_r (k+1))
    and
      merge_r (x:u) (y:v) =
        if sumcubes x <= sumcubes y then x : merge_r u (y:v) else y : merge_r (x:u) v
    and
      sumcubes (a,b) = a*a*a+b*b*b
  }
```

Figure 1.1.2. A functional program for finding Ramanujan's numbers

((9,10),(1,12)) ((9,15),(2,16)) ((18,20),(2,24)) ((19,24),(10,27)) ((18,30),(4,32)) ((15,33),(2,34))

((16,33),(9,34)) ((27,30),(3,36)) ((26,36),(17,39)) ((31,33),(12,40)) ···

The least Ramanujan's number is $9^3+10^3=1^3+12^3=1729$. The expression *ram* (*sort_r* 1) of the functional program represents an infinite list of the pairs satisfying the condition. Setting aside the details, it is observed that the functional program is an executable specification of the problem rather than a program expressing the computational process a step at a time as is typical in procedural programming. The program should be *correct* as far as the problem is formulated this way. There are, of course, many problems

of which specifications are non-constructive, e.g., axiomatic. In these cases we have to rewrite them in terms of equations and functions. Functional programs thus developed are easy to verify because proofs can be based on the well-understood concept of functions rather on the more cumbersome notion of conventional computers.

Another important point related to programming methodology is that we can write a program by first developing the most general solution and then making it specific. If we are required to find the least Ramanujan's number, we have only to make a composition $first\,(ram\,(sort\_r\ 1))$ where $first$ is the function that takes the first element of the list. This is contrast to the traditional style of programming.

High productivity

A number of studies have shown that a programmer produces a roughly fixed number of lines of code independent of the language used. The reason why a high level procedural language, e.g., Fortran, Pascal, etc., has made a step forward in software productivity is that programs written in such a language are an order of magnitude shorter than the equivalent assembly code. The significance of the fact is that the most important factor in software production costs is the level of language used in programming.

The compact notation for functional programs allows more algorithms to be expressed per line than procedural languages because the details of implementation are not specified in functional programming. A functional language would increase productivity much the same way as a high level procedural language does compared to an assembly language. Several experiments for sizable problems have been reported [Morris80, Turner81b, Peyton-Jones85] to support this view.

Efficiency problems

Although there is evidence that functional programming increases productivity, functional programs have gained an unfavorable reputation for running slowly. Major sources of the inefficiency pointed out are

- The high frequency of function calls, which results in many operations for parameter passing,
- The overhead for garbage collection,
- The lack of destructive updating of large data structures,

- The performing of the same computaion repeatedly.

Although some of them reflect the fact that functional languages are not so close to conventional computers as are procedural languages, others do merely that many implementations have been inefficient. It is worth noting that functional programs have typically been run interpretively rather than compiled. The reason is that implementation techniques for functional languages have received too little research effort compared to theoretical work. Development of compiling techniques will greatly improve the efficiency.

Evaluation method may also increase the efficiency of functional programs. For example, the use of *lazy evaluation* can lead to substantial improvement in manipulating large data structures by reducing the amount of unnecessary computation. Another important approach to the efficiency problem is program transformation. Mathematically concise expressions can be transformed into more efficient ones that use less computational resources. Lazy evaluation combined with program transformation well deserves attention from the point of implementing functional languages. We shall discuss this in the next section.

1.2. Full laziness is optimal

Functional languages are free of side effects. A programmer is never concerned about unexpected modifications to variables by other routines. Consequently the parameter passing mechanisms *call-by-value* and *call-by-name* have the same effect, provided the computation terminates[2]. In the call-by-value discipline, every argument must first be evaluated and a copy of the result then be passed to the function. In the call-by-name discipline, the argument is evaluated at each occurrence of the corresponding parameter within the function. *Evaluation* in this context roughly means the process of *simplification* of the form of expression, or *reduction*; 6 is simpler than 2* 3 by our criterion. Evaluation mechanisms greatly affect the efficiency of functional programs. This section provides the basic idea of *full laziness* with relation to parameter passing mechanisms.

Parameter mechanisms

To illustrate the differences between the parameter mechanisms mentioned above, we first introduce

[2] This fact is closely related to the Church-Rosser theorem of lambda calculus [Church41]. The call-by-name mechanism actually possesses better terminating properties.

some notations to denote expressions. We write functional application by juxtaposition;

$$f\ a$$

means that the function f is applied to the argument a. We usually omit the parentheses around arguments. A function definition is written as

$$f\ x\ y\ =\ x+y\ .$$

The function f is a higher order function which take their arguments *one at a time.*

Consider an expression

$$f\ (2*3)\ 1$$

which uses the function f defined as above. In the call-by-value mechanism, the arguments (2* 3) and 1 are evaluated and the results 6 and 1 are passed to the function f. The function then evaluates (6+1) to yield 7. On the other hand, the arguments (2* 3) and 1 are passed as they are in the call-by-name mechanism. The evaluation of the function body $x+y$ demands the evaluation of the arguments when the values of x and y are needed as operands of '+'. The difference of the parameter mechanisms just observed is the time when arguments are evaluated.

Given are a definition[3]

$$g\ x\ y\ =\ \textbf{if}\ y{=}{=}1\ \textbf{then}\ 0\ \textbf{else}\ x$$

and an expression

$$g\ (2*3)\ 1\ ,$$

the argument (2* 3) is evaluated when g is called by value, while its value is not actually needed. It is not the case if the call-by-name mechanism is used. This example demonstrates a characteristic feature of call-by-name; arguments of a function are evaluated in a *demand-driven* fashion. Call-by-name seems better than call-by-value at first sight from this property. However, if a function is defined as

$$h\ x\ y\ =\ x+x$$

and the value of

$$h\ \alpha\ \beta$$

is required, the simpleminded call-by-name mechanism demands the evaluation of the argument α twice for

[3] We use '==' to mean the equality; $y{=}{=}1$ means a predicate 'is y equal to 1 ?'.

each of two occurrences of x in the function body. It is very costly if the argument α is of the complex form. In contrast to this, the argument α is evaluated only once when the function is called by value. Call-by-value is desirable in this case. This is the major reason why modern procedure languages adopt call-by-value as their standard parameter passing mechanism and exclude parameters called by name.

Call-by-need and lazy evaluation

As described above, the call-by-name mechanism has both merits and demerits in evaluating arguments. In procedural languages, evaluation of the argument α possibly causes side effects and may yield different values each time evaluation takes place. However, there are no worries about side effects in functional languages. The property of *referential transparency* serves for introducing a new device of parameter passing. Referential transparency implies that the value of an expression depends only on its textual context, but not on computational history. In functional languages, therefore, the result of evaluation of α at the first time may be used at later time its value is needed. In general, function arguments are passed by name, and each of them is evaluated in a demand-driven way the first time its value is needed. Subsequent references to the argument use the value already evaluated. Such a parameter passing discipline is known as *call-by-need* [Wadsworth71] or *call-by-delayed-value* [Vuillemin74]. The evaluation mechanism of expressions based on call-by-need is called *lazy evaluation* [Henderson76]. Lazy evaluation is sometimes worded as

Never do today what you can put off until tomorrow,

but we also use the result that has been obtained, if any. Call-by-name and call-by-need are semantically equivalent, so it does not matter which is used. But call-by-need is much more efficient than call-by-name in practice.

Fully lazy evaluation

We may say that lazy evaluation has both call-by-name and call-by-value properties; every argument of a function is evaluated *at most once*. It is superior to either of call-by-name or call-by-value mechanism because there may be multiple evaluation stages of an argument in call-by-name and every argument is always evaluated *exactly once* in call-by-value.

Fully lazy evaluation goes one step further in evaluating functional expressions. Suppose that a function f is defined as

$$f\ x\ y\ =\ (fac\ x)+y$$

where fac is the factorial function defined elsewhere. Given is an expression with a local definition

$$(g\ 3)+(g\ 4)\ \textbf{where}\ g=f\ 5\ ,$$

several evaluation strategies may be considered[4]. In any lazy evaluation scheme, the expression $(f\ 5)$ in the local definition is not evaluated until either of $(g\ 3)$ or $(g\ 4)$ becomes to be evaluated. We assume here that $(g\ 3)$ is evaluated first. Then, $(f\ 5\ 3)$ is evaluated to obtain the value of $(g\ 3)$. This in turn cause $(fac\ 5)$+3 to be evaluated yielding the value 123. The evaluation step is schematically shown as

$$(g\ 3)\ \rightarrow\ (f\ 5\ 3)\ \rightarrow\ (fac\ 5)+3\ \rightarrow\ 120+3\ \rightarrow\ 123\ .$$

Similarly evaluation of $(g\ 4)$ is

$$(g\ 4)\ \rightarrow\ (f\ 5\ 4)\ \rightarrow\ (fac\ 5)+4\ \rightarrow\ 120+4\ \rightarrow\ 124\ .$$

As we can see from these evaluation steps, it is obvious that the second evaluation of $(fac\ 5)$ is redundant because its value 120 has been obtained at the first time its value is needed. Ordinary lazy evaluation does not concern this situation.

Evaluation by an alternative strategy may be considered. It proceeds similarly as above, but the expression $(fac\ 5)$ becomes 120 once it has been evaluated. Accordingly the definition

$$g=f\ 5$$

becomes[5]

$$g\ y=120+y\ .$$

Then the expression $(g\ 4)$ is evaluated as

$$(g\ 4)\ \rightarrow\ 120+4\ \rightarrow\ 124\ .$$

We can say in a general setting that

Every expression is evaluated at most once after the variables in it have been bound.

---

[4] We use **where**-clauses to introduce local definitions. The meaning of an expression with a **where**-clause would be obvious. It is similar to the usage in mathematics.

[5] Replacement of expressions is, of course, conceptual, and requires some devices to be implemented. We shall deal with them in later chapters.

Such evaluation mechanism is called *fully lazy evaluation* [Hughes84].

Fully lazy evaluation is very similar to well-known optimization techniques such as *constant folding* and *moving invariants from loops* in procedural languages [Aho77]. Although optimizing compilers perform these tasks at compile time, fully lazy evaluation does at execution time. Hence full laziness subsumes *dynamic compilation* or *self-optimizing evaluation*. We may summarize our conclusion as

Fully lazy evaluation is optimal

in the sense that it performs no redundant computation.

1.3. Outline of the thesis

Chapter 2 describes some characteristic features of functional programming with emphasis on the use of *higher order functions* and *partial parametrization* in practical programming.

Chapter 3 presents the concept of *laziness* and *full laziness* in evaluation of functional programs.

Chapter 4 describes several evaluators based on *combinator reduction*. Experimental results are also included[6].

Chapter 5 gives the definition of the *fully lazy normal form* which is the basis of the translation and evaluation system described in Chapters 6, 7 and 8[7].

Chapter 6 presents a program transformation technique called *lambda-hoisting* for fully lazy evaluation of functional programs[8].

Chapter 7 describes an abstract machine called *fully lazy functional machine* which is used as a target machine of portable compilers in Chapter 8[9].

Chapter 8 presents a method of implementing portable compilers for functional languages. Actual implementations of an experimental language on several computers are described with experimental results[10].

---

[6] The contents of Chapter 4 is partly described in [Takeichi84] and [Takeichi85].

[7] Related work by the author in a different context is found in [Takeichi82a,82b]

[8] An extended abstract of Chapter 6 has been presented in [Takeichi86b].

[9] An earlier version of the abstract machine is reported in [Takeichi86a].

[10] The author's experience with portable compilers can be found in [Takeichi77].

Chapter 9 illustrates how higher order functions and partial parametrization bring unexpected gains in efficiency[11].

Chapter 10 deals with the topic of types in functional specifications. An application of the *polymorphic type* discipline to the problem of inserting injection operations to denotational specifications is described[12].

Chapter 11 summarizes the contributions of the thesis and discusses directions for future work.

The appendices provide an example of program translation described in Chapters 6-8, the syntax definition of the language *uc* in Chapter 8, and an example of typechecking in Chapter 10.

---

[11] The major part of Chapter 9 has been published as [Takeichi87].

[12] The contents of Chapter 10 has been published as [Takeichi86c].

Chapter 2

Overview of functional programming

This chapter gives a brief overview of characteristic features in functional programming. We first deal with an example of program development with a view to illustrating the theme proposed by Turner

Functional programs as executable specifications

in [Turner84]. We also discuss the use of *higher order functions* and *partial parametrization* for practical purposes. This may lead to a novel programming methodology since these features provide a particularly powerful abstraction mechanism that is not found in traditional languages. Finally we discuss work on program transformation of functional programs relevant to this thesis.

2.1. Program development

Consider the problem of Ramanujan's number presented in Chapter 1:

> Find the least number that is equal to two different sums of two natural numbers raised to the third power.

A functional program has been shown in Figure 1.1.2 for comparison with a Pascal solution. We here illustrate how we can arrive at the executable solution from a specification of the program at a higher level. Those wishing to know about development of the Pascal program should consult Section 15.2 of [Wirth73].

We have to find the least number x such that

$$x = a^3 + b^3 = c^3 + d^3$$

where a, b, and c are natural numbers such that $a \neq c$ and $a \neq d$. If we can express the set R of pairs $((a,b),(c,d))$ of two pairs (a,b) and (c,d) of natural numbers that satisfy the above equation, the problem is solved immediately by selecting the pair from R that gives the least Ramanujan's number. Hence we set up a subproblem of finding the set R.

The set R can be obtained by arranging possible pairs (a,b) in an increasing order of the sum of cubes $sumcubes(a,b)$ where[1]

[1] We here assume that the function *sumcubes* takes a pair of natural numbers as the form of the parameter (a,b) indicates. It is natural that the function has a single parameter in this case, while we usually use the *curried* form for functions with more than two arguments.

$sumcubes\ (a,b) = a^3 + b^3$,

and combining adjacent pairs (a,b) and (c,d) of the sorted sequence S that are equal to each other with respect to the valuation function *sumcubes*, i.e., $sumcubes\,(a,b) = sumcubes\,(c,d)$. We now arrive at the first specification.

```
R ≡ ram S
      whererec
          ram (x:(y:z)) =
                if sumcubes x == sumcubes y then (x,y) : ram (y:z) else ram (y:z)
```

We use **whererec**-clauses to introduce local recursive definitions; the function *ram* is defined recursively. We write '$a:x$' to represent a sequence, or *list* in programming language terminology, that is formed by prefixing an element a at the front of x. Function parameters may be specified by patterns as above. The function *ram* takes a sequence of which first element is denoted by x, the second element by y, and the rest by z in the right-hand side of the definition.

The next problem is how to specify the sorted sequence S. Let

$[\alpha\ ..\ ]$

denote a set of all natural numbers greater than or equal to α. Then a set

$X \equiv \{\ (a,b)\ |\ a \leftarrow [\ 1\ ..\ ];\ b \leftarrow [\ a\ ..\ ]\ \}$

contains all of the pairs that should be taken into account. We use here the symbol '$\leftarrow$' instead of usual mathematical symbol '$\in$'. The symmetry of the valuation function $sumcubes\,(a,b)$ allows for limiting the range of b as $b \geq a$. This limitation is indeed necessary because otherwise pairs $((a,b),(b,a))$ would be included in R. Using the notation we get a definition of S:

$S \equiv sort\ sumcubes\ \{\ (a,b)\ |\ a \leftarrow [\ 1\ ..\ ];\ b \leftarrow [\ a\ ..\ ]\ \}$

where

$sort\ r\ X$

represents a sorted sequence of which elements are taken from an infinite set X and arranged in order with respect to a valuation function r. Another question may arise: is the function *sort* computable? There cannot exist a recursive definition of *sort* on infinite sets. However, we can derive a recursive definition of *sort* after the analogy of Turner's examples [Turner84].

For any $k \geq 1$, let

$$X_k = \bigcup_{a=k}^{\infty} \{ (a,b) \mid b \leftarrow [a \,..\,] \},$$

and

sort r k

be a sorted sequence of elements of X_k. Notice that the function *sort* has been redefined; the second parameter k should have been X_k for the older definition. We rewrite S as

$$S = sort\ sumcubes\ 1\ .$$

By separating the set X_k into two parts as shown in Figure 2.1.1,

$$X_k = \{ (k,b) \mid b \leftarrow [k \,..\,] \} \cup X_{k+1}$$

as shown in Figure 2.1.1, we may merge two sorted sequences of the disjoint sets to arrange the elements of X_k in order.

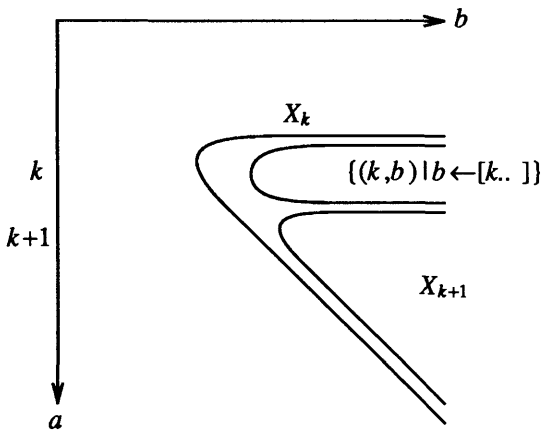


Figure 2.1.1. X_k composed of two sets

We may write the definition of *sort* as

```
sort r k = merge r (sort r {(k,b) | b←[k.. ]}) (sort r (k+1))
    where
        merge r (x:u) (y:v) =
            if r x ≤r y then x : merge r u (y:v) else y : merge r (x:u) v .
```

The function *merge* maps two sorted sequences $(x:u)$ and $(y:v)$ into a single sorted sequence. We have thus derived a recursive definition of the function *sort* with an auxiliary function *merge*.

The definition of R along with that of the function *sort* may be considered as a specification of the subproblem. It tells *what* are Ramanujan's numbers, but does not *how* these are computed. The specification is not executable, however. We have used a notation for representation of infinite objects that was devised by Turner. We can compute with such an infinite object so far as we know that as much of its elements as is required in a particular computation is obtained in finite steps. Computation of infinite objects can be realized by lazy evaluation mentioned in Chapter 1. Of course we must take care not to compute all of the elements of an infinite object. The above specification is written in a functional style and is really a 'program' of our functional language used in this thesis (See Chapter 8 for details). But it is undesirable for our purposes. It contains infinite computation that does never terminate unless we adopt some intricate evaluation mechanism other than ordinary lazy or fully lazy evaluation mechanisms.

In order to get a specification appropriate for execution we rewrite the right-hand side of the definition of *sort*. Note that we take here the function *sumcubes* as r, and that *sumcubes* is a monotonically increasing function with respect to components of the argument. We may give a more compact expression for the second argument of *merge*:

$$sort\ r\ \{\ (k,b)\ |\ b \leftarrow [\ k\ ..\ ]\ \}\ =\ [\ (k,b)\ |\ b \leftarrow [\ k\ ..\ ]\ ]$$

provided that r is monotonic. The notation used in the right-hand side represents a sequence of which elements are arranged in order as

$$(k,k) : (k,k+1) : (k,k+2)\ \cdots$$

that is, the elements are 'generated' as b takes values $k, k+1, k+2, \cdots$ to infinity. From this and the fact that any element of $(sort\ r\ (k+1))$ does not precede (k,k) in the sorted sequence $(sort\ r\ k)$, we get a new definition of *sort*:

$$sort\ r\ k\ =\ (k,k) : merge\ r\ [\ (k,b)\ |\ b \leftarrow [k+1\ ..\ ]\ ]\ (sort\ r\ (k+1))\ .$$

This finishes our example of program development. We have thus obtained the program for the subproblem as shown in Figure 2.1.2. The program presented here is slightly different from that of Figure 1.1.2. The functions *sort* and *merge* are specialized and called *sort_r* and *merge_r* in Figure 1.1.2 so that they use the function *sumcubes* instead of the given valuation function r. Rewriting this way seems to serve for reduction of computational steps for passing *sumcubes* each call of the functions. It is possible, however,

```
ram (sort sumcubes 1)
  whererec {
      ram (x:(y:z)) =
        if sumcubes x == sumcubes y then (x,y) : ram(y:z) else ram(y:z)
    and
      sort r k = (k,k) : merge r [(k,b) | b <-[k+1..] ] (sort r (k+1))
    and
      merge r (x:u) (y:v) =
        if r x <= r y then x : merge r u (y:v) else y : merge r (x:u) v
    and
      sumcubes (a,b) = a*a*a+b*b*b
  }
```

Figure 2.1.2. A functional program for finding Ramanujan's numbers

to evaluate the program of Figure 2.1.2 in a way to produce the same effect. Such an evaluation mechanism is the major theme of this thesis.

What we learn from the example are:

- We may introduce mathematical notations suitable for specifying the problem, e.g., sets and sequences.
- We have to write relations between objects in terms of equations.
- A specification might not be executable as it is, but it may be transformed into executable one by clear mathematical reasoning.

Of course, program development depends on the language used. It will be clear, however, that mathematical notations and function definitions of our *ad hoc* language can be translated into such expressions that any functional language accepts. We shall demonstrate in Chapter 8 how sets and sequences are represented by functions and fundamental data structures. Concrete representation of abstract objects is thus hidden in our language in the sense of abstract data types. Another abstraction mechanism is brought up by the use of higher order functions as described in the next section.

2.2. Higher order functions and partial parametrization

The program in the previous section includes functions that take functions as their arguments; the function *sort* and *merge* both have functional parameter *r* which is to be applied to elements of sequences.

A functional parameter is one of the common abstraction mechanisms existent in both functional and procedural languages. Pascal, for example, allows us to write functions (and procedures) that take functions (or procedures) as arguments. But such a parameter mechanism in procedural languages seems to be a treasure useless to practical programmers. The use of functional or procedural parameters in traditional programming is very rare. One of the causes for little use should be sought in restrictions on procedures or functions as values. A Pascal function (and procedure) comes into existence only by its declaration. There is no other way to produce functions and procedures by programs.

In functional languages we are dealing with allows us to write functions that take functions as arguments and return functions as results. Functions can be made elements of data structures like lists, pairs, or trees. Thus, the function in functional languages is a 'first class object'.

As an illustration we consider a problem of implementing a simple table.

- The table initially contains no entry.
- An entry is entered in the table so that it will be looked up with a unique key.
- The table is searched with a key for the entry to be sought.

We assume here that each key is chosen from a set α, and each table entry from a set β. We may consider that the table has the type[2]

$$\alpha \rightarrow \beta .$$

That is, a particular configuration of entries is represented by a function.

The initial configuration of the table should be empty:

$$init\_table \; = \; t \;\; \textbf{where} \;\; t \; x = undef$$

where $undef \in \beta$ is a distinguished element. The function *init_table* returns *undef* for any element in α.

A table t is extended with $a \in \alpha$ and $b \in \beta$ as

$$extend \; t \; (a,b) \; = \; t' \;\; \textbf{where} \;\; t' \; x = \textbf{if} \; x == a \; \textbf{then} \; b \; \textbf{else} \; t \; x$$

and t is looked up using $x \in \alpha$ as

---

[2] The term 'type' roughly means sets in this context. $\alpha \rightarrow \beta$ represents a set of functions of which source (domain) is α and target (range) β.

$$lookup\ t\ x\ =\ t\ x\ .$$

We can convince ourselves relations such as

$$lookup\ (init\_table)\ x\ =\ undef \qquad \text{for any } x \in \alpha,$$

and

$$lookup\ (extend\ t\ (a,b))\ x\ =\ b\ \text{ if } x=a\text{, and} = lookup\ t\ x\ \text{ otherwise.}$$

In this implementation, functions are passed as arguments and returned as results.

Another representation method of the table is to use a list structure of which elements are pairs of type $\alpha \times \beta$. This is a rather standard way of implementation in procedural programming. An initialized table corresponds to an empty list

$$init\_table\ =\ [\ ]\ .$$

A new entry (a,b) is entered in the table t as

$$extend\ t\ (a,b)\ =\ (a,b) : t\ .$$

We may look up such a table t as

$$\begin{aligned} lookup\ t\ x\ &=\ lookup'\ t\ x \\ &\quad \textbf{whererec} \\ &\qquad lookup'\ t'\ x\ =\ \textbf{if } t'=[\ ] \textbf{ then } undef \textbf{ else} \\ &\qquad\qquad (\ \textbf{if } x=a \textbf{ then } b \textbf{ else } lookup'\ u\ x \\ &\qquad\qquad\qquad \textbf{where } ((a,b):u)=t'\ ) \end{aligned}$$

In fact, it is most desirable that the *table* type is defined as an abstract type; only *init_table*, *extend*, and *lookup* should be made available to the user. Some functional languages provide the facilities to define abstract types [Milner84, Turner85]. The function *lookup* for tables implemented by lists seems somewhat 'procedural' while it is really pure functional.

In either implementation of the table, the function *lookup* has two parameters t and x, and therefore called second-order. The type, or *functionality*, is

$$lookup\ :\ \gamma \rightarrow [\ \alpha \rightarrow \beta\ ]$$

where γ is the type of the table implemented as above in either

$$\gamma\ =\ \alpha \rightarrow \beta$$

or[3]

[3] 'α *list*' denotes the type of lists of which elements are of type α.

$$\gamma = (\alpha \times \beta)\ list\ .$$

In general, functions may have arbitrary numbers of parameters. If a function is defined to have n arguments, it can always be applied to $m \leq n$ arguments. The result is a function that may take $(n-m)$ arguments, if $m < n$, in which the first m arguments have been fixed.

Suppose that we are required to search a table, say T, frequently with different keys x. We like to have a function *lookupT* that does specifically look up the table T and never does other tables. The function *lookupT* is easily obtained by instantiating the function *lookup* with the first argument T:

$$lookupT \equiv lookup\ \ T\ .$$

The resultant function takes a single argument x. We call such a discipline *partial parametrization.*

Higher order functions and partial parametrization provide a powerful kind of abstraction. Many similar functions can be defined by parametrizing a higher order function that represents a common pattern of computation, or an abstract *algorithm.* The advantage of programming this way is that modularity can be achieved by a simple notion of functional abstraction and application. The use of these features in practical programming is extremely important. Programs become more compact and easier to read. We shall discuss it in more detail in Chapter 9.

2.3. Program transformation

Program transformation has been studied by many researchers in many contexts. Compiler optimization and program synthesis are such examples. This section describes some transformation techniques relevant to this thesis.

All of the transformation techniques of functional programs enjoy the beneficial property of *referential transparency.* We have already transformed a specification into an executable program in Section 2.1.

The *unfold-fold* transformation method is one of the most well-known techniques. It was developed in [Burstall77]. The method is based on two transformation rules, *unfolding* and *folding*. The aim of this technique is to improve the efficiency of programs written in a functional language without changing their meaning. The transformation rules consist of

- Definition: Introduce a new equation $e=E$. The left-hand side e is not an instance of that of any pre-

vious equation.

- Instantiation: Introduce an instance of an equation by substitution.
- Unfolding: If $e=E$ and $f=F$ are equations and there is some occurrence in F an instance e' of e, replace e' by the corresponding instance E' of E getting $F[E'/e']$. Then add an equation $f=F[E'/e']$.
- Folding: If $e=E$ and $f=F$ are equations and there is some occurrence in F an instance E' of E, replace E' by the corresponding instance e' of e getting $F[e'/E']$. Then add an equation $f=F[e'/E']$.
- Abstraction: Introduce a **where**-clause by deriving from an equation $e=E$ a new equation

  $e = E[x_1/E_1, \cdots, x_n/E_n]$ **where** $x_1=E_1$ **and** $\cdots$ **and** $x_n=E_n$

  where x_i are new variables and E_i are subexpressions. The right-hand side is an expression obtained by replacing all the occurrences of E_i with x_i.
- Algebraic laws: Transform an equation by using laws about the primitives such as associativity, commutativity.

Developing a sequence of transformation steps based on these rules requires some heuristics. Bird [Bird84] proposed a method using the unfold-fold transformation technique to transform programs that traverse data structure many times into ones that do only once. In Chapter 9 of this thesis we shall give a brief overview of his method and present a more versatile transformation method based on higher order functions and partial parametrization.

Wadler [Wadler85] describes a transformation technique which can be applied to a restricted class of programs and claims that *listlessness is better than laziness*. We shall discuss laziness for executing any functional programs from a different viewpoint in this thesis.

Another kind of transformation techniques related to this thesis is for implementation of functional languages. Turner [Turner79] proposed a novel method for evaluation of functional programs called *combinator reduction*. The transformation technique used in this context is to 'compile' functional programs into *combinator code*. Simple transformation rules will be shown in Chapter 4. Several alternatives have been developed in [Noshita85a,85b]. Hughes' work is the idea of *super-combinators* and the transforma-

tion algorithm [Hughes84], which will be also discussed in Chapter 4. Lambda-lifting by [Johnsson85] is a transformation method for compiling programs to generate efficient code. It is similar to our *lambda-hoisting* transformation in Chapter 6.

Finally we would like to describe an experimental result from which we becomes convinced of the practical importance of program transformation. Turner [Turner82] gives a program for the 8-queens problem as shown in Figure 2.3.1[4].

```
queens 8
  where rec {
      queens n =
        if n == 0 then [[]]
          else [b++[q] | q <- [1..8]; b <- queens (n-1); safe q b ]
    and
      safe q b = foldr (&&) true [!(checks q b i) | i ← [1..length b]]
    and
      checks q b i = ((q==bi) || abs(q-bi)==length b-i+1) where bi=nth i b
  }
```

Library functions: *length*, *nth*, *foldr*, *abs*

Figure 2.3.1. A functional program for the 8-queens problem

We used a compiler described in Chapter 8 to translate the program of which number of the queens had been scaled down to 5. That compiler makes use of a transformation technique for full laziness (See Chapter 3) by default, and does not optionally. Two object programs compiled with or without the transformation runs 2.85 seconds and 43.7 seconds, respectively to print the results. The effect of transformation is remarkable, while we cannot find the cause only from a higher level specification in Figure 2.3.1. We can say that

> *The higher the level of specifications or languages, the more important the transformation technique for implementation.*

[4] The program is written in *uc*. See Section 8.3 and Appendix B.

Chapter 3

Laziness and full laziness

This chapter describes evaluation mechanisms in a formal way to make the differences between them become clear. We deals with only *lazy* evaluators; an ordinary lazy evaluator proposed so far is described in Section 3.1, and a *fully lazy* evaluator is defined in Section 3.2. A lazy evaluator, whether it is an ordinary one or a fully one, aims at practical utility of the following ideas [Henderson76]:

- Perform computation only when it is necessary.
- Never perform the same computational steps twice.

These intuitive statements need to be examined formally.

Section 3.3 gives some remarks on implementation of evaluators which serve for the basis of efficient evaluators described in Chapters 4-8.

3.1. Lazy evaluation

In this section we shall describe the definition of an ordinary lazy evaluator. We use a simple functional language specified in Figure 3.1.1[1]. Although the language is very simple, fundamental features of functional languages are included; functional applications by combination, and abstractions by **fn**-expressions[2]. Local definitions by **where**-clauses are excluded for simplifying the evaluator. Expressions with **where**-clauses like

$$e_0 \textbf{ where } x_1 = e_1 \textbf{ and } \cdots \textbf{ and } x_n = e_n$$

can be considered as a syntactic sugared form for

$$(\textbf{fn } x_1. \cdots \textbf{fn } x_n. e_0)\, e_1 \cdots e_n\,).$$

These are semantically equivalent. Recursive definitions may be expressed using the fixed point operator Y defined by $Y f = f\ (Y f)$ and **where**-clauses. We shall deal with recursive definitions directly in later chapters, however. Excluded also are specifications of primitive functions such as arithmetic and Boolean operations, and data constructors and destructors. A more important function would be '*if*' for conditional

[1] The language shown in Figure 3.1.1 will be extended in Chapter 5 so that expressions with **where**- and **whererec**-clauses are allowed. The extended language will be used as our *referential language* throughout this thesis.

[2] We avoid to use λ in the source language because it is used in the description of the semantics. We use **fn** instead.

Syntactic domains

$b \in \mathbf{Bas}$	basic values
$x \in \mathbf{Ide}$	identifiers
$e \in \mathbf{Exp}$	expressions

Abstract syntax

$e ::= b \mid x \mid e\ e \mid \mathbf{fn}\ x.\ e$

Semantic domains

$\mathbf{B}$	basic values
$\mathbf{E} = [\ \mathbf{B} + \mathbf{F}\ ]_+$	expressible values
$\mathbf{F} = \mathbf{D} \rightarrow \mathbf{E}$	functions
$\mathbf{D} = \mathbf{E}$	denotable values
$\mathbf{U} = \mathbf{Ide} \rightarrow \mathbf{D}_+$	environments

Semantic functions

$B : \mathbf{Bas} \rightarrow \mathbf{B}$	(unspecified)
$E : \mathbf{Exp} \rightarrow \mathbf{U} \rightarrow \mathbf{E}$	

$E[b]\rho = B[b]$
$E[x]\rho = \rho[x]$
$E[e_0\ e_1]\rho = (E[e_0]\rho)(E[e_1]\rho)$
$E[\mathbf{fn}\ x.\ e_0]\rho = \lambda\delta.E[e_0](\rho + \langle x \rightarrow \delta \rangle)$

Notations

Domain construction operator + stands for the disjoint sum.
For any domain **X**, $\mathbf{X}_+ = \mathbf{X} + \{\mathbf{err}\}$.
For an environment ρ, $\rho + \langle x \rightarrow \delta \rangle$ denotes
$\lambda y.\mathbf{if}\ x = y\ \mathbf{then}\ \delta\ \mathbf{else}\ \rho[y]$.

Initial Environment

The initial environment ρ_0 satisfies $\rho_0[x] \neq \mathbf{err}$ for pre-defined identifiers x.

Figure 3.1.1. Denotational Specification of a Very Simple Functional Language

expressions. A conventional non-lazy evaluator needs to deal with conditionals as special forms because for an expression $(if\ e_1\ e_2\ e_3)$ either e_2 or e_3 must be selected according to the value of e_1 while all of the arguments are always evaluated before invocation of functions by such an evaluator. It is not the case, however, in our lazy evaluator. We may consider that in lazy evaluation 'if' as well as other basic operations are primitive functions. Each primitive function has its own evaluation rule, which is independent of the basic algorithm presented here.

We first define an ordinary lazy evaluator using the notation in [Henderson76]. We assume that the abstract memory **M** has an very accomodating property. Each cell of **M** is capable of holding any of the

forms $e \in \mathbf{Exp}$ in Figure 3.1.1. It is certainly unrealistic, but it greatly simplifies the discussion. We shall discuss about practical evaluators in Section 3.3. A memory cell at *address* π of **M** holds an expression. A simple expression b or x is held in the cell π in an obvious way. How about compound expressions like $(e_0\, e_1)$? Such an expression is represented using addresses of cells for component expressions. Figure 3.1.2 shows the rules for representation of expressions in the abstract memory **M**.

Syntactic domains

$b \in \mathbf{Bas}$	basic values
$x \in \mathbf{Ide}$	identifiers
$e \in \mathbf{Exp}$	expressions

Representation in **M**

$\varepsilon \in \mathbf{Rep}$	representation of expressions
$\pi \in \mathbf{Addr}$	addresses

$\varepsilon ::= b \mid x \mid \pi\,\pi \mid \mathbf{fn}\, x.\ \pi \mid \pi\ \mathbf{with}\ x{=}\pi\ \mathbf{and}\ \cdots\ \mathbf{and}\ x{=}\pi$

Representation function

$Rep : \mathbf{Exp} \rightarrow \mathbf{Rep}$

$Rep[b] = \lfloor b \rfloor$
$Rep[x] = \lfloor x \rfloor$
$Rep[e_0\, e_1] = \lfloor \pi_0\, \pi_1 \rfloor$ where $\pi_0 ::= Rep[e_0]$ and $\pi_1 ::= Rep[e_1]$
$Rep[\mathbf{fn}\, x.\, e_0] = \lfloor \mathbf{fn}\, x.\, \pi_0 \rfloor$ where $\pi_0 ::= Rep[e_0]$

Notations

A representation ε of an expression e is quoted by $\lfloor$ and $\rfloor$.
$\pi ::= \varepsilon$ means that the cell at address π contains ε.

Figure 3.1.2. Representation of expressions in the abstract memory **M**

The last form of ε, i.e.,

$$\lfloor\ \pi_0\ \mathbf{with}\ x_1{=}\pi_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n{=}\pi_n\ \rfloor$$

stands for a representation of a suspended expression which comes out in the course of evaluation. See below for details.

The state of a computation is described by a partial function μ

$$\mu : \mathbf{Mem}$$

where

$$\mathbf{Mem} = \mathbf{Addr} \rightarrow \mathbf{Rep}\ .$$

If an address π appears as a component of any representation of expression, $\mu\,\pi$ is defined. Addresses for which μ is undefined are the free cells. We use a notation $\mu+<\pi\rightarrow\varepsilon>$ to describe a new memory which differs from μ only at address π:

$$\mu+<\pi_0\rightarrow\varepsilon> = \lambda\pi.\ \textbf{if}\ \pi=\pi_0\ \textbf{then}\ \varepsilon\ \textbf{else}\ \mu\,\pi\ .$$

This notation is very similar to the $\rho+<x\rightarrow\delta>$ notation in Figure 3.1.1. We assume also that '+' in $\mu+<\pi\rightarrow\varepsilon>$ associates to the left. That is,

$$\mu + <\pi_1\rightarrow\varepsilon_1> + <\pi_2\rightarrow\varepsilon_2> \ = \ (\,\mu + <\pi_1\rightarrow\varepsilon_1>\,) + <\pi_2\rightarrow\varepsilon_2>$$

and the rightmost term represents the last change to the memory.

Evaluation starts with loading the memory **M** with the representation ε of an expression *e* to be evaluated. The result can be found in the memory location which originally contains ε. Our lazy evaluator *Eval* (π,μ) evaluates ε at location π of the memory status μ and returns a new memory μ':

$$\textit{Eval}\ :\ \textbf{Addr}\times\textbf{Mem}\rightarrow\textbf{Mem}\ .$$

We may have a new memory

$$\mu' = \textit{Eval}\,(\pi,\mu)\ \text{ where }\ \mu\,\pi\ =\ \varepsilon$$

with the result

$$\mu'\,\pi$$

The evaluator *Eval* is defined by cases of the form of expressions. We shall give some remarks in parentheses '{' and '}' for ease of understanding.

Lazy evaluator *Eval* (π,μ)

[1] Case $\mu\,\pi\ =\ \lfloor b \rfloor$

return μ ;

[2] Case $\mu\,\pi\ =\ \lfloor x \rfloor$

return μ ;

[3] Case $\mu\,\pi\ =\ \lfloor \pi_0\,\pi_1 \rfloor$

let $\mu_1 = \textit{Eval}\,(\pi_0,\mu)$;

if $\mu_1\ \pi_0 = \lfloor$ **fn** $x.\pi_2$ **with** $\theta \rfloor$ **then**

let $\mu_2 = \mu_1 + <\pi \rightarrow \lfloor \pi_2$ **with** $x=\pi_1$ **and** $\theta \rfloor >$;

return $Eval\,(\pi,\mu_2)$;

else error ;

{ For a combination $\lfloor \pi_0\ \pi_1 \rfloor$, first evaluate π_0, which must be a function. If so, the cell at location π is made to contain a new expression composed of the function body π_2 with an argument binding $x=\pi_1$. Then evaluate it with the altered memory μ_2. }

[4] Case $\mu\ \pi\ = \lfloor$ **fn** $x.\pi_0 \rfloor$

return μ ;

{ No more evaluation possible. }

[5] Case $\mu\ \pi\ = \lfloor \pi'$ **with** $\theta \rfloor$ where $\theta = \lfloor x_1=\pi_1$ **and** $\cdots$ **and** $x_n=\pi_n \rfloor$

[5-1] Case $\mu\ \pi'\ = \lfloor b \rfloor$

return $\mu + <\pi \rightarrow \lfloor b \rfloor >$;

{ Update the cell π with a simpler expression b . }

[5-2] Case $\mu\ \pi'\ = \lfloor x \rfloor$

if there is an x_i such that $x=x_i$, $(1 \le i \le n)$ **then**

choose smallest such i ;

let $\mu_1 = Eval\,(\pi_i,\mu)$;

return $\mu_1 + <\pi \rightarrow \mu_1\ \pi_i >$;

else error ;

{ Find an expression x denotes from the binding θ. Evaluate it and update the location π with the result. }

[5-3] Case $\mu\ \pi'\ = \lfloor \pi_0\ \pi_1 \rfloor$

let π_0' and π_1' be distinct free cells;

let $\mu_1 = \mu + <\pi \rightarrow \lfloor \pi_0' \pi_1' \rfloor > + <\pi_0' \rightarrow \lfloor \pi_0$ **with** $\theta \rfloor > + <\pi_1' \rightarrow \lfloor \pi_1$ **with** $\theta \rfloor >$;

return $Eval(\pi, \mu_1)$;

{ Distribute the binding θ to each components. New cells are required to hold the instances of expressions represented by π_0 and π_1 which may contain free variables. }

[5-4] Case $\mu\, \pi' = \lfloor$ **fn** $x.\pi_0 \rfloor$

return μ ;

{ No more evaluation done for $\lfloor$ **fn** $x.\pi_0$ **with** $\theta \rfloor$. }

[5-5] Case $\mu\, \pi' = \lfloor \pi_0'$ **with** $\theta' \rfloor$

let $\mu_1 = \mu + <\pi' \rightarrow \lfloor \pi_0'$ **with** θ' **and** $\theta \rfloor >$;

return $Eval(\pi, \mu_1)$;

{ Combine two nested bindings into one. }

To illustrate how the lazy evaluator works, we shall consider an expression

$$(g\ 3) + (g\ 4)\ \textbf{where}\ g = (f\ 5\ \textbf{where}\ f\ x\ y = (fac\ x) + y\ )$$

where fac is the factorial function. This expression has already been presented as an example in Section 1.2, where the differences of lazy and fully lazy evaluation mechanisms are explained. In the language of Figure 3.1.1, we must write the expression as

$$(\textbf{fn}\ g.(add\ (g\ 3)\ (g\ 4)))\ (\ (\textbf{fn}\ f.f\ 5)\ (\textbf{fn}\ x.\ \textbf{fn}\ y.\ (add\ (fac\ x)\ y))\ )$$

The functions add and fac are considered as primitives.

The function add eventually evaluates its two arguments $(g\ 3)$ and $(g\ 4)$. Both terms are evaluated by first computing g by the evaluator. Let us consider the computation of

$$(\ (\textbf{fn}\ f.f\ 5)\ (\textbf{fn}\ x.\ \textbf{fn}\ y.\ (add\ (fac\ x)\ y))\ )$$

which is assigned to g. This term is evaluated to become something like

$$\lfloor \pi_1 \textbf{ with } f = \pi_4 \rfloor .$$

We may write the initial configuration of the memory μ_0 as Figure 3.1.3(a)[3]. That is,

[3] For simplicity, we omit representation details for '$(add\ (fac\ x))$' held in the cell at π_7. Computation here will not proceed beyond this point.

$\mu_0\,\pi_0 = \lfloor\, \pi_1$ **with** $f = \pi_4 \,\rfloor$

which represents an expression with a **where**-clause

$$f\ 5\ \textbf{where}\ f = \textbf{fn}\ x.\ \textbf{fn}\ y.\ (fac\ x)+y$$

in an extended language.

(a) Initial memory state μ_0	(b) Final memory state $\mu_3 = Eval\,(\pi_0,\mu_0)$
π_0 :: $\lfloor\, \pi_1$ **with** $f = \pi_4 \,\rfloor$	π_0 :: $\lfloor\, \pi_5$ **with** $x = \pi_{10} \,\rfloor$
π_1 :: $\lfloor\, \pi_2\ \pi_3 \,\rfloor$	π_1 :: $\lfloor\, \pi_2\ \pi_3 \,\rfloor$
π_2 :: $\lfloor\, f \,\rfloor$	π_2 :: $\lfloor\, f \,\rfloor$
π_3 :: $\lfloor\, 5 \,\rfloor$	π_3 :: $\lfloor\, 5 \,\rfloor$
π_4 :: $\lfloor\, \textbf{fn}\ x.\pi_5 \,\rfloor$	π_4 :: $\lfloor\, \textbf{fn}\ x.\pi_5 \,\rfloor$
π_5 :: $\lfloor\, \textbf{fn}\ y.\pi_6 \,\rfloor$	π_5 :: $\lfloor\, \textbf{fn}\ y.\pi_6 \,\rfloor$
π_6 :: $\lfloor\, \pi_7\ \pi_8 \,\rfloor$	π_6 :: $\lfloor\, \pi_7\ \pi_8 \,\rfloor$
π_7 :: $\lfloor\, (add\ (fac\ x)) \,\rfloor$	π_7 :: $\lfloor\, (add\ (fac\ x)) \,\rfloor$
π_8 :: $\lfloor\, y \,\rfloor$	π_8 :: $\lfloor\, y \,\rfloor$
	π_9 :: $\lfloor\, \textbf{fn}\ x.\pi_5 \,\rfloor$
	π_{10} :: $\lfloor\, \pi_3$ **with** $f = \pi_4 \,\rfloor$

Figure 3.1.3. An example of lazy evaluation

Evaluation proceeds as follows.

$Eval\,(\pi_0,\mu_0)$: $\mu_0\,\pi_0 = \lfloor\, \pi_1$ **with** $f = \pi_4 \,\rfloor$ and $\mu_0\,\pi_1 = \lfloor\, \pi_2\ \pi_3 \,\rfloor$ [Case 5-3]
  | **let** π_9 and π_{10} be free cells;
  | **let** $\mu_1 = \mu_0 + <\pi_0 \rightarrow \lfloor\, \pi_9\ \pi_{10} \,\rfloor, \pi_9 \rightarrow \lfloor\, \pi_2$ **with** $f = \pi_4 \,\rfloor, \pi_{10} \rightarrow \lfloor\, \pi_3$ **with** $f = \pi_4 \,\rfloor >$;
  | $\mu_2 \leftarrow Eval\,(\pi_0,\mu_1)$: $\mu_1\,\pi_0 = \lfloor\, \pi_9\ \pi_{10} \,\rfloor$ [Case 3]
    | $Eval\,(\pi_9,\mu_1)$: $\mu_1\,\pi_9 = \lfloor\, \pi_2$ **with** $f = \pi_4 \,\rfloor$ and $\mu_1\,\pi_2 = f$ [Case 5-2]
      | $Eval\,(\pi_4,\mu_1)$: $\mu_1\,\pi_4 = \lfloor\, \textbf{fn}\ x.\pi_5 \,\rfloor$ [Case 4]
        | **return** μ_1;
      | $\leftarrow \mu_1$
      | **return** $\mu_1 + <\pi_9 \rightarrow \mu_1\,\pi_4>$;
    | $\leftarrow \mu_1 + <\pi_9 \rightarrow \lfloor\, \textbf{fn}\ x.\ \pi_5 \,\rfloor >$;
  | $\mu_2 = \mu_1 + <\pi_9 \rightarrow \lfloor\, \textbf{fn}\ x.\ \pi_5 \,\rfloor >$;
  | **let** $\mu_3 = \mu_2 + <\pi_0 \rightarrow \lfloor\, \pi_5$ **with** $x = \pi_{10} \,\rfloor >$;
  | $Eval\,(\pi_0,\mu_3)$: $\mu_3\,\pi_0 = \lfloor\, \pi_5$ **with** $x = \pi_{10} \,\rfloor$ and $\mu_3\,\pi_5 = \textbf{fn}\ y.\pi_6$ [Case 5-4]
    | **return** μ_3 ;
  | $\leftarrow \mu_3$
  | **return** μ_3 ;
$\leftarrow \mu_3$

The evaluator returns μ_3 shown in Figure 3.1.3(b). The result of evaluation is

$$\mu_3\,\pi_0 = \lfloor\, (\textbf{fn}\ y.\ (fac\ x)+y\ )\ \textbf{with}\ x = (5\ \textbf{with}\ f = \textbf{fn}\ x.\textbf{fn}\ y.\pi_6)\,\rfloor\ .$$

That is, g becomes

(**fn** $y.(fac\ x)+y$) **where** $x=5$

and evaluation of $(g\ 3)$ proceeds as[4]

$(g\ 3) \rightarrow (fac\ x)+y$ **with** $y=3$ **and** $x=5$ [Case 3]
$\rightarrow$ (($(add\ (fac\ x))$ **with** $y=3$ **and** $x=5$) (y **with** $y=3$ **and** $x=5$)) [Case 5-3]

The step taken as a [Case 5-3] creates two new cells to hold components of expressions with bindings. As remarked in the algorithm, these cells are needed for distributing the binding to components of an combination. The binding is used to determine values of free variables in that combination. The original expression must be preserved for the use when other bindings are given, and therefore new cells are allocated each time the combination is instantiated by a binding. Similar steps are taken to assign a cell, say π', to hold

$\pi' :: \lfloor (add\ (fac\ x))$ **with** $y=3$ **and** $x=5 \rfloor$.

The result of $(add\ (fac\ 5))$ supersedes the original expression at π' as the algorithm shows[5]. When the other operand of '+', i.e., $(g\ 4)$ is evaluated, similar computation is performed. In that process, a new cell is allocated to hold

$\lfloor (add\ (fac\ x))$ **with** $y=4$ **and** $x=5 \rfloor$

which will be evaluated in such a way that the term $(fac\ 5)$ is computed again. Thus the computation of $(fac\ 5)$ is performed twice by the lazy evaluator described above.

3.2. Fully lazy evaluation

We shall examine the lazy evaluator *Eval* in more detail before discussing full laziness. The difference between *call-by-name* and *call-by-need* parameter mechanisms has been explained in an informal manner in Section 1.2. Our lazy evaluator *Eval* uses the call-by-need mechanism. When an expression

$\mu\,\pi = \lfloor \pi'$ **with** $x_1=\pi_1$ **and** $\cdots$ **and** $x_n=\pi_n \rfloor$

where $\mu\,\pi'=\lfloor x_i \rfloor$ is evaluated, the expression at π_i bound by a variable (parameter) x_i is replaced by the result of evaluation as shown in the case [5-2] of the algorithm:

let $\mu_1 = Eval\,(\pi_i, \mu)$;
return $\mu_1 + <\pi \rightarrow \mu_1\,\pi_i>$;

If the call-by-name strategy were used, these two statements should be

---

[4] We do not show here the details of memory changes, but illustrate only the effect of evaluation.

[5] It is, however, unusual in most practical evaluators. Rewriting each expression by its result costs too much in practice. See Section 3.3.

let $\mu_1 = \mu + <\pi \rightarrow \mu\, \pi_i>$;
return *Eval* (π, μ_1) ;

In this case we first fetch the expression at π_i with which we replace the cell π, and then evaluate it in the altered memory. It would be obvious that the evaluator based on call-by-name computes the argument every time it appears because the contents of the cell π_i is never changed. We use the term 'lazy evaluation' to mean the way of evaluation performed by the original algorithm *Eval* based on call-by-need.

We now turn to the discussion of full laziness. As illustrated in the previous section, our lazy evaluator *Eval* suffers from multiple computations of the same expression. Strictly speaking, the goals stated intuitively at the beginning of this chapter have not been achieved by the ordinary lazy evaluator. It is somewhat surprising that this point had not been argued until Hughes mentioned *full laziness* of combinator reducers [Hughes84]:

Every expression is evaluated at most once after the variables in it have been bound.

It would be clear that *Eval* in the previous section does not have the property of full laziness. Hughes claims only that his reduction method for super-combinators performs fully lazy evaluation, but does not present any general algorithm for fully lazy evaluators. We shall define a fully lazy evaluator *Eval** by modifying the evaluator *Eval*. The fundamental difference between laziness and full laziness will be made clear by this approach.

If the result of evaluation of $(f\ 5)$ were[6]

π_0' :: $\lfloor$ **fn** $y.\ \pi_1' \rfloor$
π_1' :: $\lfloor\ \pi_2'\ y\ \rfloor$
π_2' :: $\lfloor$ $(add\ (f ac\ x))$ **with** $x=5\ \rfloor$,

it would be obvious that the steps for computing $(f ac\ 5)$, and in fact $(add\ (f ac\ 5))$, are taken only once when evaluating $(g\ 3)+(g\ 4)$. A cell π_2' for an instance of expression '$(add\ (f ac\ x))$' is allocated as soon as x is bound, and the contents of that cell will be updated when the expression is evaluated first time after y is bound. It will be referenced by evaluation for other occurrences of that expression.

We may rewrite the algorithm of *Eval* according to such observation.

---

[6] We omit here again further details on memory configuration for ease of reading.

Fully lazy evaluator $Eval^*(\pi,\mu)$

[1] Case $\mu\,\pi = \lfloor b \rfloor$

return μ ; { Same as *Eval*.}

[2] Case $\mu\,\pi = \lfloor x \rfloor$

return μ ; { Same as *Eval*.}

[3] Case $\mu\,\pi = \lfloor \pi_0\,\pi_1 \rfloor$

let $\mu_1 = Eval^*(\pi_0,\mu)$;

if $\mu_1\,\pi_0 = \lfloor$ **fn** $x.\pi_2 \rfloor$ **then**

let $\mu_2 = \mu_1 + <\pi \rightarrow \lfloor \pi_2$ **with** $x=\pi_1 \rfloor >$;

let $\mu_3 = Refine(\pi,\mu_2)$;

return $Eval^*(\pi,\mu_3)$;

else error ;

{ Note that the cell π_0 in memory μ_1 holds a function of the form $\lfloor$ **fn** $x.\,\pi_2 \rfloor$ and '**with** θ disappears in $Eval^*$. *Refine* distributes the binding $\lfloor x=\pi_1 \rfloor$ to the function body π_2.}

[4] Case $\mu\,\pi = \lfloor$ **fn** $x.\pi_0 \rfloor$

return μ ; { Same as *Eval*. }

{ '[5] Case $\mu\,\pi = \lfloor \pi'$ **with** $\theta \rfloor$' does not occur in $Eval^*$.}

The idea behind the changes is to bring the binding into the function body and to make instances of expressions with bindings as soon as the function is invoked. Of course, this is only conceptual. See the next section for implementation.

Any representation of the form

$\lfloor \pi$ **with** $\theta \rfloor$

is assumed to be fully reduced to a representation without '**with** θ' by *Refine*. Since association of variables with values is established in *Refine*, the rule [5] in *Eval* becomes unneccesary.

The algorithm for *Refine* is defined as follows.

Refine (π,μ)

let $\mu\,\pi = \lfloor \pi'$ **with** $\bar{x}=\bar{\pi}\rfloor$

[R-1] Case $\mu\,\pi' = \lfloor b \rfloor$

return $\mu+<\pi\rightarrow\lfloor b \rfloor>$;

[R-2] Case $\mu\,\pi' = \lfloor x \rfloor$

if $x=x_i$ **then**

return $\mu+<\pi\rightarrow\mu\,\bar{\pi}>$;

{ Replace π with $\mu\bar{\pi}$, which is an expression x denotes. }

else

return $\mu+<\pi\rightarrow\lfloor x \rfloor>$;

{ The variable x has not been bound yet. }

[R-3] Case $\mu\,\pi' = \lfloor \pi_0\,\pi_1 \rfloor$

let π_0' and π_1' be distinct free cells;

let $\mu_1 = \mu + <\pi\rightarrow\lfloor\pi_0'\,\pi_1'\rfloor> + <\pi_0'\rightarrow\lfloor\pi_0$ **with** $\bar{x}=\bar{\pi}\rfloor> + <\pi_1'\rightarrow\lfloor\pi_1$ **with** $\bar{x}=\bar{\pi}\rfloor>$;

let $\mu_2 = Refine\,(\pi_0',\mu_1)$;

return $Refine\,(\pi_1',\mu_2)$;

{ Distribute the binding $\lfloor\bar{x}=\bar{\pi}\rfloor$ to each component. New cells are required to hold the instances of expressions represented by π_0 and π_1 which may contain free variables. }

[R-4] Case $\mu\,\pi' = \lfloor$ **fn** $x.\pi_0 \rfloor$

if $x=\bar{x}$ **then**

return $\mu + <\pi\rightarrow\lfloor$ **fn** $x.\pi_0\rfloor>$;

{ The binding $\lfloor\bar{x}=\bar{\pi}\rfloor$ does not go through **fn** $\bar{x}$. }

else

let π^* be a free cell;

let $\mu^* = \mu + <\pi \rightarrow \lfloor$ **fn** $x.\pi^* \rfloor > + <\pi^* \rightarrow \lfloor \pi_0$ **with with** $\bar{x}=\bar{\pi}\rfloor >$;

return *Refine* (π^*, μ^*) ;

{ *Refine* distributes the binding to a new instance π^* of the function body. }

The memory μ_0 in Figure 3.1.3(a) represents

f 5 **where** $f =$ **fn** $x.$ **fn** $y.\ (fac\ x)+y$

as

$\pi_0 :: \lfloor \pi_1$ **with** $f = \pi_4 \rfloor$.

If we refine μ_0 by *Refine* (π_0,μ_0), we obtain a memory state $\hat{\mu}_0$ as follows. It is shown in Figure 3.2.1(a).

$\hat{\mu}_0 \leftarrow$ *Refine* (π_0,μ_0) : $\mu_0\,\pi_0 = \lfloor \pi_1$ **with** $f = \pi_4 \rfloor$ and $\mu_0\,\pi_1 = \lfloor \pi_2\,\pi_3 \rfloor$ [Case R-3]
| **let** π_9 and π_{10} be free cells;
| **let** $\hat{\mu}_1 = \mu_0 + <\pi_1 \rightarrow \lfloor \pi_9\,\pi_{10} \rfloor > + <\pi_9 \rightarrow \lfloor \pi_2$ **with** $f = \pi_4 \rfloor > + <\pi_{10} \rightarrow \lfloor \pi_3$ **with** $f = \pi_4 \rfloor >$;
| $\hat{\mu}_2 \leftarrow$ *Refine* $(\pi_9,\hat{\mu}_1)$: $\hat{\mu}_1\,\pi_9 = \lfloor \pi_2$ **with** $f = \pi_4 \rfloor$ and $\hat{\mu}_1\,\pi_2 = \lfloor f \rfloor$ [Case R-2]
| | **return** $\hat{\mu}_1 + <\pi_9 \rightarrow \hat{\mu}_1\,\pi_4$;
| $\hat{\mu}_2 = \hat{\mu}_1 + <\pi_9 \rightarrow \lfloor$ **fn** $x.\ \pi_5 \rfloor >$;
| $\hat{\mu}_3 \leftarrow$ *Refine* $(\pi_{10},\hat{\mu}_2)$: $\hat{\mu}_2\,\pi_{10} = \lfloor \pi_3$ **with** $f = \pi_4 \rfloor$ and $\hat{\mu}_2\,\pi_3 = \lfloor 5 \rfloor$ [Case R-1]
| | **return** $\hat{\mu}_2 + <\pi_{10} \rightarrow \lfloor 5 \rfloor >$;
| $\hat{\mu}_3 = \hat{\mu}_2 + <\pi_{10} \rightarrow \lfloor 5 \rfloor >$;
| **return** $\hat{\mu}_3$;
$\hat{\mu}_0 = \hat{\mu}_3$

The fully lazy evaluator *Eval** produces $\hat{\mu}_5$ of Figure 3.2.1(b) as an intermediate stage of evaluation.

*Eval** $(\pi_0,\hat{\mu}_0)$: $\hat{\mu}_0\,\pi_0 = \lfloor \pi_9\ \pi_{10} \rfloor$ [Case 3]
| $\hat{\mu}_1 \leftarrow$ *Eval** $(\pi_9,\hat{\mu}_0)$: $\hat{\mu}_0\,\pi_9 = \lfloor$ **fn** $x.\ \pi_5 \rfloor$ [Case 4]
| | **return** $\hat{\mu}_0$;
| $\hat{\mu}_1 = \hat{\mu}_0$;
| **let** $\hat{\mu}_2 = \hat{\mu}_1 + <\pi_0 \rightarrow \lfloor \pi_5$ **with** $x = \pi_{10} \rfloor >$;
| $\hat{\mu}_3 \leftarrow$ *Refine* $(\pi_0,\hat{\mu}_2)$: $\hat{\mu}_2\,\pi_0 = \lfloor \pi_5$ **with** $x = \pi_{10} \rfloor$ and $\hat{\mu}_2\,\pi_5 = \lfloor$ **fn** $y.\pi_6 \rfloor$ [Case R-4]
| | **let** π_{11} be a free cell;
| | **let** $\hat{\mu}_3 = \hat{\mu}_2 + <\pi_0 \rightarrow \lfloor$ **fn** $y.\pi_{11} \rfloor > + <\pi_{11} \rightarrow \lfloor \pi_6$ **with** $x = \pi_{10} \rfloor >$;
| | $\hat{\mu}_4 \leftarrow$ *Refine* $(\pi_{11},\hat{\mu}_3)$: $\hat{\mu}_3\,\pi_{11} = \lfloor \pi_6$ **with** $x = \pi_{10} \rfloor$ and $\hat{\mu}_3\,\pi_6 = \lfloor \pi_7\,\pi_8 \rfloor$ [Case R-3]
| | | **let** π_{12} and π_{13} be free cells;
| | | **let** $\hat{\mu}_5 = \hat{\mu}_3 + <\pi_{11} \rightarrow \lfloor \pi_{12}\,\pi_{13} \rfloor > + <\pi_{12} \rightarrow \lfloor \pi_7$ **with** $x = \pi_{10} \rfloor > + <\pi_{13} \rightarrow \lfloor \pi_8$ **with** $x = \pi_{10} \rfloor >$;
| | | . . .
| | . . .

We are convinced that the result will be[7]

[7] Note that the cell π_7 actually contains addresses of cells for '*add* ' and '$(fac\ x)$ **with** $x = \pi_{10}$'

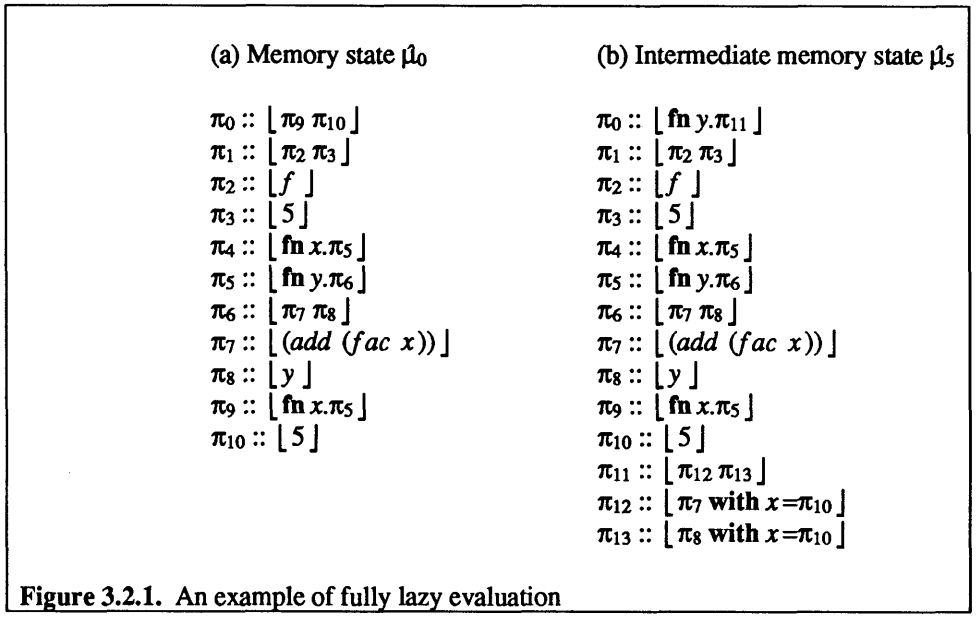


Figure 3.2.1. An example of fully lazy evaluation

π_0 :: ⌊ **fn** $y.\pi_{11}$ ⌋
π_{11} :: ⌊ π_{12} π_{13} ⌋
π_{12} :: ⌊ π_7 **with** $x=\pi_{10}$ ⌋
π_{13} :: ⌊ y ⌋

and π_{12} holds the maximal part of the expression which is dependent only on x and independent of y.

3.3. Practical fully lazy evaluators

We have described algorithms for lazy and fully lazy evaluators using an abstract memory **M** of which cell is capable of holding any kind of expressions. A real memory word does, however, contain only a limited amount of information. It may holds an atomic values and a pointer (address). A natural approach to implementing practical evaluators is to represent variable-sized memory cells in **M** with linked lists of machine words. We shall discuss several implementation methods for the fully lazy evaluator from practical viewpoints.

Refinement of bindings

The first thing we must consider is an efficient implementation technique for the *Refine* algorithm. The definition of *Refine* would be unrealistic; it distributes an argument to all the component expressions of a function body whenever the function is invoked. There are several solutions proposed.

- Assume that we deal with only a fixed number of predefined functions called *combinators* each of which contains no free variables. For example,

  $\mathbf{S} = \mathbf{fn} f.\ \mathbf{fn} g.\ \mathbf{fn} x.\ f\ x\ (g\ x)$

  and

  $\mathbf{K} = \mathbf{fn} x.\ \mathbf{fn} y.\ x$

  are combinators, but

  $F = \mathbf{fn} x.\ \mathbf{fn} y.\ x\ z$

  is not because a free variable z appears in the function body. If every expressions is composed of predefined combinators, we can implement *Refine* by embedding the rules for each combinator in the evaluator. Since there is no free variable in function bodies, association of arguments and variables is established in a simple manner.

  In fact, it is possible to transform any expression of Figure 3.1.1 into an expression comprising only combinators **S** and **K**, and other constants. Of course, a few other primitive functions are necessary for practical languages. Turner's combinator approach [Turner79] is based on this idea. See Chapter 4 for details.

- The *Refine* process essentially locates occurrences of the parameter $\bar{x}$ of the function concerned. An important implication of *Refine* is that in a function body π maximal parts $\bar{\pi}_1, \cdots, \bar{\pi}_n$ dependent on $\bar{x}$ should be identified, and only the cells in memory μ

  $\pi_i :: \lfloor\, \bar{\pi}_i \textbf{ with } \bar{x}=\bar{\pi} \,\rfloor$

  must be allocated when the function is invoked. That is, *Refine* (π_i, μ) for distributing $\lfloor\, \bar{x}=\bar{\pi} \,\rfloor$ to $\mu\, \bar{\pi}_i$ may be put off until evaluation of $\mu\, \pi_i$ occurs. What we expect from *Refine* is to allocate sharable cells π_i for maximal occurrences of expressions dependent of $\bar{x}$. Fortunately we can identify such maximal occurrences $e_1, \cdots, e_n$ corresponding to $\bar{\pi}_1, \cdots, \bar{\pi}_n$ by analysis of the scope of variables in the function body e.

  Recall that the form

  $\lfloor\, \pi \textbf{ with } \bar{x}=\bar{\pi} \,\rfloor$

  comes into existence when a form

$\lfloor \pi' \bar{\pi} \rfloor$

is in memory μ, and $Eval^*(\pi', \mu)$ results in μ_1 such that

$\mu_1 \pi' = \lfloor \mathbf{fn}\, \bar{x}.\, \pi \rfloor$.

This suggests that we may rewrite the expression e as a combination

$((\mathbf{fn}\, x_1. \cdots .\mathbf{fn}\, x_n.\, e')\, e_1 \cdots e_n)$

where e' is an expression obtained from e by replacing every occurrence of e_i in e with x_i. By doing so, we may evaluate expressions in a fully lazy way by an ordinary lazy evaluators. Hughes proposed a method based on this strategy and call such functions *super-combinators*. See Section 4.2.

Representation of expressions

The abstract memory suggests that graphical representation may be appropriate to hold expressions in actual memory. Such evaluators are known as *graph reducers*. Chapter 4 describes them in greater detail.

Another important method, and in fact which turns out to be more efficient than graph reducers on conventional computers, is to generate *fixed-code* which executes the evaluation process as in ordinary compiler languages. In such evaluators,

$\lfloor \pi \textbf{ with } \bar{x}=\bar{\pi} \rfloor$

need to be represented by a data structure called *closure*[8]. Refinement is performed just as in the super-combinator approach before code generation, but transformation for producing super-combinators is of course optional. A fixed-code approach to combinator reduction can be found in Section 4.5.

An implementation method for more efficient evaluators that achieve full laziness will be discussed in Chapters 5-8.

[8] Combinator reduction and closure reduction are compared in [Ida85] from a different viewpoint.

Chapter 4

Combinator reduction

Turner proposes an implementation technique for functional languages which is based on combinator calculus. Function definition in a functional language is translated into a lambda expression which is then transformed into a combinator expression with no free variables. Functional application in the form of combinator expression is reduced to a simpler one by an evaluator. In this chapter several implementation techniques for conventional computers are presented and these are compared each other from experimental results.

It is well known in the theory of lambda calculus that variables in lambda expressions are unnecessary if a few functions called *combinators* which embody certain common patterns of application are introduced. Combinators S and K defined as

$$\mathbf{S}\,f\,g\,x = f\,x\,(g\,x)$$
$$\mathbf{K}\,x\,y = x$$

are adequate for eliminating variables from any lambda expression. By convention we denote functional application by juxtaposition and assume that it associates to the left. Turner [Turner79] uses additional combinators for practical reasons to evaluate functional programs. Combinators included are

$$\mathbf{I}\,x = x$$
$$\mathbf{B}\,f\,g\,x = f\,(g\,x)$$
$$\mathbf{C}\,f\,g\,x = f\,x\,g$$

and extended ones of **S**, **B**, and **C** denoted by **S'**, **B'**, and **C'**.

$$\mathbf{S'}\,h\,f\,g\,x = h\,(f\,x)\,(g\,x)$$
$$\mathbf{B'}\,h\,f\,g\,x = h\,f\,(g\,x)$$
$$\mathbf{C'}\,h\,f\,g\,x = h\,(f\,x)\,g$$

Moreover combinators for conditional expressions, recursive applications, arithmetic operations, and data constructors are also introduced.

$$\mathbf{IF}\,e\,t\,f = t \text{ if } e \text{ is } \mathbf{true},\ f \text{ otherwise}$$
$$\mathbf{Y}\,f = f\,(\mathbf{Y}\,f)$$
$$+\,x\,y = x+y \text{ ,etc.}$$

A combinator expression is simply an applicative expression, i.e., a constant or a combination of a function and an argument, with the restriction that its constituents must be also applicative expressions.

Although the above combinators are considered standard, any closed function that does not have any free variable can be taken as a combinator. Hughes [Hughes82] proposes a new idea of transforming lambda expression into such a combinator expression. Specifically chosen combinators for each program are called *super-combinators.*

The combinator expression is *reduced* to a simpler expression by applying reduction rules repeatedly. This process corresponds to evaluation of programs in conventional programming systems. We may take equations defining combinators as reduction rules. For example, from the definition

$$\mathbf{B}\ f\ g\ x = f\ (g\ x)$$

we have a rule

$$\mathbf{B}\ f\ g\ x \rightarrow f\ (g\ x)\ .$$

Every occurrence of an applicative expression ($\mathbf{B}\ f\ g\ x$) may be reduced to a simpler term ($f\ (g\ x)$) whenever possible.

Section 4.1 describes a basic algorithm by Turner [Turner79] for translating applicative expressions into combinator expressions. Section 4.2 introduces the idea of super-combinators by Hughes [Hughes82]. Section 4.3 gives Turner's evaluator based on graph rewriting. Section 4.4 describes another evaluation scheme proposed by the author [Takeichi85]. It is based on graph copying and has an advantage in implementing on conventional computers. Section 4.5 deals with a compiling scheme for generating fixed-code for combinator reduction. The evaluators are compared each other using experimental results in Section 4.6.

Sections 4.1-4.3 give brief overview of the work by Turner and Hughes. Those wishing to know more on their work consult [Turner79] and [Hughes82], respectively. These sections are included for this chapter being self-contained. The rest of this chapter describes the author's work on combinator reduction in detail.

4.1. Combinators

As mentioned above, lambda expressions can be translated into expressions composed only of combinators and constants. We may call such expressions *combinator expressions*. Combinators are, in fact, considered as predefined constants representing functions. In this section, we describe Turner's method of translating *applicative expressions* into combinator expressions.

We first define a class of expressions, i.e., applicative expressions, to be dealt with in this chapter. An applicative expression is either

(1) *simple* and is either

(1.1) a constant symbol that may be a combinator,

or

(1.2) a variable

or

(2) *compound* and is a combination, i.e., a functional application of the form $(e_1\ e_2)$ where e_1 and e_2 are applicative expressions.

or

(3) a *lambda expression* of the form $\lambda x.e$ where x is a variable and e is an applicative expression[1].

Variables may appear only inside of lambda expressions, and they must be bound by some lambda binding.

A combinator expression is simply an applicative expression that contains no lambda expressions, i.e., ones of case (3) above.

For a function definition of the form

$$f\ x_1 x_2 \cdots x_n \equiv e$$

where e is an applicative expression, f can be expressed as an applicative expression

$$f = \lambda x_1.\lambda x_2.\cdots.\lambda x_n.e$$

Hence, we do not consider translation rules for definitions in this chapter[2].

---

[1] We use fn-binding instead of λ-binding in other chapters of this thesis.

[2] Turner describes such a rule for definitions in [Turner79].

Consider, for example, a definition of the factorial function

$$fac\ n \equiv \mathbf{IF}\ (=n\ 0)\ 1\ (*\ n\ (fac\ (-n\ 1)))$$

where **IF**, +, * and – are predefined function constants. For simplicity, we consider a symbol fac as a constant. The right-hand side is an applicative expression for

if n=0 **then** 1 **else** n*$fac\,(n-1)$

in conventional notation. Recall that functional application associates to the left, and the right-hand side of the above definition is a short hand for

$$(((\mathbf{IF}\ (=n\ 0))\ 1)\ (*\ n\ (fac\ (-n\ 1))))$$

The above definition can be written using lambda notation as

$$\lambda n.\ \mathbf{IF}\ (=n\ 0)\ 1\ (*\ n\ (fac\ (-n\ 1)))$$

Translation of a lambda expression into combinators is to abstract variables from the body expression. If the body expression e of a lambda expression $\lambda x.e$ is a lambda-free applicative expression, abstracting x from e basically proceeds according to the following rules:

$$\lambda x.x \rightarrow \mathbf{I}$$
$$\lambda x.a \rightarrow \mathbf{K}\ a \qquad \text{where } x \neq a$$
$$\lambda x.(e_1\ e_2) \rightarrow \mathbf{S}\ (\lambda x.e_1)\ (\lambda x.e_2)$$

The rules for abstraction must be applied recursively until no lambda expression remains. Note that the translation removes lambda expressions by introducing combinators **I**, **K**, and **S**. However, applying these rules to a lambda expression tends to yield a very large combinator expression. Turner proposes optimization rules using following relations to reduce the size of combinator expressions.

$$\mathbf{S}\ (\mathbf{K}\ \alpha)\ (\mathbf{K}\ \beta) = \mathbf{K}\ (\alpha\ \beta)$$
$$\mathbf{S}\ (\mathbf{K}\ \alpha)\ \beta = \mathbf{B}\ \alpha\ \beta$$
$$\mathbf{S}\ \alpha\ (\mathbf{K}\ \beta) = \mathbf{C}\ \alpha\ \beta$$

He also introduces several combinators as mentioned in the beginning of this chapter.

The above translation rules for the lambda expression assume that its body has already been translated into a lambda-free applicative expression in the same manner if its original contains lambda expressions. Thus, we have a straight algorithm *trans* for translating any applicative expression e :[3]

[3] Syntactic terms are quoted by quasi-parentheses ⌜ and ⌝ to distinguish them from ones describing the algorithm.

trans e =

if e is a variable **then** e

else if e is a constant **then** e

else if e is a combination $\lceil(e_1\ e_2)\rceil$ **then** $\lceil(($*trans* $\lceil e_1\rceil)$ (*trans* $\lceil e_2\rceil))\rceil$

else let e be a lambda expression $\lceil\lambda x.e_0\rceil$ **in** *abstr* $\lceil x\rceil$ (*trans* $\lceil e_0\rceil$)

abtsr v e =

let $v=\lceil x\rceil$ **in**

if e is a variable $\lceil x\rceil$ **then** $\lceil$**I**$\rceil$

else if e contains no occurrences of $\lceil x\rceil$ **then** $\lceil$**(K** e**)**$\rceil$

else if e is a combination $\lceil(e_1\ e_2)\rceil$ **then**

if $\lceil e_1\rceil$ contains no occurrences of $\lceil x\rceil$ **then**

$\lceil$**(B** $\lceil e_1\rceil$**)** (*abstr* $\lceil x\rceil$ $\lceil e_2\rceil$**)**$\rceil$

else if $\lceil e_2\rceil$ contains no occurrences of $\lceil x\rceil$ **then**

$\lceil$**(C** (*abstr* $\lceil x\rceil$ $\lceil e_1\rceil$**)** $\lceil e_2\rceil$**)**$\rceil$

else $\lceil$**(S** (*abstr* $\lceil x\rceil$ $\lceil e_1\rceil$**))** (*abstr* $\lceil x\rceil$ $\lceil e_2\rceil$**)**$\rceil$

Translating the lambda expression in the above definition of fac, we get

fac = **S (C (B IF (= 0)) 1) (S * (B** fac **(C** $-$ **1)))**

Related works include discussions on the complexity of Turner's algorithm and proposals for new representation of combinators. Kennaway [Kennaway82] deals with the complexity of the translation algorithm and proves that the size of the resulting combinator expression is $O(n^2)$ in the *worst* case where n is the size of the original lambda expression. Hikita [Hikita84] shows that the size of the combinator expressions is $O(n^{3/2})$ in the *average* case. Noshita [Noshita85a] proposes an idea of encoding sequences of combinators to a single symbol to reduce the size of the resulting expression to $O(n \log n)$. However, we do not discuss their works here.

4.2. Super-combinators

Turner's method described in the previous section transforms an applicative expression to a combinator expression consisting of predefined combinators and constants. However, an idea of generating combinators according to the source expression is proposed by Hughes [Hughes82]. In fact, any lambda expression with no free variable can be taken as a combinator. Consider a function

$\Phi = \lambda n.\ IF\ (=n\ 0)\ 1\ (*\ n\ (fac\ (-n\ 1)))$.

It contains no free variables and therefore it is a combinator. We may use Φ in defining the factorial function fac as

$$fac\ n\ =\ \Phi n\ .$$

The right-hand side is a combinator expression.

In this case we have derived the combinator Φ from the definition of fac with no difficulties because it takes only one parameter and the body expression contains no lambda expression. It is, however, not the case with functions that have more than one parameters or contain lambda expressions. A simple abstraction rule is to translate the lambda expression as follows: if $\lambda x.e$ contains free variables $y_1, y_2, \cdots, y_m$, then translate it into

$$\Phi\ y_1\ y_2\ \cdots\ y_m$$

where

$$\Phi\ =\ \lambda y_1.\ \lambda y_2.\ \cdots\ .\ \lambda y_m.\ \lambda x.\ e$$

or

$$\Phi\ y_1\ y_2\ \cdots\ y_m\ x\ =\ e$$

in the form of a function definition. Repeated application of this rule eliminates all the lambda expression in e. It is true, therefore, that the function Φ becomes really a combinator in the sense described above. However, such a simpleminded method does not generate combinators for fully lazy evaluation (See Section 3.2).

Hughes improves this to generate *super-combinators* for full laziness. The basic idea is to extract maximal free occurrences $\alpha_1, \alpha_2, \cdots, \alpha_m$ of expressions from $\lambda x.e$ with respect to the bound variable x, and construct a combinator Φ as

$$\Phi\ a_1\ a_2\ \cdots\ a_m\ x\ =\ e\ [a_1/\alpha_1, a_2/\alpha_2, \cdots, a_m/\alpha_m]$$

where $e\ [a_1/\alpha_1, \cdots, a_m/\alpha_m]$ represents an expression obtained from e by replacing all the occurrences of α_j with a_j simultaneously. Informally speaking, the maximal free occurrence of expressions with respect to x is the maximal part of an applicative expression independent of x. See Section 6.3 for the formal definition.

Consider for example a highly recursive function

$$f\ x\ y\ z \equiv IF\ (>\ z\ y)\ (f\ (f\ y\ z\ (-\ x\ 1))\ (f\ z\ x\ (-\ y\ 1))\ (f\ x\ y\ (-\ z\ 1)))\ y$$

Maximal terms independent of the last parameter z are

$$\alpha_1 = x,\ \alpha_2 = (-\ x\ 1),\ \alpha_3 = (f\ x\ y),\ \alpha_4 = (-\ y\ 1),\ \alpha_5 = (f\ y),\ \text{and}\ \alpha_6 = y.$$

We can define a super-combinator

$$\phi_z\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ z\ =\ IF\ (>\ z\ a_6)\ (f\ (a_5\ z\ a_2)\ (f\ z\ a_1\ a_4)\ (a_3\ (-\ z\ 1)))\ a_6$$

and rewrite the original definition as

$$\begin{aligned} f\ x\ y\ &= \phi_z\ \alpha_1\ \alpha_2\ \alpha_3\ \alpha_4\ \alpha_5\ \alpha_6 \\ &= \phi_z\ x\ (-\ x\ 1)\ (f\ x\ y)\ (-\ y\ 1)\ (f\ y)\ y. \end{aligned}$$

Similarly we get another super-combinator by abstracting the right-hand side with respect to y.

$$\phi_y\ a_1\ a_2\ y\ =\ a_1\ (a_2\ y)\ (-\ y\ 1)\ (f\ y)\ y$$

and

$$f\ x\ =\ \phi_y\ (\phi_z\ x\ (-\ x\ 1))\ (f\ x).$$

Finally we have

$$\phi_x\ x\ =\ \phi_y\ (\phi_z\ x\ (-\ x\ 1))\ (f\ x)$$

and

$$f\ =\ \phi_x.$$

Note that the super-combinators ϕ_x, ϕ_y, and ϕ_z have been derived so that the occurrences of maximal free terms with respect to x, y, and z are replaced by parameters and they appear as argument to those combinators in reconstructing the original expression. As explained by Hughes, the order of parameters is critical for defining super-combinators. We shall leave the details to [Hughes82].

4.3. Graph rewriting evaluator

Turner [Turner79] shows an evaluator for combinator expressions based on graph rewriting. Every combinator expression is represented by a graph of which nodes correspond to functional applications. Every node has a function field and an argument field as shown in Figure 4.3.1. We have a graph representation of the combinator expression for fac in Section 4.1. as Figure 4.3.2. The evaluator uses a stack called *left ancestor stack (LAS)* to keep the pointers to the nodes. The stack initially contains the pointer to the node representing the expression to be evaluated. The function field of the node pointed to by the top of

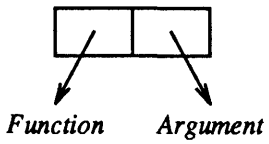


Figure 4.3.1: Representation of functional application

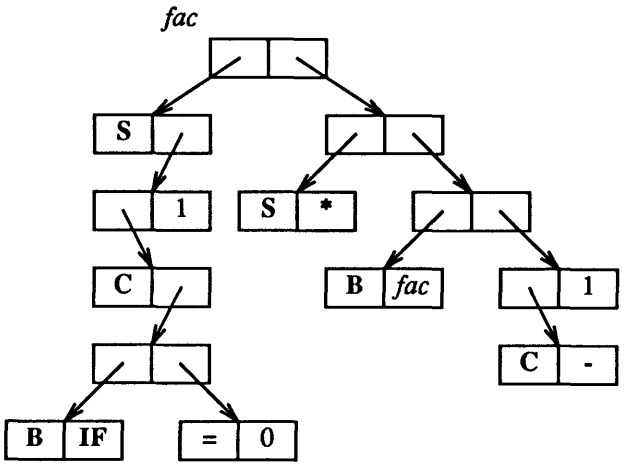


Figure 4.3.2: Representation of a combinator expression

the stack is pushed down in turn until a combinator appears at the top of the stack. The reduction rule for that combinator is then applied as shown in Figure 4.3.3.

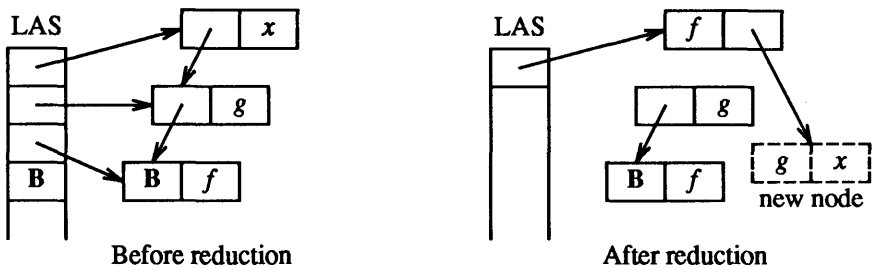


Figure 4.3.3: Reduction of **B** by graph rewriting

After combinator reduction is performed, stacking operation is resumed. This process terminates in either

of the following cases. No more reduction proceeds if data has been obtained, in which case it is the result of evaluation. If the number of arguments supplied is less than required for yielding data, reduction cannot be taken. The evaluator returns the partially reduced graph that represents a function.

The graph is transformed according to the rules for combinator reduction. Any node representing an expression might be pointed to by pointers from several nodes as a result of sharing common expressions. The graph rewriting evaluator deals with such a shared node in the graph as follows. The result of combinator reduction is always left in the root node of the original expression. Hence the node to which the reduction rule is applied is necessarily overwritten with the result as shown in Figure 4.3.4.

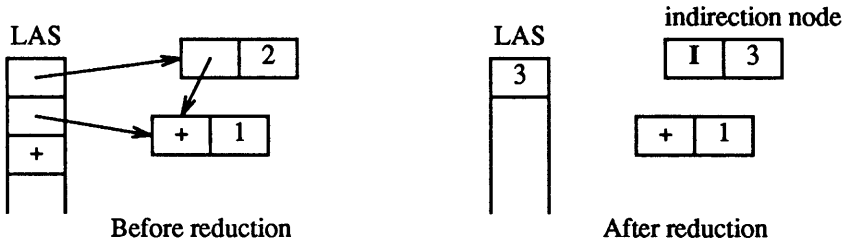


Figure 4.3.4: Reduction of + by graph rewriting

If the reduction does not result in a combination, the root node becomes an *indirection node* with the identity function **I** in the function field and the value obtained in the argument field. Combinators for arithmetic operations and the standard combinator **K** always require indirection nodes. The use of the indirection node is essential for call-by-need evaluation.

4.4. Graph copying evaluator

The key to the *graph copying* scheme relies on the fact that the expression graph can be simplified by copying its nodes and arcs instead of rewriting the root node. The node pointed to from multiple nodes in the graph appears only at reduction of specific combinators. Consider for example the reduction of **S** with f, g, and x as its arguments. It yields a graph representing $(f\ x\ (g\ x))$ in which only the occurrences of x should refer to the identical node for x. The root node for $(f\ x\ (g\ x))$ may be created.

The evaluator by graph copying also uses a stack called *argument stack (ARGS)* to keep the arguments themselves instead of the pointers. The mechanism of pushing arguments onto the stack and

dispatching combinators for reduction follows the way the graph rewriting evaluator does. Since rewriting the node is unnecessary in graph copying, the stack keeps only the argument. Instead of rewriting, a copy of the expression graph is made according to the need. Figure 4.4.1 shows how the standard combinators B and '+' are reduced by graph copying.

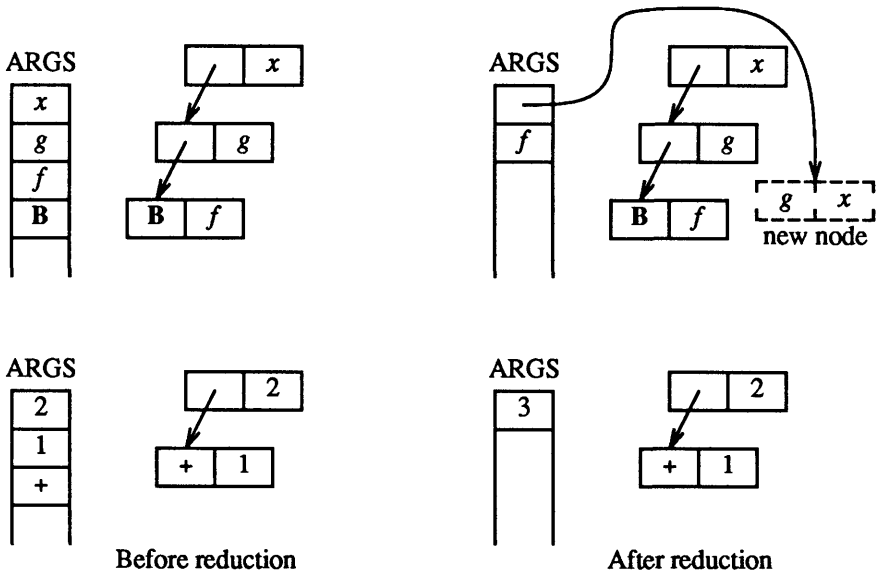


Figure 4.4.1: Reductions of B and + by graph copying

A problem arises along with the reduction of S as mentioned above. Although a straightforward application of the method could implement the copy rule of call-by-name parameters, each occurrence of the identical expression need to be evaluated repeatedly. Two occurrences of the third argument x should be identical after the reduction. Once either of these happens to be reduced to yeild a (possibly functional) value, the other should become that value. Such a call-by-need mechanism is most important for lazy evaluators. Fortunately, it can be implemented by inserting a special combinator T only to the expression to be shared. The name of that combinator comes from an established compiling technique of using *thunks* for call-by-name parameters in procedural languages [Ingerman61]. Figure 4.4.2 shows the effect of reducing S by graph copying. Note that the node with T in its function field is pointed to from two arguments fields. The reduction rule for T differs slightly from the standard one.

$$\mathbf{T}\ (\text{eval}\ e) \rightarrow e'$$

with rewriting the node to become (T (deliver e')) where e' is the result of evaluation of e, and

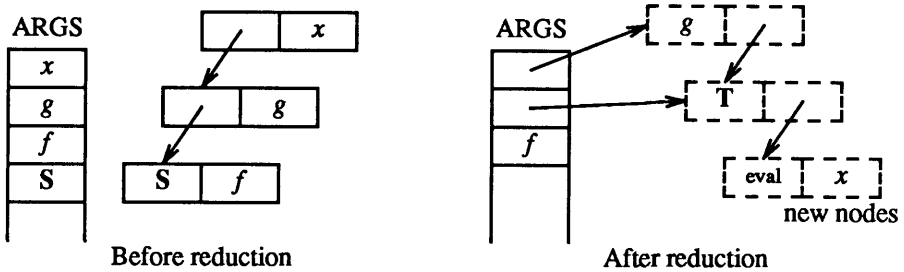


Figure 4.4.2: Reduction of S using thunk T

T (deliver e) $\rightarrow$ e

Two distinguished constants *eval* and *deliver* are used with T. The effect of applying the rule for T is shown in Figure 4.4.3. A similar mechanism is suggested in [Jones82] for fixed-code generation of combinators (See Section 4.5).

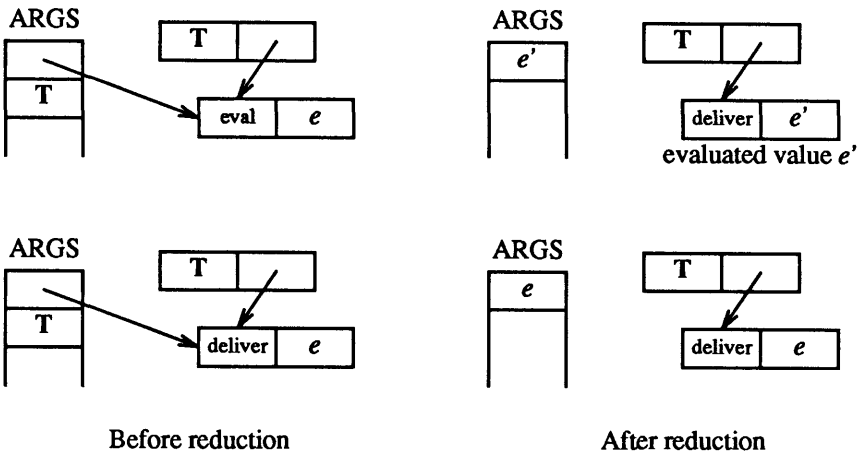


Figure 4.4.3: Reduction of thunk combinator T

Extension of the above method to super-combinators presents no major problems. Assume that the reduction rule for a combinator Φ is

$$\Phi\, x_1\, x_2 \cdots x_n \rightarrow F(x_1, x_2, \cdots, x_n)$$

where F is an applicative expression comprising variables $x_1, x_2, \cdots, x_n$, and constants. When reducing Φ, thunks, or the nodes with T in their function field, are necessary only for arguments corresponding to variables with multiple occurrences in F. Reduction by graph copying for Φ proceeds in general as follows.

Let X_t be the set of variables that appear more than once in F.

(1) Make thunks with *eval* indicator each for x in X_t.

(2) Fill the argument field of the thunk for x_i in X_t with i-th argument on the stack.

(3) Construct a graph representing an applicative expression for F by replacing x_j not in X_t with j-th argument on the stack and x_i in X_t with the corresponding thunk.

Algorithm for graph copying evaluator

1:

while top of *ARGS* points to an application node **do**

Discard the top of *ARGS*;

Push argument and function parts of the node in this order;

if top of *ARGS* is a combinator **then**

if number of arguments on *ARGS* is sufficient to reduce **then**

Reduce the combinator using as many arguments on *ARGS*;

Put the resulting expression onto *ARGS* in place of them;

goto 1;

else

Construct a graph for partially evaluated function;

return the result as the value of the original expression;

if top of *ARGS* is a data **then return** it;

if top of *ARGS* is a thunk **then**

if it has *eval* indicator **then**

Evaluate the expression in the thunk recursively;

Rewrite the thunk with *deliver* indicator and the result;

Put the result on *ARGS* in place of the thunk;

goto 1;

else

Get the value from the thunk;

Put it on *ARGS* in place of the thunk;

goto 1;

It should be noted that the evaluator is called recursively for evaluating thunks and special combinators as **IF**, and +.

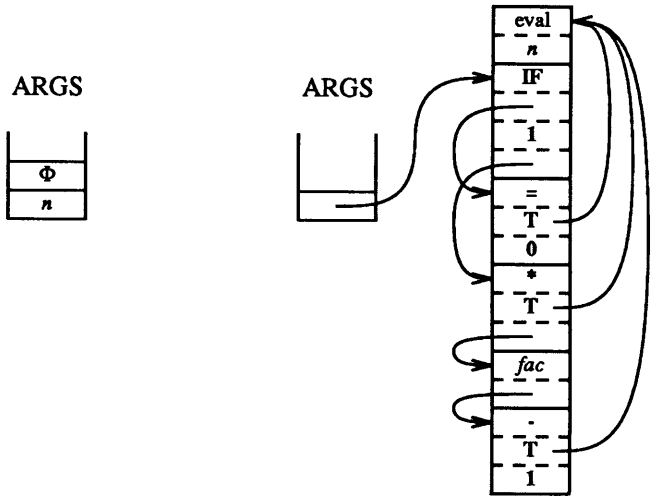
Comparison with the graph rewriting evaluator

Rewriting the node to which the reduction rule is applied ensures the call-by-need property in the graph rewriting scheme. If we consider combinators as machine instructions, and super-combinators as microprogrammed instructions, graph rewriting evaluation looks like self-reorganizing execution of machine-coded programs. From a methodological point of view, such programs should be avoided at least on conventional hardware. This holds also for a similar scheme used in SKIM machine [Clarke80]. The fixed-code scheme to be described in the next section relies on compiling combinator expressions to generate stack machine code. As in the fixed-code, the graph copying evaluator does not change any part of the expression graph except thunks. It is therefore easy to adapt the method to super-combinators.

The number of cells for representing the application node seems to be approximately the same in both evaluators. In fact, it turns out that the numbers of **S** reductions and **C** reductions are approximately the same. Evaluation of the factorial function fac applied to n reduces both combinators $2n+1$ times each.

Graph copying scheme using chunks

As to storage allocation strategies in the graph copying evaluator, there is much room for choice. One may use *chunks*, i.e., contiguous memory words, of arbitrary size for storing applicative form with many arguments as shown in Figure 4.4.4. The size of the chunk has to be kept either in itself or with the pointer to it. An algorithm for the evaluator using chunks follows.



Before reduction After reduction

Figure 4.4.4: Chunks for super-combinator reduction

1:

while top of *ARGS* points to a chunk of application **do**

Discard the top of *ARGS*;

* Move the chunk onto *ARGS*;

if top of *ARGS* is a combinator **then**

if number of arguments on *ARGS* is sufficient to reduce **then**

* Reduce it using as many arguments on *ARGS*;

Put the result on *ARGS* in place of them;

goto 1;

else

* Dump the arguments on *ARGS* as a chunk representing a function;

return it as the value of the expression;

Other cases are the same as before

The lines marked * offer the advantage of using data transfer instructions such as

load multiple memory words into registers,

store registers into multiple words, or

move multiple words from memory to memory

Such instructions cannot be used in graph rewriting.

The second of * marked lines would require explanation. As shown in Figure 4.4.4, the result of reduction can be represented by chunks allocated in contiguous memory words; in this case, 6 chunks are allocated in 17 words. In this way, we can allocate as many words as required all at once, and then copy the template of the resulting graph to these words using data transfer instructions. After copying the template, it is only necessary to adjust pointers to the chunks just being created, and fill argument positions with arguments on the stack.

See Section 4.6 for experimental results.

4.5. Fixed-code for combinator reduction

In the previous sections, we have discussed implementation techniques for combinator reduction based on graph rewriting and graph copying. The graph reduction is usually performed by an interpreter that manipulates combinator graphs according to the reduction rules. Our purpose in this section is to present an alternative scheme for efficient combinator reduction on conventional computers; a technique of translating combinator expressions into *fixed-code* programs for a stack machine. The code is executed directly by the machine and it remains unchanged throughout the execution as the object code of conventional compiler languages. Hence the code is fixed in this scheme, while the combinator graph is transformed during evaluation.

Objects manipulated by the stack machine are either

- an irreducible atomic value such as numbers and data structures. or
- a pair consisting of a code address γ and a pointer to an environent chunk ε in the heap store.

The second kind of object which we may call *closure* and denote by $[\gamma{:}\varepsilon]$ corresponds to a combination, i.e., functional application, that is not yet evaluated. (See Chapter 7 for detailed discussions on the closure.)

The object is pushed onto the *stack* and is moved to the *heap* store as an element of an environment.

Environment chunks are created when reductions are taken and variables (parameters) are bound. They are used to evaluate particular instances of expressions. If values of variables are required in evaluation of these instances, they are retrieved from the environment. A combinator may be considered as an irreducible value because it represents itself and is a kind of constants. If the reduction rule for a combinator Φ is to be implemented by a code chunk with address γ_Φ, the value of the combinator can be represented by a closure $[\gamma_\Phi : \phi]$ where ϕ is an empty environment.

Other irreducible values, i.e., numbers, data structures, etc., can also be represented by a *pseudo-closure*. If a code chunk beginning at address *deliver* returns the environment part of the closure, a pseudo-closure [*deliver*:α] always yields the value α. Hence we may consider that the object manipulated by our stack machine is a single kind of quantity, i.e., a closure.

The stack machine (Figure 4.5.1) has four registers[4]:

ap: argument pointer

fp: frame pointer

ep: environment pointer

pc: program counter

The registers **ap** and **fp** points to the stack; **ap** watches the top element of the stack, and **fp** holds the base address of the current stack frame created by a recursive invocation of the code. The environment pointer register **ep** normally contains the address of a chunk in the heap as its name suggests, or sometimes has an irreducible value as mentioned above. The program counter **pc** is incremented by executing the code except when an explicit transfer of the control is forced.

Consider the reduction rule

$$\Phi\, x_1\, x_2 \cdots x_n \rightarrow F(x_1, x_2, \cdots, x_n)$$

where Φ is a standard combinator or a super-combinator, and F is an applicative expression. The code that

[4] Implementation of the heap store calls for another register **hp** (heap pointer) to allocate chunks. However, it is irrelevant here and is omitted for brevity.

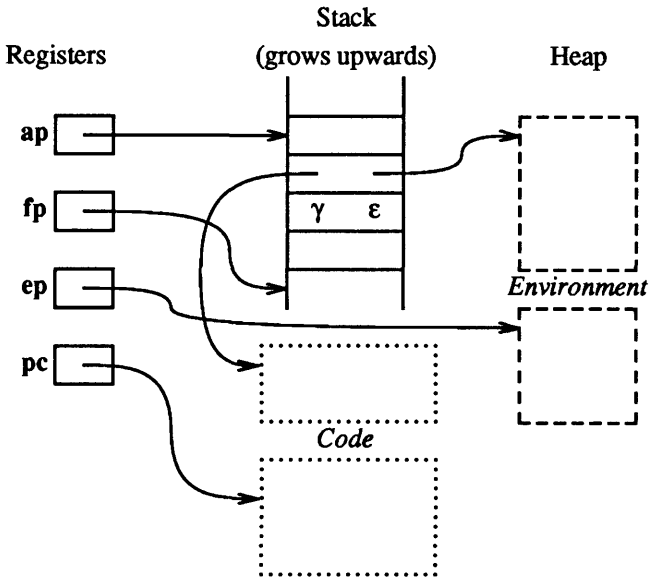


Figure 4.5.1: The stack machine for combinator reduction

reduces Φ should do the following action:

if (number of arguments on the stack) ≥ n **then**

Pop up n arguments $\alpha_1, \alpha_2, \cdots, \alpha_n$ from the stack;

Create an environment chunk ε containing $\alpha_1, \alpha_2, \cdots, \alpha_n$;

Set **ep** point to ε;

Jump to the code address γ for the expression F;

else

return a functional value, i.e., a closure

Putting aside the translation rule for generating fixed-code in this section, we shall investigate actions that should be taken by the stack machine. Translation rules for a more general setting are described in Chapter 7.

Assume that

$$F(x_1, x_2, \cdots, x_n) = (e_0, e_1, \cdots, e_m)$$

where e_j, $(j=0,1,\cdots,m)$ are applicative expressions comprising variables x_i and constants, but no other

variables. Since all the e_j may contain only variables in the environment ε, the values of e_j's in a particular instance of the right-hand side can be completely determined using ε.

If e_j is an irreducible constant such as a number, or it is a combinator, its value is itself.

If e_j is a variable x_i, the value currently bound to x_j can be found in the environment chunk ε which is accessible through **ep**.

If e_j is again a combination, its value can be obtained by evaluating it in the same way using ε. In combinator reduction, evaluation of e_j should be delayed until its value is actually needed[5]. It is, therefore, appropriate to represent its value by a closure $[\gamma_j : \varepsilon]$ where γ_j is the code address for the reduction of the combination e_j. It should be noted that irreducible values can also be represented by pseudo-closures as explained above.

The stack machine put the values of $e_m, \cdots, e_1, e_0$ onto the stack in this order. Then the machine pops up and examines the value on the top of the stack, i.e., the value of e_0. If it is an irreducible value other than a combinator, that value should be returned to the caller as the result of evaluation. If it is a combinator, reduction proceeds much the same way as above. Recall that the value of a combinator is simply a closure $[\gamma_\Phi : \phi]$ where γ_Φ is the code address for Φ.

As a more concrete example, we have a part of the fixed-code for reducing the combinator **B**

$$\mathbf{B}\ f\ g\ x \rightarrow f\ (g\ x)$$

as follows.

γ_B: code for reducing **B**

if (fb–ap)≥3 then

Pop up α_f, α_g, and α_x from the stack;

Create an environment chunk ε and fill it with α_f, α_g, and α_x;

ep := ε;

go to $\gamma_{(f\ (g\ x))}$;

---

[5] See Chapter 3 for details on lazy evaluation.

else return a functional value;

$\gamma_{(f\ (g\ x))}$: code for reducing $f\ (g\ x)$

Push a closure [$\gamma_{(g\ x)}$: **ep**] onto the stack;

Push α_f which is the first element of ε poined to by **ep**;

Pop up and examine the top of the stack;

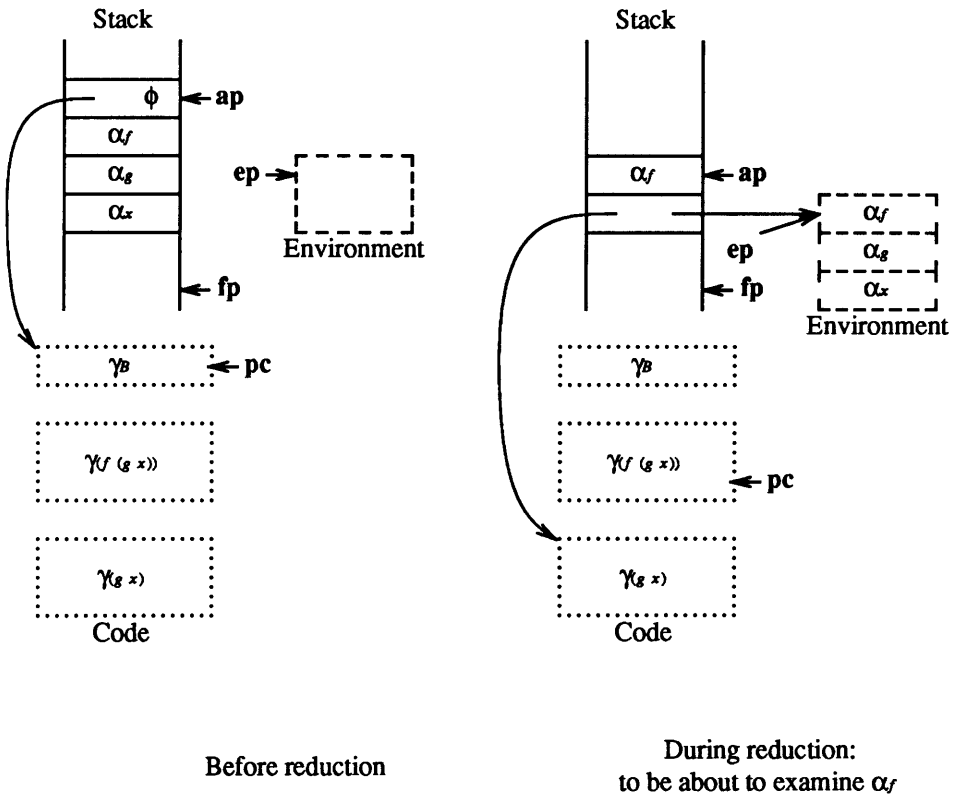
$\gamma_{(g\ x)}$: code for reducing $g\ x$

Push α_x which is the third element of ε;

Push α_g which is the second element of ε;

Pop up and examine the top of the stack;

Snapshots of the reduction process is illustrated in Figure 4.5.2.



Before reduction

During reduction:
to be about to examine α_f

Figure 4.5.2: Snapshots of reduction of **B**

It should be noted that the thunk combinator T is necessary for call-by-need evaluation as in the graph copying scheme.

A similar but more versatile stack machine is introduced in Chapter 7, where an algorithm for compiling functional programs into fixed-code programs of that machine is also presented. We do not give a compilation algorithm in this section, since it is very similar to one given in that chapter.

4.6. Experimental results

In this section we present some experimental results on the efficiency of evaluators based on the methods explained in Sections 4.3-4.5. The evaluators were written on a typical conventional computer. As we have mentioned at the beginning of this chapter, it is the main subject here to investigate techniques for implementing efficient combinator reducers on such a machine.

The combinator reducers

In order to ensure that our measurements of the efficiency faithfully reflect the intrinsic qualities for implementation on conventional computers, we write the evaluators in the assembly language. Experimental results reported so far were obtained using interpreters written in higher level languages [Hughes84]. Another experimentation gave the number of reduction steps instead of actual computation time [Peyton-Jones82]. However, there may be the possibility of losing the true nature of the schemes applied to implementation on conventional machines.

Three evaluators were written by the author in the assembly language with fair keenness on a computer Melcom 70/250. The machine normally accesses data in the memory using general registers.

- Graph Rewriting Reducer (GRR) based on the scheme described in Section 4.3.
- Graph Copying Reducer (GCR) based on the scheme described in Section 4.4. This evaluator uses instructions for transferring multiple words from memory to memory, and allocates chunks for graph nodes.
- Fixed-code (FC) of the stack machine described in Section 4.5. The instruction of the stack machine is implemented by code sequences of the target machine. We assign general registers

to the registers of the stack machine. Our target machine has no sophisticated hardware stack.

The graph rewriting reducer GRR manipulates combinator graphs comprising standard **S, K, I, B, C, IF, Y** and strict combinators such as arithmetic and Boolean operators; we call it *GRR[SKIBC]*. The graph copying reducer GCR deals with both standard combinators and super-combinators; we call them *GCR[SKIBC]* and *GCR[SC]*, respectively. The fixed-code FC is generated from super-combinator expressions because generation of fixed-code for pre-defined combinators is less meaningful. Such a fixed-code program is referred as *FC[SC]*.

Basis for comparison

The comparison among the three combinator reducers was performed according to the following observations.

(1) As mentioned in Section 4.4, GCR has the advantage over GRR that it can easily be adapted for super-combinators. Is there any drawback to GCR applied to the standard combinators? Which of GRR[SKIBC] and GCR[SKIBC] runs faster?

(2) It would be a natural consequence that GCR[SC] should be more efficient than GCR[SKIBC] because the use of the standard combinators is considered as a specific choice of (super-) combinators, while combinators are generated from the source expression in the super-combinator approach. To what degree does GCR[SC] run faster than GRR[SC]?

(3) It would be obvious that FC[SC] runs faster than its graph reduction counterpart GCR[SC], because it is considered as a compiled code for reduction performed by the GCR interpreter. How fast does FC[SC] run compared with GCR[SC]?

(4) As for the amount of store claimed, actual numbers of words required for reduction should be measured. The cell representing a node of the graph manipulated by GRR requires 2 words, and the chunk for GCR and FC varies in size as described in Sections 4.4 and 4.5.

(5) The time used by the garbage collector seems almost the same among the three reducers. The total number of words required is more informative than the time for garbage collection.

Experiment 1

In order to compare the time and the store required for reduction, we chose an expression defined by a highly recursive function f [6].

$f\ 0\ 30\ 60$

whererec

$f\ x\ y\ z =$

$IF\ (>\ z\ y)\ (f\ (f\ y\ z\ (-\ x\ 1))\ (f\ z\ x\ (-\ y\ 1))(f\ x\ y\ (-\ z\ 1)))\ y$

The value of $(f\ 0\ n\ 2n)$ is $2n$. The time required is proportional to n^2 by call-by-need evaluation as we have done with the reducers, while it is proportinal to 30^n by call-by-value evaluation. All of our reducers use the call-by-need mechanism. The results are summarized in Table4.6.1.

Table 4.6.1: Run-time and store claimed for $(f\ 0\ 30\ 60)$

Reducer [combinator]	GRR [SKIBC]	GCR [SKIBC]	GCR [SC]	FC [SC]
Run-time in msec. (ratio)	8646 (2.85)	7428 (2.45)	3030 (1)	1572 (0.52)
Store claimed in words (ratio)	149432 (1.32)	120626 (1.06)	113446 (1)	66432 (0.59)

We may say that GCR is better than GRR even in the case of the standard combinators on our machine. As for the choice of combinators, the super-combinator expression is reduced about two and a half times faster than the expression based on the standard combinators using approximately the same amount of store. Hughes also reported similar results [Hughes82].

The fixed-code runs two times faster than graph reducers and saves significant amount of space.

Experiment 2

As a typical example of using infinite lists implemented by lazy evaluation, we chose the problem of finding the n-th prime number. We measured the run-time and the amount of store claimed by GRR[SKIBC] and GCR[SC] for the 30-th prime[7]:

[6] See also Section 4.2 for the definition of f and the super-combinators derived from it.

[7] The program shown is written in *uc*. See Chapter 8 for details.

```
nth 30 primes
    whererec {
      primes = sieve [ 2 .. ]
    and
      sieve (p:x) = p : sieve [ n | n <- x; n %p !=0 ]
    and
      nth n (a:x) = if n==1 then a else nth (n-1) x
    }
```

The results are shown in Table 4.6.2.

Table 4.6.2: Run-time and Store claimed for 30-th prime

Reducer [combinator]	GRR [SKIBC]	GCR [SC]
Run-time in msec. (ratio)	1950 (2.33)	836 (1)
Store claimed in words (ratio)	30572 (2.05)	14916 (1)

This experiment demonstrates more on the effectiveness of the super-combinator approach. It is worth noting that tha saving of the store is significant as well as the running time.

Experiment 3

In order to estimate the number of function calls per unit time, we used the function *nFib* in [Henderson83]:

```
nFib n = if n≤1 then 1 else nFib (n-1)+nFib (n-2)+1
```

The result of this function gives the number of function invocations required to calculate it. Each of the timings in Table 4.6.3 was the average of the results obtained by calls of *nFib* on various arguments.

Table 4.6.3: Number of function calls per second by *nFib* benchmark

Reducer [combinator]	GRR [SKIBC]	GCR [SC]	FC [SC]
Function calls per second (ratio)	561 (0.39)	1430 (1)	2447 (1.71)

Relative efficiencies in time of the reducers are observed from the result. These figures are well comformed with the timings in Experiment 1. The count for FC[SC] is comparable to the micro-coded evaluator for a lazy SECD machine [Henderson83][8].

Experiment 4

The function f used in Experiment 1 was treated as a curried function with the functionality

$$f \ : \ \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int}$$

That is, $(f\ x)$ is again a function with two arguments, and $(f\ x\ y)$ becomes another function with an argument. It is a well-known fact that every curried function f has its uncurried counterpart f' and vice versa. In our case,

$$f' \ : \ \text{int} \times \text{int} \times \text{int} \rightarrow \text{int}$$

and the call of the uncurried function f' corresponds to make the function itself as a super-combinator with three arguments. Table 4.6.4 compares the run-time required by curried and uncurried functions applied to arguments 0, 30, and 60.

Table 4.6.4: Run-time for curried and uncurried functions

Reducer [combinator]	GCR [SC]		FC [SC]	
Function definition	curried	uncurried	curried	uncurried
Run-time in msec. (ratio)	3030 (1.59)	1910 (1)	1572 (1.48)	1060 (1)

As described in Chapter 2, the higher order function of the curried form is useful in functional programming. There is a problem, however, in the combinator approach as shown from this experiment. We shall device a mechanism for efficient implementation of the higher order function in Chapter 5-8.

Conclusion

We may conclude from these experiments that:

[8] However, an elabolate compilation scheme for generating fixed-code gains much more in both time and space. See Chapters 5-8.

- The graph copying scheme is superior to the graph rewriting scheme on conventional computers.
- The use of super-combinators produces a significant improvement in graph reduction methods.
- The fixed-code runs about twice as fast as the graph reducer.

Chapter 5

Fully lazy normal form

This chapter introduces a class of expressions called the *fully lazy normal form* which will be used through Chapters 6-8. The reason for using such expressions is motivated in Section 5.1. Section 5.2 provides the definition of the fully lazy normal form for a simple language. The relation with the super-combinator approach is given in Section 5.3 with some experimental results. Finally in Section 5.4, fully lazy evaluation of expression is demonstrated through an example.

5.1. Motivation

As explained in Section 4.2, a lambda expression $\lambda x.e$ with maximal free occurrences of expressions $\alpha_1, \alpha_2, \cdots, \alpha_m$ can be transformed into

$$\Phi\ \alpha_1\ \alpha_2\ \cdots\ \alpha_m$$

where Φ is a super-combinator defined as

$$\Phi\ a_1\ a_2\ \cdots\ a_m\ x\ =\ e\ [a_1/\alpha_1, a_2/\alpha_2, \cdots, a_m/\alpha_m]\ .$$

In general the number of the maximal free occurrences may be greater than m since the expressions α_i occur possibly more than once in the body e. The right-hand side represents an expression obtained from e by replacing all the occurrences of α_j with a_j simultaneously.

The *maximal free occurrence* of an expression with respect to the lambda variable x is the maximal part of an expression that does not depend on x. (See Chapter 6 for the formal definition.) It is a simple extension of the *free variable* not bound by x, while the occurrence of such a variable is in fact the *minimal* free occurrence. Consider a lambda expression

$$\lambda x.\ (+\ y\ (-\ (+\ y\ x)\ (-\ y\ z)))$$

where + and − are constants, and x, y, and z are variables. There are four free occurrences of variables; three y's and one z. These are minimal as described above. Since the expression $(-\ y\ z)$ does not depend on x and is not part of any larger free occurrences of expressions, it occurs maximal free. Similarly two occurrences of the expression $(+\ y)$ are maximal free occurrences. Hence the lambda expression has three maximal free occurrences of two expressions $(+\ y)$ and $(-\ y\ z)$. We may choose either one of

$\alpha_1=(+\,y)$, $\alpha_2=(-\,y\,z)$, and $\Phi\,a_1\,a_2\,x=a_1\,(-\,(a_1\,x)\,a_2)$,

or

$\alpha_1=(-\,y\,z)$, $\alpha_2=(+\,y)$, and $\Phi\,a_1\,a_2\,x=a_2\,(-\,(a_2\,x)\,a_1)$.

The lambda expression becomes

$$\Phi\,(+\,y)\,(-\,y\,z)$$

and

$$\Phi\,(-\,y\,z)\,(+\,y)$$

respectively.

As this example illustrates, the transformation does not define the super-combinator uniquely, since the parameters $a_1, a_2, \cdots, a_m$ are allowed to appear in any order. However, if we consider that the expression $\lambda x.e$ is located in the body of another lambda expression as in

$$\lambda x'.\ \cdots\ (\lambda x.e)\ \cdots\ ,$$

the arguments of Φ dependent on x' should be moved to the last part of the argument list, say $\alpha_{k+1}, \cdots, \alpha_m$, $(k \le m)$. This leads to making the term $\Phi\alpha_1\alpha_2\cdots\alpha_k$ be a maximal free term with respect to x', and reduces the number of parameters of the super-combinator Φ' for this lambda expression to the minimum. If this strategy had not been taken for the right-hand side of a function f in Section 4.2

$$f \equiv \lambda x.\,\lambda y.\,\lambda z.\,IF\ (>\,z\ y)\ (f\ (f\ y\ z\ (-\,x\ 1))\ (f\ z\ x\ (-\,y\ 1))\ (f\ x\ y\ (-\,z\ 1)))\ y\ ,$$

a set of super-combinators

$$\phi'_z\ b_1\,b_2\,b_3\,b_4\,b_5\,b_6\,z\ =\ IF\ (>\,z\ b_1)\ (f\ (b_3\,z\ b_2)\ (f\ z\ b_4\,b_5)\ (b_6\,(-\,z\ 1)))\ b_1$$
$$\phi'_y\ b_1\,b_2\,b_3\,y\ =\ \phi'_z\ y\ b_1\ (f\ y)\ b_2\ (-\,y\ 1)\ b_3$$
$$\phi'_x\ x\ =\ \phi'_y\ (-\,x\ 1)\ x\ (f\ x)$$

would be derived. The optimized super-combinators illustrated in Section 4.2 were

$$\phi_z\ a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,z\ =\ IF\ (>\,z\ a_6)\ (f\ (a_5\,z\ a_2)\ (f\ z\ a_1\,a_4)\ (a_3\,(-\,z\ 1)))\ a_6$$
$$\phi_y\ a_1\,a_2\,y\ =\ a_1\ (a_2\,y)\ (-\,y\ 1)\ (f\ y)\ y$$
$$\phi_x\ x\ =\ \phi_y\ (\phi_z\ x\ (-\,x\ 1))\ (f\ x)\ .$$

Note that the combinator ϕ'_y has more arguments than ϕ_y. The choice of parameter order is essential to obtain super-combinators with fewer arguments. Such effort has been made in part to minimize the number of stack operations when reducing the combinator expression.

As a summary of the idea of the super-combinator is that it is based on

- extracting maximal free occurrences of expression of every lambda expression as arguments of a global function, i.e., a super-combinator,
- defining the global function with corresponding number of parameters, i.e., the number of maximal free expressions plus the number of lambda variables in the original lambda expression, and
- choosing the parameter order so that the total number of parameters of the super-combinators for a program should be as few as possible.

It should be noted that the transformation based only on the first two rules generates super-combinators that share the best property of full laziness. The last principle is provided for efficiency. The number of arguments does nevertheless increase as the nesting level of lambda expressions becomes deeper. All of the combinator reduction schemes presented in Chapter 4 use the stack to pass the arguments to combinators. This suggests that many operations on the stack should be taken in reduction of super-combinators and they may cause the loss of efficiency.

In this chapter we present a restricted class of applicative expressions, which we call *fully lazy normal form*. The fully lazy normal form is more general than the form of super-combinators in the sense that the latter is a special instance of the former. Evaluation of the program of the fully lazy normal form in an ordinary lazy way results in fully lazy evaluation of the original program. In other words, it enables us to implement full laziness by means of an ordinary lazy evaluator. This is our basic strategy to achieve full laziness. See Chapter 3 for the formal definitions of lazy and fully lazy evaluators.

We shall use a simple functional language shown in Figure 5.1.1[1].

5.2. Definition

The basic idea of defining a sublanguage of Figure 5.1.1 is to make a restriction on the form of expressions. The function may contain local definitions in a restricted way, while the super-combinator

[1] We avoid to use λ in the source language because it is used in the description of the semantics. We use **fn** instead.

Syntactic domains

$b \in$ **Bas**	basic values
$x \in$ **Ide**	identifiers
$e \in$ **Exp**	expressions

Abstract syntax

$e ::= b \mid x \mid e\ e \mid$ **fn** $x.\ e \mid$
e **where** $x=e$ **and** $\cdots$ **and** $x=e \mid$
e **whererec** $x=e$ **and** $\cdots$ **and** $x=e$

Semantic domains

$\mathbf{B}$	basic values
$\mathbf{E} = [\mathbf{B} + \mathbf{F}]_+$	expressible values
$\mathbf{F} = \mathbf{D} \to \mathbf{E}$	functions
$\mathbf{D} = \mathbf{E}$	denotable values
$\mathbf{U} = \mathbf{Ide} \to \mathbf{D}_+$	environments

Semantic functions

B : **Bas** $\to$ **B**	(unspecified)
E : **Exp** $\to$ **U** $\to$ **E**	

$E[b]\rho = B[b]$
$E[x]\rho = \rho[x]$
$E[e_0\ e_1]\rho = (E[e_0]\rho)(E[e_1]\rho)$
$E[$**fn** $x.\ e_0]\rho = \lambda\delta.E[e_0](\rho+<x \to \delta>)$
$E[e_0$ **where** $x_1=e_1$ **and** $\cdots$ **and** $x_n=e_n]\rho = E[e_0]\rho'$
where $\rho'=\rho+<x_1 \to E[e_1]\rho>+ \cdots +<x_n \to E[e_n]\rho>$
$E[e_0$ **whererec** $x_1=e_1$ **and** $\cdots$ **and** $x_n=e_n]\rho = E[e_0]\rho'$
where $\rho'=\rho+<x_1 \to E[e_1]\rho'>+ \cdots +<x_n \to E[e_n]\rho'>$

Notations

Domain construction operator + stands for the disjoint sum.
For any domain **X**, $\mathbf{X}_+=\mathbf{X}+\{\mathbf{err}\}$.
For an environment ρ, $\rho+<x \to \delta>$ denotes
$\lambda y.$**if** $x=y$ **then** δ **else** $\rho[y]$.

Initial Environment

The initial environment ρ_0 satisfies $\rho_0[x] \neq$ **err** for pre-defined identifiers x.

Figure 5.1.1. Denotational Specification of a Simple Functional Language

must not. The syntax rules and a context condition of the fully lazy normal form are shown in Figure 5.2.1. The semantic function E of Figure 5.1.1 is supposed to be applied to the new syntactic category e' as well as e. Note that **where**-clauses disappear in the fully lazy form. We shall explain the reason in Chapter 6. Local definitions by **whererec**-clauses may appear only in the outermost expression, i.e., the program, or in the body of **fn**-abstraction. That is, the program of the fully lazy normal form is of either form

Syntax

e ::= e' | e' **whererec** x=e' **and** $\cdots$ **and** x=e'
e' ::= b | x | e' e' | **fn** x. e

Context condition

Expression e does not contain any free occurrences of compound expressions.

Figure 5.2.1. Fully Lazy Normal Form

e_0

or

e_0 **whererec** x_1=e_1 **and** $\cdots$ **and** x_n=e_n .

The expression in general is composed of primitive elements b from **Bas** and x from **Ide**, and functions of the form

fn x. e_0

or

fn x. e_0 **whererec** x_1=e_1 **and** $\cdots$ **and** x_n=e_n .

We shall develop an algorithm in Chapter 6 for transforming any expressions into expressions of the fully lazy normal form that satisfy the context condition.

It should be noted that expressions composed of combinators or super-combinators are of the fully lazy normal form. The combinator expression uses only the first kind of the function that contains no local definitions.

5.3. Relation with combinators

Reduction of a combinator expression achieves fully lazy evaluation of the source program. Several implemention methods of the evaluator have been presented in Chapter 4. The combinators selected by Turner [Turner79] are so primitive that many execution steps are needed to reduce the combinator expression. Hughes [Hughes82] improves this by generating super-combinators specifically chosen for the source program. It has been observed from experimental results in Section 4.6 that the super-combinator code runs several times faster than Turner's code.

In order to compare the fully lazy normal form with super-combinators, consider the function f presented in Sections 4.2 and 5.1. We may rewrite the definition as[2]

$$
\begin{aligned}
f \;=\; & \textbf{fn}\, x.\, (\\
& \quad \textbf{fn}\, y.\, (\textbf{fn}\, z.\, IF\ (>\ z\ y)\ (f\ (b_3\ z\ a_2)\ (f\ z\ x\ b_2)\ (b_1\ (-\ z\ 1)))\ y\ ) \\
& \qquad \textbf{whererec}\ \{b_1 = (a_1\ y)\ \textbf{and}\ b_2 = (-\ y\ 1)\ \textbf{and}\ b_3 = (f\ y)\}\ ) \\
& \quad \textbf{whererec}\ \{a_1 = (f\ x)\ \textbf{and}\ a_2 = (-\ x\ 1)\}\ .
\end{aligned}
$$

It should be observed that

- maximal free occurrences of expressions in the original expression have been moved out of inner scope as in the case of super-combinators, but free variables remain as they are, and
- the structure of **fn** nesting is kept unchanged.

It is obvious that the right-hand side is syntactically correct and is consistent with the context condition of the fully lazy normal form.

The fully lazy normal form allows functions with local declarations, while it is not the case with the super-combinator approach. This reduces the number of stack operations for passing arguments from one combinator to another. To see the difference between these approaches applied to actual implementation, we have made a few experiments. It is of course difficult to discuss the comparative merits in general because thier computational models are different. It would be worthwhile, however, to give experimental results on the timings using the evaluators coded with the aim of efficient implementation on conventional computers. The code based on the *fully lazy functional machine*(FLFM) generated from expressions of the fully lazy normal form has been used. Chapter 7 provides the description of FLFM. The results for the code FC[SC] based on the super-combinator are reproduced from Section 4.6.

Table 5.3.1 shows the timings for evaluation of (f 0 30 60) on the Melcom 70/250. See also Experiments 1 and 4 in Section 4.6. The results of the *nFib* benchmark are shown in Table 5.3.2.

It can be expected from these results that the code based on FLFM for the fully lazy normal form gives a greater efficiency to implementation of functional languages on conventional computers.

---

[2] We use **fn** here instead of λ. See the footnote in Section 5.1. Functions written in upper case letters as *IF*, *HEAD* and *TAIL*, and prefix operators like = and – are taken as elements of Ide. These are pre-defined functions defined in the initial environment ρ_0. We do not specify the concrete syntax of the language in this chapter; we insert symbols { and } or indent where-clauses to clarify textual scope.

Table 5.3.1: Run-time for (*f* 0 30 60)

Code		FC[SC]	FLFM
Run-time in msec.	curried	1572	980
	(ratio)	(1.60)	(1)
	uncurried	1060	586
	(ratio)	(1.81)	(1)

Table 5.3.2: Number of function calls per second by *nFib* benchmark

Code	FC[SC]	FLFM
Function calls per second	2447	3588
(ratio)	(1)	(1.47)

As another example, we consider the function *el* by which Hughes [Hughes82] explains the motivation of using the super-combinator. The function *el* selects the n-th element of a linear list s:

$$el = \mathbf{fn}\ n.\ (\mathbf{fn}\ s.\ IF\ (=n\ 1)\ (HEAD\ s)\ (el\ (-n\ 1)\ (TAIL\ s)))\ .$$

We can derive super-combinators ϕ_n and ϕ_s by Hughes' method as

$$el\ =\ \phi_n$$

where

$$\phi_n\ n\ =\ \phi_s\ (IF\ (=n\ 1))\ (el\ (-n\ 1))$$

and

$$\phi_s\ a\ b\ s\ =\ a\ (HEAD\ s)\ (b\ (TAIL\ s))\ .$$

We assume here that the function variable *el* is global as we have done in the case of f. In addition to the problem of super-combinators caused by the increase in the number of operations for passing arguments, it remains open how to compile recursive definitions into what kind of combinators. For example, the function *el* might be expressed using the fixed-point combinator Y as

$$el = \mathbf{Y}\ (\mathbf{fn}\ el.\ \phi\ (IF\ (=n\ 1))\ (el\ (-n\ 1)))$$

However, when more than one recursive functions are included, it becomes difficult to express the combinator code in a form from which the original recursive definition can be presumed. Hughes proposes an

approach to this problem that uses *graphical combinators* which is similar to ours, but gives no credit for the efficiency. The fully lazy normal form assumes that the recursive definition is properly treated by a primitive operation of the evaluator. By rewriting the above definition, we get an expression of the fully lazy normal form:

$$el = \textbf{fn}\ n.\ (\ \textbf{fn}\ s.\ a\ (HEAD\ s)\ (b\ (TAIL\ s))\ )$$
$$\quad \textbf{whererec}\ \{\ a{=}IF\ (=n\ 1)\ \textbf{and}\ b{=}el\ (-n\ 1)\ \}\ .$$

5.4. Evaluation of expressions

In the fully lazy scheme, every expression is evaluated at most once, while only every argument of a function is evaluated at most once in ordinary lazy evaluation with a call-by-need [Wadsworth71] or a call-by-delayed-value [Vuillemin74] mechanism. To illustrate the difference between them, we take a function *snd* derived from *el*. The function *snd* that gives the second element of a list can be defined by instantiating the function *el* with the first argument:

$$snd = el\ 2\ .$$

We trace first the case of using the straightforward definition. Evaluation of the right-hand side proceeds as[3]

$$el\ 2 \rightarrow$$
$$\quad \textbf{fn}\ s.\ (IF\ (=\nu_1\ 1)\ (HEAD\ s)\ (el\ (-\nu_1\ 1)\ (TAIL\ s))\ \textbf{where}\ \nu_1{=}2\ )$$

No more computaion proceeds unless the argument for s is supplied. Assume that a list s_1 is given:

$$snd\ s_1$$
$$\rightarrow IF\ (=\nu_1\ 1)\ (HEAD\ \sigma_1)\ (el\ (-\nu_1\ 1)\ (TAIL\ \sigma_1))\ \textbf{where}\ \{\ \sigma_1{=}s_1\ \textbf{and}\ \nu_1{=}2\ \}$$
$$\rightarrow IF\ \textbf{false}\ (HEAD\ \sigma_1)\ (el\ (-\nu_1\ 1)\ (TAIL\ \sigma_1))\ \textbf{where}\ \{\ \sigma_1{=}s_1\ \textbf{and}\ \nu_1{=}2\ \}$$
$$\rightarrow IF\text{-}FALSE\ (HEAD\ \sigma_1)\ (el\ (-\nu_1\ 1)\ (TAIL\ \sigma_1))\ \textbf{where}\ \{\ \sigma_1{=}s_1\ \textbf{and}\ \nu_1{=}2\ \}$$
$$\rightarrow el\ (-\nu_1\ 1)\ (TAIL\ \sigma_1)\ \textbf{where}\ \{\ \sigma_1{=}s_1\ \textbf{and}\ \nu_1{=}2\ \}$$
$$\rightarrow (\ IF\ (=\nu_2\ 1)\ (HEAD\ \sigma_2)\ (el\ (-\nu_2\ 1)\ (TAIL\ \sigma_2))$$
$$\quad \textbf{where}\ \{\ \sigma_2{=}TAIL\ \sigma_1\ \textbf{and}\ \nu_2{=}(-\nu_1\ 1)\ \}\ )\ \textbf{where}\ \{\ \sigma_1{=}s_1\ \textbf{and}\ \nu_1{=}2\ \}$$
$$\rightarrow (\ IF\ (=\nu_2\ 1)\ (HEAD\ \sigma_2)\ (el\ (-\nu_2\ 1)\ (TAIL\ \sigma_2))$$
$$\quad \textbf{where}\ \{\ \sigma_2{=}TAIL\ \sigma_1\ \textbf{and}\ \nu_2{=}1\ \}\ )\ \textbf{where}\ \{\ \sigma_1{=}s_1\ \textbf{and}\ \nu_1{=}2\ \}$$
$$\rightarrow IF\ \textbf{true}\ (HEAD\ \sigma_2)\ (el\ (-\nu_2\ 1)\ (TAIL\ \sigma_2))\ \textbf{where}\ \{\ \sigma_2{=}TAIL\ \sigma_1\ \textbf{and}\ \nu_2{=}1\ \}$$

[3] We shall write down the computational process using a similar notation as the source language; expressions bound to parameters are represented by where- or whererec-clauses with generated fresh variables.

where $\{\sigma_1=s_1$ **and** $\nu_1=2\}$

$\rightarrow$ *IF-TRUE* (*HEAD* σ_2) (*el* $(-\nu_2\ 1)$ (*TAIL* σ_2)) **where** $\{\sigma_2$=*TAIL* σ_1 **and** $\nu_2=1\}$

where $\{\sigma_1=s_1$ **and** $\nu_1=2\}$

$\rightarrow$ (*HEAD* σ_2 **where** $\{\sigma_2$=*TAIL* σ_1 **and** $\nu_2=1\}$) **where** $\{\sigma_1=s_1$ **and** $\nu_1=2\}$

When the function *snd* is applied to another list s_2 after (*snd* s_1) is evaluated, similar steps are necessarily taken. However, since the first argument of *el* is common to both (*snd* s_1) and (*snd* s_2), some part of computation is made to be done only once. This is the very purpose of using the fully lazy normal form.

If we use the definition of *el* written in the fully lazy normal form, *snd* can be defined as

snd = *el* 2

$\rightarrow$ **fn** *s*. α_1 (*HEAD* *s*) (β_1 (*TAIL* *s*))

whererec $\{\alpha_1$=*IF* $(=\nu_1\ 1)$ **and** β_1=*el* $(-\nu_1\ 1)$ **and** $\nu_1=2\}$.

Evaluation of (*snd* s_1) proceeds as follows:

snd s_1

$\rightarrow \alpha_1$ (*HEAD* σ_1) (β_1 (*TAIL* σ_1)) **whererec** $\sigma_1=s_1$

whererec $\{\alpha_1$=*IF* $(=\nu_1\ 1)$ **and** β_1=*el* $(-\nu_1\ 1)$ **and** $\nu_1=2\}$

$\rightarrow \alpha_1$ (*HEAD* σ_1) (β_1 (*TAIL* σ_1)) **whererec** $\sigma_1=s_1$

whererec $\{\alpha_1$=*IF-FALSE* **and** β_1=*el* $(-\nu_1\ 1)$ **and** $\nu_1=2\}$

$\rightarrow$ *IF-FALSE* (*HEAD* σ_1) (β_1 (*TAIL* σ_1)) **whererec** $\sigma_1=s_1$

whererec $\{\alpha_1$=*IF-FALSE* **and** β_1=*el* $(-\nu_1\ 1)$ **and** $\nu_1=2\}$

$\rightarrow\beta_1$ (*TAIL* σ_1) **whererec** $\sigma_1=s_1$

whererec $\{\alpha_1$=*IF-FALSE* **and** β_1=*el* $(-\nu_1\ 1)$ **and** $\nu_1=2\}$

The first term β_1 becomes

β_1 = **fn** *s*. α_2 (*HEAD* *s*) (β_2 (*TAIL* *s*))

whererec $\{\alpha_2$=*IF* $(=\nu_2\ 1)$ **and** β_2=*el* $(-\nu_2\ 1)$ **and** $\nu_2=(-\nu_1\ 1)\}$

and

β_1 (*TAIL* σ_1)

$\rightarrow \alpha_2$ (*HEAD* σ_2) (β_2 (*TAIL* σ_2)) **whererec** σ_2=*TAIL* σ_1

whererec $\{\alpha_2$=*IF* $(=\nu_2\ 1)$ **and** β_2=*el* $(-\nu_2\ 1)$ **and** $\nu_2=(-\nu_1\ 1)\}$

$\rightarrow \alpha_2$ (*HEAD* σ_2) (β_2 (*TAIL* σ_2)) **whererec** σ_2=*TAIL* σ_1

whererec $\{\alpha_2$=*IF* $(=\nu_2\ 1)$ **and** β_2=*el* $(-\nu_2\ 1)$ **and** $\nu_2=1\}$

$\rightarrow \alpha_2$ (*HEAD* σ_2) (β_2 (*TAIL* σ_2)) **whererec** σ_2=*TAIL* σ_1

whererec $\{\alpha_2$=*IF-TRUE* **and** β_2=*el* $(-\nu_2\ 1)$ **and** $\nu_2=1\}$

$\rightarrow$ *IF-TRUE* (*HEAD* σ_2) (β_2 (*TAIL* σ_2)) **whererec** σ_2=*TAIL* σ_1

whererec { α_2=*IF-TRUE* **and** β_2=*el* (− ν_2 1) **and** ν_2=1 }

→ *HEAD* σ_2 **whererec** σ_2=*TAIL* σ_1

whererec { α_2=*IF-TRUE* **and** β_2=*el* (− ν_2 1) **and** ν_2=1 }

We have thus obtained the result of (*snd* s_1). At the same time, we have a new version of the function *snd*:

snd = **fn** *s*. α_1 (*HEAD s*) (β_1 (*TAIL s*))

whererec { α_1=*IF-FALSE*

and β_1 = **fn** *s*. α_2 (*HEAD s*) (β_2 (*TAIL s*))

whererec { α_2=*IF-TRUE* **and** β_2=*el* (− ν_2 1) **and** ν_2=1 }

and ν_1=2 }

All the terms dependent on *n* have already been evaluated. When (*snd* s_2) is to be evaluated, only the necessary computational steps dependent on s_2 have to be taken. We have thus attained full laziness in evaluating both the terms (*snd* s_1) and (*snd* s_2) in a program.

We shall deal with an algorithm for translating any expression into the fully lazy normal form in Chapter 6.

Chapter 6

Lambda-hoisting

Lambda-hoisting is a technique for transforming functional programs into the fully lazy normal form which has been defined in Chapter 5. The proposed method has a great advantage in that efficient code for conventional computers can be generated from the transformed program. In this chapter the basic idea of lambda-hoisting is described with remarks on similar techniques, and a simple algorithm is presented in a formal way.

6.1. Transformation to the fully lazy normal form

Compilation techniques for lazy functional languages have been studied by many researchers. Turner [Turner79,81a] proposes a novel scheme of generating *combinator expressions* as object code, and applies it to implement several functional languages [Turner76,81b,85]. Programs are compiled into expressions consisting of only pre-defined combinators. Hughes [Hughes82,84] generalizes this idea to generate similar code using *super-combinators* that are dependent on the source program and produced during compilation. In Chapter 4 of this thesis, we have described these methods in detail and claimed that the super-combinator approach gains in running time by a factor of two over the combinator method on conventional computers. As a different approach to compilation of lazy languages, modified versions of the classical *SECD machine* [Henderson80] adapted for lazy evaluation are used [Henderson83].

Combinator code, including one using super-combinators, is usually represented by a graph and evaluation is taken by an interpreter that performs graph reduction. Such an evaluation method differs from the way the code of usual compiler languages runs to evaluate expressions. It is also possible, however, to generate fixed-code for conventional computers that evaluates combinator expressions [Jones82, Takeichi84,85] (See Section 4.5.). The fixed program thus obtained can be considered as a program coded for a lazy SECD machine. Hence, there is no essential difference between two approaches at least in regard to lazy evaluation, though they seem quite different at first sight.

However, there remains a great difference. The combinator approach enjoys *full laziness* in its nature. Full laziness is the property that every expression is evaluated at most once after the variables in it

have been bound [Hughes84] (See Chapter 3). This property is not found in the compiler for a lazy SECD machine [Henderson83] or in the *lambda-lifting* algorithm [Johnsson85] for generating conventional machine code.

In the rest of this chapter, we present a technique called *lambda-hoisting* for transforming programs into ones convenient for fully lazy evaluation, i.e., of the *fully lazy normal form* which has been introduced in Chapter 5. Evaluation of the transformed program in the ordinary lazy way [Friedman76, Henderson76] results in fully lazy evaluation of the original program. We use a simple functional language shown in Figure 6.1.1[1].

The basic idea of lambda-hoisting is to transform source programs into programs consisting of a more general form of functions than super-combinators. The resulting function may contain local definitions in a restricted way, while the super-combinator must not. The *fully lazy normal form* is defined as in Figure 6.1.2. It is a sublanguage of the language of Figure 6.1.1. A program of the fully lazy normal form should be either

$$e_0$$

or

$$e_0 \textbf{ whererec } x_1=e_1 \textbf{ and } \cdots \textbf{ and } x_n=e_n \ .$$

The **fn**-abstraction[2] has the form

$$\textbf{fn } x.\ e_0$$

or

$$\textbf{fn } x.\ e_0 \textbf{ whererec } x_1=e_1 \textbf{ and } \cdots \textbf{ and } x_n=e_n \ .$$

As explained in Chapter 5, expressions $e_1, \cdots, e_n$ in the last form are *maximal occurrences* of expressions dependent on x in the original **fn**-body, and e_0 is obtained by replacing those occurrences by $x_1, \cdots, x_n$, correspondingly. We shall develop an algorithm for transforming any expression into an expression of the fully lazy normal form that satisfies the condition.

---

[1] The language is the same one in Chapter 5. Figures 6.1.1 and 6.1.2 are reproduced from Figures 5.1.1 and 5.2.1, respectively.

[2] Recall that we are using **fn** rather than λ in our source language. Accordingly, the term *lambda-abstraction* should be called **fn**-*abstraction* here.

Syntactic domains

$b \in$ **Bas**	basic values
$x \in$ **Ide**	identifiers
$e \in$ **Exp**	expressions

Abstract syntax

$e ::= b \mid x \mid e\ e \mid$ **fn** $x.\ e \mid$
e **where** $x=e$ **and** $\cdots$ **and** $x=e \mid$
e **whererec** $x=e$ **and** $\cdots$ **and** $x=e$

Semantic domains

$\mathbf{B}$	basic values
$\mathbf{E} = [\mathbf{B} + \mathbf{F}]_+$	expressible values
$\mathbf{F} = \mathbf{D} \to \mathbf{E}$	functions
$\mathbf{D} = \mathbf{E}$	denotable values
$\mathbf{U} = \mathbf{Ide} \to \mathbf{D}_+$	environments

Semantic functions

B : **Bas** $\to$ **B** (unspecified)
E : **Exp** $\to$ **U** $\to$ **E**

$E[b]\rho = B[b]$
$E[x]\rho = \rho[x]$
$E[e_0\ e_1]\rho = (E[e_0]\rho)(E[e_1]\rho)$
$E[$**fn** $x.\ e_0]\rho = \lambda\delta.E[e_0](\rho+<x \to \delta>)$
$E[e_0$ **where** $x_1=e_1$ **and** $\cdots$ **and** $x_n=e_n]\rho = E[e_0]\rho'$
where $\rho'=\rho+<x_1 \to E[e_1]\rho>+ \cdots +<x_n \to E[e_n]\rho>$
$E[e_0$ **whererec** $x_1=e_1$ **and** $\cdots$ **and** $x_n=e_n]\rho = E[e_0]\rho'$
where $\rho'=\rho+<x_1 \to E[e_1]\rho'>+ \cdots +<x_n \to E[e_n]\rho'>$

Notations

Domain construction operator + stands for the disjoint sum.
For any domain **X**, $\mathbf{X}_+=\mathbf{X}+\{\mathbf{err}\}$.
For an environment ρ, $\rho+<x \to \delta>$ denotes
$\lambda y.$**if** $x=y$ **then** δ **else** $\rho[y]$.

Initial Environment

The initial environment ρ_0 satisfies $\rho_0[x] \neq$ **err** for pre-defined identifiers x.

Figure 6.1.1. Denotational Specification of a Simple Functional Language

6.2. Rewriting where-clauses

The first stage of lambda-hoisting is to transform **where**-clauses in the source expression into **whererec**-clauses by renaming variables according to the rules shown in Figure 6.2.1. By doing this, we become free from worries concerning the conflict of identifiers that may be caused when free occurrences of expressions are moved to outside the function from where they originally appear.

Syntax

$e ::= e' \mid e'$ **whererec** $x=e'$ **and** $\cdots$ **and** $x=e'$
$e' ::= b \mid x \mid e'\ e' \mid$ **fn** $x.\ e$

Context condition

Expression e does not contain any free occurrences of compound expressions.

Figure 6.1.2. Fully Lazy Normal Form

Environment for renaming

$\pi \in \mathbf{R} = [\ \mathbf{Ide} \rightarrow \mathbf{Ide}_+\ ]$

Rewriting rules

$R : \mathbf{Exp} \rightarrow \mathbf{R} \rightarrow \mathbf{Exp}$

$R\,[b\,]\pi = b$
$R\,[x\,]\pi = \pi[x\,]$
$R\,[e_0\,e_1]\pi = (R\,[e_0]\pi)(R\,[e_1]\pi)$
$R\,[\mathbf{fn}\ x.\ e_0]\pi = \mathbf{fn}\ x'.\ R\,[e_0](\pi{+}{<}x{\rightarrow}x'{>})$
where x' is a fresh identifier
$R\,[e_0\ \mathbf{where}\ x_1{=}e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n{=}e_n\,]\pi =$
$R\,[e_0]\pi'\ \mathbf{whererec}\ x'_1{=}R\,[e_1]\pi\ \mathbf{and}\ \cdots\ \mathbf{and}\ x'_n{=}R\,[e_n\,]\pi$
where $\pi'{=}\pi{+}{<}x_1{\rightarrow}x'_1{>}{+}\ \cdots\ {+}{<}x_n{\rightarrow}x'_n{>}$
and x'_i are fresh identifiers
$R\,[e_0\ \mathbf{whererec}\ x_1{=}e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n{=}e_n\,]\pi =$
$R\,[e_0]\pi'\ \mathbf{whererec}\ x'_1{=}R\,[e_1]\pi'\ \mathbf{and}\ \cdots\ \mathbf{and}\ x'_n{=}R\,[e_n\,]\pi'$
where $\pi'{=}\pi{+}{<}x_1{\rightarrow}x'_1{>}{+}\ \cdots\ {+}{<}x_n{\rightarrow}x'_n{>}$
and x'_i are fresh identifiers

Notation

For an environment for renaming π, $\pi{+}{<}x{\rightarrow}x'{>}$ denotes
$\lambda y.\mathbf{if}\ x{=}y\ \mathbf{then}\ x'\ \mathbf{else}\ \pi[y\,]$.

Initial Environment

The initial environment π_0 satisfies $\pi_0[x\,]{=}x$ for pre-defined identifiers x.

Figure 6.2.1. Rules for Renaming Identifiers and Rewriting **where**-clauses

Although we do not give a formal definition of *fresh variables* for brevity, the next proposition should be observed.

Proposition

For any $\pi{\in}\mathbf{R}$, $x{\neq}y$ and a fresh variable $x'{\in}\mathbf{Ide}$, $(\pi{+}{<}x{\rightarrow}x'{>})\,[y\,] \neq x'$.

Using this, we can prove that

Lemma

For any $\pi \in \mathbf{R}$ and $\rho, \rho' \in \mathbf{U}$ satisfying $\rho'(\pi[x]) = \rho[x]$,

$E[R[e]\pi]\rho' = E[e]\rho$

holds for any $e \in \mathbf{Exp}$.

Since initial environments $\pi_0 \in \mathbf{R}$ and $\rho_0 \in \mathbf{U}$ satisfy $\rho_0(\pi_0[x]) = \rho_0[x]$, the next theorem holds.

Theorem

$E[R[e]\pi_0]\rho_0 = E[e]\rho_0$ for any program e.

The theorem states that the meaning of the program is preserved through the transformation by the rules in Figure 6.2.1.

6.3. Lexical levels of expressions

The main part of lambda-hoisting is to identify maximal free occurrences of expressions (See Section 6.4). To do so, it is necessary to determine **fn**-variables on which each expression depends. Each **fn**-variable can be identified by the level number, i.e., the number of nested **fn**-abstractions. We assume that the outermost **fn**-variable is assigned the level number 1.

In Hughes' algorithm for finding super-combinators, each compound expression, or *combination* of the form $(e_0\, e_1)$, is assigned the maximum level number of its constituents. It is insufficient for our purpose, however. As it will turn out, it is necessary to assign a set of level numbers to each expression. For a combination, the union of the sets of level numbers for its constituents is assigned. Every constant has the singleton set $\{0\}$, and each variable has $\{0\} \cup \{l\}$ where l is the level of the variable.

We denote the maximum of a set of level numbers

$$\bar{l} = \{l_1, l_2, \cdots, l_n\}$$

by

$$|\bar{l}| = \max\{l_1, l_2, \cdots, l_n\}$$

The rules for assigning the set of level numbers to the expression are shown in Figure 6.3.1. Since lexical levels are computed for expressions transformed by the rules in Section 6.2, the rule for **where**-abstraction is not used in our lambda-hoisting algorithm, though it is included in Figure 6.3.1.

Level numbers

$l \in \mathbf{N}$

Environment for level numbers

$\omega \in \mathbf{L} = [\,\mathbf{Ide} \rightarrow \mathbf{N}_+\,]$

Assignment rules

$L : \mathbf{Exp} \rightarrow \mathbf{L} \rightarrow \mathbf{N} \rightarrow 2^{\mathbf{N}}$

$L[b]\omega l = \{0\}$

$L[x]\omega l = \{0\} \cup \{\omega[x]\}$

$L[e_0\, e_1]\omega l = L[e_0]\omega l \cup L[e_1]\omega l$

$L[\mathbf{fn}\ x.\ e_0]\omega l = L[e_0](\omega + \langle x \rightarrow l+1 \rangle)(l+1) - \{l+1\}$

$L[e_0\ \mathbf{where}\ x_1 = e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n = e_n]\omega l = L[e_0]\omega' l$

where $\omega' = \omega + \langle x_1 \rightarrow l_1 \rangle + \cdots + \langle x_n \rightarrow l_n \rangle$

where $l_i = |\, L[e_i]\omega l \,|$ for $i = 1, \cdots, n$.

$L[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n = e_n]\omega l = L[e_0]\omega' l$

where $\omega' = \omega + \langle x_1 \rightarrow l_1 \rangle + \cdots + \langle x_n \rightarrow l_n \rangle$

where $l_i = |\, L[e_i]\omega' l \,|$ for $i = 1, \cdots, n$.

Notation

For an environment for assignment ω, $\omega + \langle x \rightarrow l \rangle$ denotes

$\lambda y.\mathbf{if}\ x = y\ \mathbf{then}\ l\ \mathbf{else}\ \omega[y]$.

Initial Environment

The initial environment ω_0 satisfies $\omega_0[x] = 0$ for pre-defined identifiers x.

Figure 6.3.1. Rules for Assigning Level Numbers to Expressions

The rule for **fn**-abstractions is worth noting. Assume that **fn** $x.\ e_0$ appears in the context $\omega \in \mathbf{L}$ of the level l. If e_0 contains x and no other variables, we get

$$|\, L[\mathbf{fn}\ x.\ e_0]\omega l \,| = 0$$

from the rule. Hence the combinator has the level 0 in any context and at any level. For example, an occurrence of a combinator

$$\mathbf{S} = \mathbf{fn}\ f\ g\ x.\ f\ x\ (g\ x)$$

is considered as a constant[3].

Consider the case that e_0 contains a variable y of $\omega[y] = l$ as in

$$e_0 = \cdots\ y\ \cdots$$

and

[3] In the extended **fn**-binding **fn** $x_1\ \cdots\ x_n$, we consider that variables $x_1, \cdots, x_n$ are of the same level, say $(l+1)$ if the **fn**-abstraction appears at level l.

fn y. (**fn** x. e_0) .

We have

$$L\,[e_0](\omega+<x\rightarrow l+1>)(l+1) = \{\,0, l, \cdots\,\}$$

and it is concluded that

$$|\,L\,[\textbf{fn}\ x.\ e_0]\omega l\ |\ = l$$

whether x appears in e_0 or not. That is, the function **fn** $x.e_0$ depends on the variable of level l. This example illustrates the reason why we compute the set of lexical levels instead of only the maximal level. It is necessary to find the largest or the second largest of the level numbers assigned to e_0.

Finally, it should be noted that the rule for **whererec**-abstractions is stated using a recursive equation for ω':

$$\omega' = \omega + <x_1\rightarrow |\,L\,[e_1]\omega' l\,|> + \cdots + <x_n\rightarrow |\,L\,[e_n]\omega' l\,|>$$

A question may arise: does the equation have a solution at all? If it does, we need an algorithm to find an ω' satisfying the equation. We answer this by presenting a simple algorithm. This can be used in practice for lambda-hoisting, though it may not be an optimal algorithm. It is based on a simple iteration starting with initial approximations to $|\,L\,[e_i]\omega' l\,|$ and improving them successively.

$l'_i := 0$ for $i = 1, \cdots, n$;
repeat
  $l_i := l'_i$ for $i = 1, \cdots, n$;
  $\omega' := \omega + <x_1\rightarrow l_1> + \cdots + <x_n\rightarrow l_n>$;
  $l'_i := |\,L\,[e_i]\omega' l\,|$ for $i = 1, \cdots, n$;
until $\{\, l_i = l'_i$ for every $i = 1, \cdots, n\,\}$

From the observation that the operations employed are monotonic, and $0 \le l_i \le l$ holds for $i = 1, \cdots, n$, it can be shown that the algorithm terminates.

6.4. Maximal free occurrences

We now define *free occurrences* of combinations. The free occurrence of combinations is a simple extension of the free occurrence of variables that is not bound by an **fn**-abstraction. Note that we are dealing with only expressions transformed by the rules in Section 6.2.

Definition (Free occurrences of combinations)

An occurrence of an expression of the form $(e_0\, e_1)$ is called a *free occurrence* with respect to $\omega \in \mathbf{L}$ and $l \in \mathbf{N}$, if

$$0 \leq | L[e_0]\omega l | < l \ , \ 0 \leq | L[e_1]\omega l | < l \ , \text{ and } | L[e_0\, e_1]\omega l | \neq 0$$

hold.

Since

$$| L[e_0\, e_1]\omega l | = | L[e_0]\omega l \cup L[e_1]\omega l | = | L[e_0]\omega l | \text{ or } | L[e_1]\omega l |$$

from the rules in Figure 6.3.1, the above condition can be restated as: both $|L[e_0]\omega l|$ and $|L[e_1]\omega l|$ are less than l, but at least either of them is greater than 0.

Definition (Maximal free occurrences of combinations)

A free occurrence of a combination $e^* = (e_0\, e_1)$ with respect to $\omega \in \mathbf{L}$ and $l \in \mathbf{N}$ is called *maximal*, if either of following conditions holds.

(1) There is an occurrence of a combination containing e^* as $(e'\ e^*)$ or $(e^*\ e')$, and

$$| L[e^*]\omega l | < | L[e']\omega l |$$

holds.

(2) The occurrence e^* appears as either

fn $x.\ e^*$

e^* **whererec** $x_1 = e_1$ **and** $\cdots$ **and** $x_n = e_n$

or

$x_i = e^*$ in a **whererec**-clause.

6.5. Algorithm

Rules for lambda-hoisting are shown in Figure 6.5.1. In short, maximal free occurrences of combinations are moved outside the original **fn**-body by creating new declarations with fresh variables.

Algorithm

An expression e is transformed into e^* of the fully lazy normal form by

$$<\mu^*, \omega^*, e^*> = H[R[e]\pi_0]\mu_0 0$$

For example,

Level numbers

$l \in \mathbf{N}$

Environment for level numbers

$\omega \in \mathbf{L}$

Declarations of maximal free occurrences of combinations

$\mu \in \mathbf{M} = [\mathbf{N} \to 2^{\mathbf{Dec}}]$

$d \in \mathbf{Dec}$ declarations

$d ::= x=e$

Hoisting rules

$H : \mathbf{Exp} \to \mathbf{M} \to \mathbf{L} \to \mathbf{N} \to [\mathbf{M} \times \mathbf{L} \times \mathbf{Exp}]$

$H[b]\mu\omega l = <\mu, \omega, \lceil b \rceil>$

$H[x]\mu\omega l = <\mu, \omega, \lceil x \rceil>$

$H[e_0\, e_1]\mu\omega l = <\mu^*, \omega^*, e^*>$

let $<\mu'', \omega'', e'_1> = H[e_1]\mu'\omega' l$ where $<\mu', \omega', e'_0> = H[e_0]\mu\omega l$ in

if e'_i $(i=0,1)$ is a maximal free occurrence of a combination w.r.t. ω'' and l,

$\mu^* = \mu'' + <k \to \mu'' k \cup \lceil x' = e'_i \rceil>$, $\omega^* = \omega'' + <x' \to k>$,

and $e^* = \lceil (x'\ e'_1) \rceil$ or $e^* = \lceil (e'_0\ x') \rceil$ for $i=0,1$, respectively,

where $k = |L[e_i]\omega'' l|$ and x' is a fresh identifier

else $\mu^* = \mu''$, $\omega^* = \omega''$ and $e^* = (e'_0\ e'_1)$

$H[\mathbf{fn}\ x.\ e_0]\mu\omega l = <\mu^*, \omega^*, \lceil \mathbf{fn}\ x.\ e_0^* \rceil>$

let $<\mu', \omega', e'_0> = H[e_0](\mu + <l+1 \to \{\}>)(\omega + <x \to l+1>)(l+1)$ in

if $\mu'(l+1) = \{\}$

and e'_0 is a maximal free occurrence of a combination w.r.t. ω' and l,

$\mu^* = \mu' + <k \to \mu' k \cup \lceil x' = e'_0 \rceil>$, $\omega^* = \omega' + <x' \to k>$, and $e^* = \lceil x' \rceil$

where $k = |L[e'_0]\omega' l|$ and x' is a fresh identifier

else $\mu^* = \mu'$, $\omega^* = \omega'$, and $e^* = \lceil e'_0\ \mathbf{whererec}\ \mu'(l+1) \rceil$

$H[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n = e_n]\mu\omega l = H[e_0]\mu_n\omega_n l$

where $\mu_i = \mu'_i + <k \to \mu'_i \cup \lceil x_i = e'_i \rceil>$ and $\omega_i = \omega'_i + <x_i \to k>$

where $<\mu'_i, \omega'_i, e'_i> = H[e_i]\mu_{i-1}\omega_{i-1} l$ and $k = |L[e'_i]\omega'_i l|$

for $i = 1, \cdots, n$, and $\mu_0 = \mu$, $\omega_0 = \omega$

Notations

Tuples in [M×L×Exp] are written as $<\mu, \omega, e>$.

Syntactic elements are quoted by $\lceil$ and $\rceil$.

For a declaration set μ, $\mu + <k \to \sigma>$ denotes

$\lambda l.\mathbf{if}\ j=k\ \mathbf{then}\ \sigma\ \mathbf{else}\ \mu j$

If $\mu l = \{\lceil x_1 = e_1 \rceil, \cdots, \lceil x_n = e_n \rceil\}$, $\lceil e_0\ \mathbf{whererec}\ \mu l \rceil$ denotes

$\lceil e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n = e_n \rceil$

Initial set of declarations

The initial set of declarations μ_0 satisfies $\mu_0 l = \{\}$ for any $l \in \mathbf{N}$.

Figure 6.5.1. Lambda-hoisting Rules for the Fully Lazy Normal Form

fn x. (**fn** y. (+ (− x 1) y))

is transformed into[4]

fn x. (**fn** y. (z y)) **whererec** z=(+ (− x 1)) .

A more realistic example is shown in Figure 6.5.2. This example demonstrates the use of full laziness for eliminating multiple traversals of data structures [Takeichi87]. See Chapter 9 of this thesis.

Source program:

```
fn x.
    (ξ FORK (fn u. TIP (ξ MIN I))
        where ξ = btree x
            whererec
                btree = fn x.
                        (fn g f.
                            IF (ISTIP x) (f (TIPVAL x))
                            (g (btree (LEFT x) g f) (btree (RIGHT x) g f))))
```

Fully lazy normal form:

```
fn x'. (ξ' FORK (fn u'. α)) whererec {ξ' = btree' x' and α = TIP (ξ' MIN I)})
whererec
    btree' = fn x''. (fn g' f'. β2 (f' β1) (g' (β3 g' f') (β4 g' f')))
                whererec {β1 = TIPVAL x'' and β2 = IF (ISTIP x'')
                    and β3 = btree' (LEFT x'') and β4 = btree' (RIGHT x'')}
```

Figure 6.5.2. An Example of Lambda-hoisting

As the function *btree* in the source program has no free variables, it has been moved to the outermost level in the fully lazy form. The reader will have observed from this example how the lambda-hoisting algorithm works.

We believe that the meaning of programs remains unchanged through the transformation. That is,

$$E[e^*]\rho_0 = E[e]\rho_0 \quad \text{where } <\mu^*, \omega^*, e^*> = H[R[e]\pi_0]\mu_0 0$$

Although we have not completed the proof of this *soundness* theorem, it seems possible to prove it with great care in dealing with *strict* functions such as arithmetic operations and predicates. Then the lambda-hoisting transformation preserves the meaning of programs. Evaluation of the resulting program in a lazy way turns out to be fully lazy. Since full laziness implies ordinary laziness by definition, it would be obvious at least intuitively that the transformed program requires no more evaluation steps than the original.

---

[4] The form **fn** x. e **whererec** ··· should read as **fn** x. (e **whererec** ···).

6.6. Remarks

In this chapter we have developed an algorithm for lambda-hoisting. The algorithm presented in the previous section can be considered as a functional program, though there remain some informal descriptions like " $\cdots$ for $i=1,\cdots,n$ ". We have a program written in Lisp for transforming Lispkit Lisp [Henderson80,83] programs into the fully lazy normal form. Programs of the fully lazy normal form are compiled into fixed-code of the *Fully Lazy Functional Machine* [Takeichi86a] (See Chapter 7). The FLFM code is then translated into machine code for conventional computers. We have developed code generators for MC68000, i8086, Melcom 70/250, and Melcom MX2000. As the task of the code generator is a simple macro processing driven by a table, it is easy to generate code for other machines. In Chapter 8, we shall describe a portable compiler developed this way.

The fully lazy normal form relaxes restrictions on the representation of functions; the function body possibly contains local definitions. The super-combinator approach keeps the traditional form based on lambda calculus, and **where**- and **whererec**- clauses are transformed into functional applications, e.g., e_0 **where** $x_1=e_1$ into (**fn** $x_1.e_0)e_1$. Although these are semantically equivalent in our language as well, we have not followed this transformation because the number of parameter passing becomes large for nested function definitions.

The key to the lambda-hoisting technique is to transform programs into ones with *local recursion*. Elimination of non-recursive local declarations by **where**-clauses greatly simplifies the algorithm. A similar compilation technique called *lambda-lifting* [Johnsson85] takes no account of full laziness. Functions expressed by **fn**-abstractions inside another function remain as they are by lambda-hoisting, while all the functions are made global by lambda-lifting. The presence of local functions enables one to instantiate a function to obtain other functions by partial parametrization.

We have presented an example that demonstrates the use of full laziness for eliminating multiple traversals of data structures. Fully lazy evaluation brings unexpected gains in efficiency. More investigation on the novel feature with relation to partial parametrization in functional programming will be found in later chapters.

Chapter 7

Fully lazy functional machine

In order to explore the applicability of full laziness in functional programming, an abstract machine called *Fully Lazy Functional Machine* (FLFM) has been developed. The machine is a variant of the SECD machine suitable for evaluating expressions in a fully lazy way. This chapter describes the structure of FLFM and an algorithm for translating functional programs into FLFM code. Although the FLFM program can be executed using a small interpreter, generation of machine code for conventional computers is expected to gain efficiency. The implementation technique described in this chapter is useful to generate efficient code for a wide class of functional languages. Actual implementation on a variety of machines will be described in Chapter 8.

Hughes [Hughes82,84] introduces the idea of fully lazy evaluation of applicative expressions in relation with combinators. Full laziness implies ordinary laziness in [Henderson76, Friedman76]. Among others, it has an important property that every expression is evaluated at most once, whereas in ordinary lazy evaluation scheme only the expression passed as argument to function is evaluated at most once. Considering the whole computational process for an expression, fully lazy evaluation is optimal in the sense that necessary computation is performed only once and unnecessary computation is not done at all. See Chapter 3 for the detailed discussion.

Hughes also describes an algorithm that translates applicative expressions possibly with lambda abstraction into super-combinators (See Section 4.2). Another transformation technique called *lambda-hoisting* has been developed [Takeichi86b] for generating more efficient code. As described in Chapters 5 and 6, any expression is lambda-hoisted into the *fully lazy normal form*, which uses a more general form of functions than super-combinators. The lambda-hoisting approach deals with expressions of the form

$$e_0 \textbf{ where } x_1 = e_1 \textbf{ and } \cdots \textbf{ and } x_n = e_n$$

and

$$e_0 \textbf{ whererec } x_1 = e_1 \textbf{ and } \cdots \textbf{ and } x_n = e_n$$

in a naive way, while some kind of artificial combinator has to be introduced for the super-combinator method. Our functional machine provides instructions to manipulate these expressions efficiently.

We begin with a review of the fully lazy normal form to clarify our intention to design a functional machine for fully lazy evaluation. Section 7.2 describes how evaluation of the fully lazy normal form proceeds and explains the motivation to design a new functional machine. A machine model for fully lazy evaluation which we call Fully Lazy Functional Machine (FLFM) is defined in Section 7.3. Rules for compiling functional programs to generate FLFM instructions are formulated in Section 7.4. Implementation techniques of FLFM with code generation for conventional computers is the topic of Section 7.5.

7.1. Expressions of the fully lazy normal form

We use a simple functional language shown in Figure 7.1.1 of which formal semantics has been given in Figure 5.1.1. The figure includes the syntax and the context condition of the fully lazy normal form which is extracted from Figure 5.2.1.

Syntactic domains

$b \in \mathbf{Bas}$	basic values
$x \in \mathbf{Ide}$	identifiers
$e \in \mathbf{Exp}$	expressions

Abstract syntax

$e ::= b \mid x \mid e\ e \mid \mathbf{fn}\ x\ x \cdots x.\ e \mid$
$e\ \mathbf{where}\ x=e\ \mathbf{and} \cdots \mathbf{and}\ x=e \mid$
$e\ \mathbf{whererec}\ x=e\ \mathbf{and} \cdots \mathbf{and}\ x=e$

Semantic domains

$\mathbf{B}$	basic values
$\mathbf{E} = [\mathbf{B} + \mathbf{P} + \mathbf{F}]_+$	expressible values
$\mathbf{P} = \mathbf{E} \times \mathbf{E}$	pairs
$\mathbf{F} = \mathbf{D} \rightarrow \mathbf{E}$	functions
$\mathbf{D} = \mathbf{E}$	denotable values
$\mathbf{U} = \mathbf{Ide} \rightarrow \mathbf{D}_+$	environments

Syntax of the Fully Lazy Normal Form

$e ::= e' \mid e'\ \mathbf{whererec}\ x=e'\ \mathbf{and} \cdots \mathbf{and}\ x=e'$
$e' ::= b \mid x \mid e'\ e' \mid \mathbf{fn}\ x.\ e$

Context condition of the Fully Lazy Normal Form

Expression e does not contain any free occurrences of compound expressions.

Figure 7.1.1. A Simple Functional Language and the fully lazy normal form

The domain **E** of expressible values is now extended to include **P** which represents a domain of pairs produced by a primitive constructor of data structure. Data structure of some kind should be included in any functional language to solve practical problems. We are taking a simple data structure into account to write programs that deal with *lists* and *trees*. However, interpretation of the pair structure depends on the language implemented on FLFM. What we need here for the design of FLFM is to distinguish values in **P** with ones in **B**. The value in **B** is considered as a *scalar* that is represented by a single entity in a FLFM word, and the value in **P** by a pair of FLFM words possibly with a pointer to it. It should be noted that we have not introduced any syntactic domain for expressions which represent values in **P**. We assume here that only the function *cons* can produces these values, and therefore the value of a combination

$$(cons\ e_1\ e_2)$$

lies in **P** where e_1 and e_2 are any expressions.

Another extension is the form of **fn**-abstraction expressions. An expression

$$\mathbf{fn}\ x_1\ x_2\ \cdots\ x_k.\ e$$

represents a functional value in **F**. Although we have mentioned in Section 6.3 that the variables x_1, x_2, $\cdots$, x_k are of the same level, this form is an abbreviation of

$$\mathbf{fn}\ x_1.\ (\mathbf{fn}\ x_2.\ (\cdots(\mathbf{fn}\ x_k.\ e)))\ ,$$

and represents a *curried* function. The function thus defined may be partially parametrized by fewer, say $l<k$, arguments to obtain a function returning the value of e when applied to $k-l$ arguments.

7.2. Design principles

In this section we first review how an expression is evaluated mechanically using a few devices. The design principles of our machine model, called *Fully Lazy Functional Machine* (FLFM), is then explained. The machine can be considered as a variant of the *SECD* machine [Henderson80, Landin69] specifically designed for our purpose.

Evaluation

Let us look at how evaluation of an expression proceeds. An expression of the fully lazy normal form is either

(1) an expression $b \in$ **Bas** representing a value in **B**,

(2) a variable $x \in$ **Ide**,

(3) a combination

$$(e'_0\, e'_1 \cdots e'_n)$$

where each e'_i is an expression of the form (1), (2), (3), or (4),

(4) an expression of the form

$$\mathbf{fn}\ x_1 \cdots x_k.e$$

representing a value in **F** where e is any expression of the fully lazy normal form,

or

(5)

$$e'_0\ \mathbf{whererec}\ x_1{=}e'_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_m{=}e'_m$$

where each e'_i is an expression of the form (1), (2), (3), or (4).

An expression of the class (1) represents a *constant* in the sense that it has been completely reduced to a value that cannot be simplified any more. In other words, it remains unchanged if ever evaluation is taken for it. Hence the evaluation of such an expression is trivial; do nothing. Similarly, every expression of the class (4) represents a function in **F** and is also treated as a constant.

Evaluation of a variable x of the class (2) depends on the static context where it appears in the program and on the dynamic behavior of the program. Bindings of variables with values are established in the course of evaluation, and they are kept in the *environment* from which we can find the value of the variable. The value associated to x may be further reducible to a simpler form, because its evaluation may have been suspended. Consequently the evaluation of the variable proceeds by first taking the value out of the variable in the environment, and then evaluating that value. The environment is affected when a function is invoked to obtain the value of a combination, or an expression with local definition is evaluated.

Suppose that an expression of the class (3)

$$(e'_0\, e'_1 \cdots e'_n)$$

is forced to be evaluated. FLFM maintains the *stack* and the *environment* as follows. Let α_i be the

suspended value corresponding to each e'_i in the current environment. That is, α_i contains the expression e'_i and everything that is necessary to evaluate it, which will be explained shortly. The machine pushes the value $\alpha_n, \cdots, \alpha_1, \alpha_0$ onto the stack in this order. and evaluates α_0 at the top of the stack. Then it *applies* the result to arguments $\alpha_1, \cdots, \alpha_n$ on the stack. The values on the stack might be *constants* or *closures*. A closure is a pair consisting of the code for evaluating e'_i and the current environment.

For simplicity, assume that e'_0 is an expression representing a function

$$\mathbf{fn}\, x_1 \cdots x_k.\, e.$$

In this case the value α_0 at the top of the stack is a constant and no more evaluation is necessary. When applying the function α_0 to the arguments $\alpha_1, \cdots, \alpha_n$, the value α_0 on the stack is removed, and the arguments left on the stack are used to create a new environment for the function body e. The new environment is obtained by extending the current environment. The arguments are moved from the stack to extend the environment in such a way that α_i is bound to x_i. If sufficient number of arguments, i.e., k, have been found on the stack, evaluation proceeds to e (Figure 7.2.1).

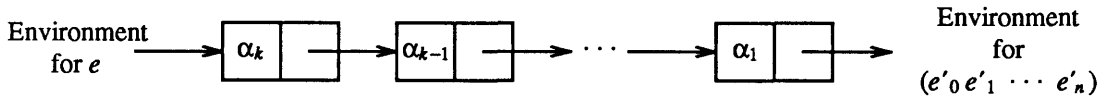


Figure 7.2.1. Evaluation proceeds to the function body e

Otherwise, no further computation can be done and a closure consisting of the code for further computation and the partially filled environment is returned as the result of the original expression (Figure 7.2.2). The closure is shown in dashed boxes in the figure.

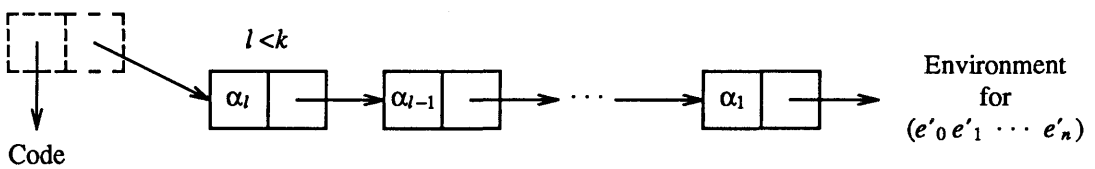


Figure 7.2.2. Evaluation results in a partially parametrized function

Consider next the class (5) of expressions with local recursive definitions:

e'_0 **whererec** $x_1=e'_1$ **and** $\cdots$ **and** $x_m=e'_m$

The evaluation of e'_0 requires a new environment that represents recursive bindings of x_i with the suspended value β_i of e'_i. Evaluation of e'_i should be suspended in the same way as in function arguments, and therefore β_i is a closure unless it is a constant. Recursive definition introduced by the **whererec**-clause produces a circular environment as shown in Figure 7.2.3. Dashed boxes are closures. Note that this form of expression may appear as the outermost expression of the fully lazy normal form, or as a function body. The figure illustrates the latter case where arguments of the function α_i have already been inserted to the environment.

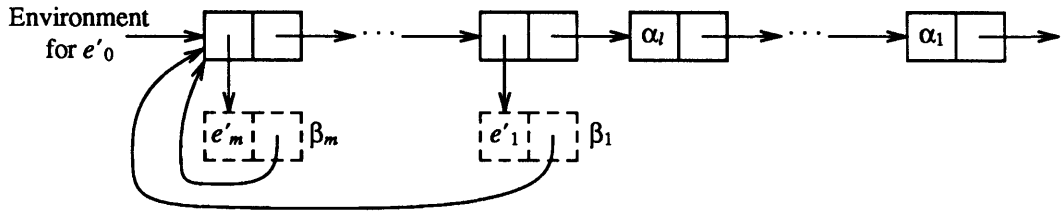


Figure 7.2.3 A circular environment

Finally consider what steers the evaluation. The first request to evaluate, or simplify, an expression is usually made by the user of the program; for example, he wants to print out the value of the expression e. He already knows a representation of the value, i.e., e, because it is written according to the rules of the language. However, what he needs is the *simplest* representation of the value. For an expression $(fac\ 6)$ using the factorial function fac, he prefers 720 to $6\times(fac\ 5)$ or $6\times5\times4\times3\times2\times1$.

Once requested, evaluation proceeds as described above. As we have seen in the case of the class (3) expressions, the head term e'_0 of a combination is always forced to be evaluated when that combination is evaluated. In addition to this case, *strict* functions such as *add*, *eq*, etc., call for evaluation of their arguments. Hence the evaluation process is necessarily recursive.

Values

Values pushed onto the stack and held in the environment are represented as either

(1) an *evaluated value*, which may be a basic value in **B**, a data structure in **P=E×E**, or a function in **F=D→E**,

or

(2) an *unevaluated value* represented by a *closure* consisting of the code and the environment.

For brevity, we simply use the word *value* to mean the *representation of values* in FLFM.

In case of an evaluated value, it should never be evaluated again. As described above, the closure structure is a kind of values corresponding to the suspended expression. It may yield a basic value, a pair, or a function. Once evaluated, every *unevaluated* value becomes an *evaluated* value equivalent to the original. Hence the distinction between evaluated and unevaluated values should be made in FLFM.

To make full use of the call-by-need mechanism, every value held in the environment has to be updated by the result when it is evaluated. This can be attained in FLFM using the information on the kind of value representations just mentioned. The evaluated value may be referenced many times, while the unevaluated one is evaluated only once at the first time when its value is required.

As we have seen in evaluation of a combination, an expression may produce a functional value that is a partially parametrized function as its value. It is also represented by a closure. Even in the case that the program generates a non-functional value as its result, functional values may appear in the course of evaluation. Such functional values may be referenced more than once to be shared by several occurrences of that expression.

Suppose that an expression ψ

$$\psi \equiv \mathbf{fn}\, x_1 \cdots x_l \cdots x_k .\, e$$

appears in the context

$$\theta \equiv (\psi\, e'_1 \cdots e'_l)$$

where $l < k$, and θ appears in

$$\varepsilon_1 \equiv (\theta\, f'_{l+1} \cdots f'_k)$$

and

$\varepsilon_2 \equiv (\theta\ g'_{l+1} \cdots g'_k)$.

Then, an environment structure like Figure 7.2.4 should be made. In the figure, α_i stands for the value of e'_i, β_i for the value of f'_i, and γ_i for the value of g'_i.

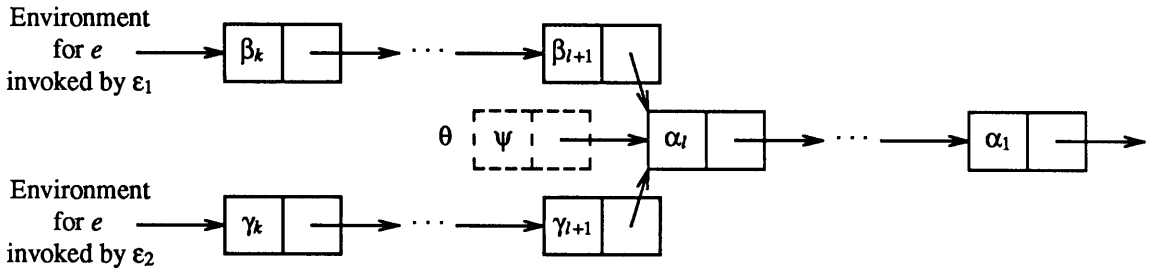


Figure 7.2.4. A partially parametrized function and environments

Note that every argument α_i in an environment might be replaced by the result of evaluation. If we had produced duplicated copies of the list of $\alpha_k, \cdots, \alpha_1$, one for ε_1, and one for ε_2, we could not update the value efficiently. It is concluded that every partially parametrized function should be represented as a value to be shared, and the environment should have a structure as shown in Figure 7.2.4.

7.3. Machine Structure

The FLFM machine consists of four registers S, E, C, and D, each of which holds a list representing the *stack*, the *environment*, the *control* code, or the *dump*, respectively. It should be noted that the SECD model of FLFM is a conceptual one; the stack need not be of the list structure in actual implementation, for example. We will discuss about implementation details in Section 7.5.

We follow the notation used in [Henderson80] to specify the machine by state transition as

$$S\ \ E\ \ C\ \ D\ \ \rightarrow\ \ S'\ \ E'\ \ C'\ \ D'$$

To describe the change of the environment E, we will use the convention that E_i means the i-th link of E, and $*E_i$ the contents, or the value, of the i-th element of E (Figure 7.3.1). Note that arguments passed to functions are moved from the stack S to the environment so that their values are to be referenced as $*E_i$, not as E_i.

We denote a closure consisting of code C and environment E by $[C{:}E]$, and an empty list by ϕ instead of **nil**. The symbol ϕ is also used for a distingushed reference for C, which will be explained later.

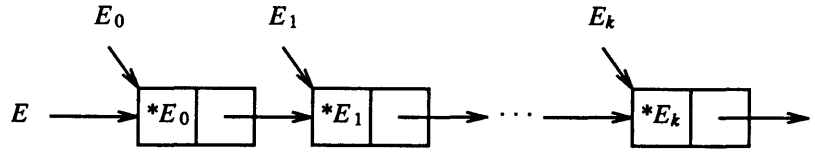


Figure 7.3.1. The environment structure

As mentioned earlier, primitive functions *if*, *eq*, *add*, etc., are considered as functions defined in the standard environment and treated as *global* objects. Each of them has its own evaluation rule which is different from functions defined in programs. We put them aside for the present and confine ourselves to general FLFM instructions.

Load Instructions

Load Constant

$$S \quad E \quad (CONST\ b\ .\ C)\ \ D \ \rightarrow\ (b\ .\ S)\ \ E \quad C \quad D$$

where b is a basic value in **B**.

Load Global

$$S \quad E \quad (GLOB\ g\ .\ C)\ \ D \ \rightarrow\ ([C':E]\ .\ S)\ \ E \quad C \quad D$$

where g is a global symbol representing a (possibly unevaluated) value. The value can be obtained by executing the code C' for g.

Load Closure

$$S \quad E \quad (CLOS\ C'\ .\ C)\ \ D \ \rightarrow\ ([C':E]\ .\ S)\ \ E \quad C \quad D$$

where C' stands for the code to be evaluated under environment E.

Load Argument

$$S \quad E \quad (ARG\ i\ .\ C)\ \ D \ \rightarrow\ (*E_i\ .\ S)\ \ E \quad C \quad D$$

where $*E_i$ is the i-th element of the current environment E.

Environment Control Instructions

Extend Environment

$$(x\,.\,S)\;\;E\;\;(EXT\_ENV\,.\,C)\;\;D\;\;\rightarrow\;\;S\;\;(x\,.\,E)\;\;C\;\;D$$
$$\phi\;\;E\;\;(EXT\_ENV\,.\,C)\;\;(S'\;E'\;C'\,.\,D)\;\;\rightarrow\;\;([C'':E]\,.S')\;\;E'\;\;C'\;\;D$$

where C'' stands for $(EXT\_ENV\,.\,C)$. The second rule shows how partially evaluated function is obtained.

Make Dummy Environment

$$S\;\;E\;\;(DUMMY\_ENV\,.\,C)\;\;D\;\;\rightarrow\;\;S\;\;(\phi\,.E)\;\;C\;\;D$$

This instruction creates a dummy entry for environment which will be filled by the *FILL_ENV* instruction. This entry becomes the first element of the recursive environment formed by local definitions.

Insert Environment

$$(x\,.\,S)\;\;E\;\;(INS\_ENV\,.\,C)\;\;D\;\;\rightarrow\;\;S\;\;E'\;\;C\;\;D$$

where E' points to the same environment entry as E, which is in fact the dummy entry created by *DUMMY_ENV*. $*E_0=\phi$ remain unchanged as $*E'_0=\phi$, but changes are made by inserting a new entry with value x from the stack as $*E'_1=x$. Accordingly, the entries E_i, $(i\geq 1)$, become E'_{i+1} of the new environment E' (Figure 7.3.2). This instruction along with the *CLOS* instruction effectively creates circular structures for recursive definitions.

Fill Environment

$$(x\,.\,S)\;\;E\;\;(FILL\_ENV\,.\,C)\;\;D\;\;\rightarrow\;\;S\;\;E'\;\;C\;\;D$$

where $E'=E$. The only change is $*E'_0=x$ while $*E_0=\phi$ before execution of *FILL_ENV*. That is, the value x on the stack is moved to the first element which has had a dummy value ϕ.

Evaluation and Application Instructions

Evaluate

$$(x\,.\,S)\;\;E\;\;(EVAL\,.\,C)\;\;D\;\;\rightarrow\;\;(x\,.\,S)\;\;E\;\;C\;\;D$$
$$([\phi:x]\,.\,S)\;\;E\;\;(EVAL\,.\,C)\;\;D\;\;\rightarrow\;\;(x\,.\,S)\;\;E\;\;C\;\;D$$

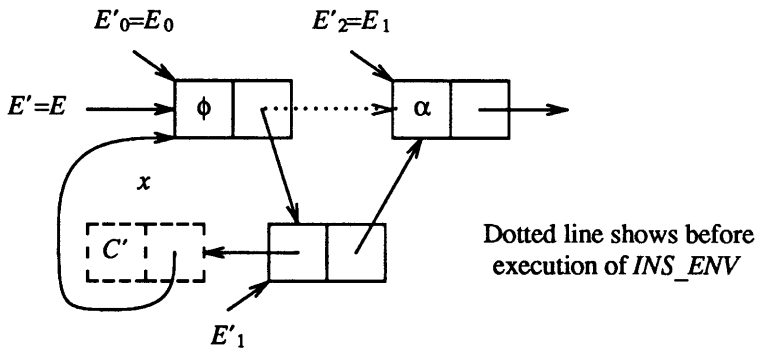


Figure 7.3.2. *INS_ENV* instruction

$$([\phi{:}[C'{:}E]].S)\ E\ (EVAL\ .\ C)\ D \rightarrow ([\phi{:}[C'{:}E]].S)\ E\ C\ D$$
$$([C'{:}E'].S)\ E\ (EVAL\ .\ C)\ D \rightarrow \phi\ E'\ C'\ (([C'{:}E'].S)\ E\ (UPDATE)\ C\ .\ D)$$

where x is not a closure, and the closure $[C':E']$ in the right-hand sides stand for the same one on the left-hand sides. The second and the third cases deal with the *indirection closure* that is produced by the *UPDATE* instruction to avoid repeated evaluation. See the rule for *UPDATE*.

Apply

$$(x\ .\ \phi)\ E\ (APPLY)\ (S'\ E'\ C'\ .\ D) \rightarrow (x\ .\ S')\ E'\ C'\ D$$
$$([\phi{:}[C'{:}E']].S)\ E\ (APPLY)\ D \rightarrow S\ E'\ C'\ D$$

where x may be a closure or an indirect closure. The first rule shows that the value x is returned to the caller of the function when the stack is empty after that value is taken out. The other rule states application of the functional value $[C':E']$ to arguments on the stack. The functional value has already been evaluated and should be kept in an indirection closure.

Call

$$S\ E\ (CALL\ C'\ .\ C)\ D \rightarrow \phi\ E\ C'\ (S\ E\ C\ D)$$

This instruction transfers control to the code C' with saving the current status of the machine.

Update

$$(x\ [C'{:}E'].S)\ E\ (UPDATE)\ (C\ .\ D) \rightarrow (x\ .\ S)\ E\ C\ D$$
$$([C''{:}E''][C'{:}E'].S)\ E\ (UPDATE)\ (C\ .\ D) \rightarrow ([\phi{:}[C''{:}E'']]\ S)\ E\ C\ D$$

where x is not a closure. The top of the stack before execution of this instruction holds the value returned by recursive evaluation invoked by the *EVAL* instruction. The second top of the stack contains the closure to be updated. This closure is changed to an *indirection closure* with its code part ϕ and environment part x or $[C'':E'']$. The indirection closure is used to realize full laziness as shown in Figure 7.3.3. Firstly the closure is evaluated only once because it becomes an indirection closure containing the result. In case that a closure $[C'':E'']$ is obtained as the result of evaluation, the original closure $[C':E']$ is updated to become an indirection closure $[\phi:[C'':E'']]$ which represents an *evaluated value*. It is the indirection closure that should be placed on the stack, not the resulting closure. Another important role of the indirection closure is that it provides a mechanism of sharing as illustrated in Figure 7.3.3. Repeated evaluation of the closure $[C':E']$ can be avoided by indirection.

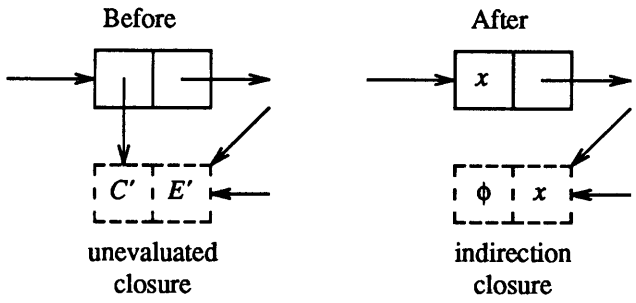


Figure 7.3.3. Indirection closure

The second case of *EVAL* instruction deals with evaluation of the indirection closure.

Primitive Operations

In addition to the instructions having been described, several primitive operations are necessary for implementing arithmetic and Boolean functions. Since such functions differ from language to language, various operations may be needed to implement a particular language on FLFM. It is a reasonable assumption that these functions are classified into several classes, e.g., *strict* functions, *constructor* functions, etc., and functions of a class are implemented in a similar way using corresponding operations. The function provided in a standard environment does not have any free variables in it and therefore it can be referenced

by a *global* symbol. We assume here that some mechanism of linking global symbols g with occurrences of g in the program. The *GLOB* instruction performs part of this process.

We show here how these primitive functions can be implemented by a small set of FLFM instructions. In later sections, we will discuss about some optimization rules to gain efficiency.

Arithmetic and Boolean Operations

We first consider the function *add* for addition of two integers. Assume that we have an instruction *ADD* which adds two elements on the stack S and puts the result on the top of S.

$$(x\ y\ .S)\ \ E\ \ (ADD\ .C)\ \ D\ \ \rightarrow\ \ (\alpha.S)\ \ E\ \ C\ \ D$$

where α represents the sum of two numbers obtained from x and y by evaluation.

The code for *add* can be written as

$$add \equiv (EXT\_ENV\ ;EXT\_ENV\ ;ARG\ 0\ ;EVAL\ ;ARG\ 1\ ;EVAL\ ;ADD\ ;APPLY\ )$$

Semicolons are inserted to mark the boundary of instructions for readability. The last instruction *APPLY* returns the sum on the stack to the caller

Other arithmetic functions such as *sub*, *mul*, etc., and Boolean functions as *eq* are defined quite similarly.

List Operations

Primitive functions for the list structure differ a bit from strict functions. The constructor *cons* for list cells should not evaluate arguments in lazy evaluation [Friedman76].

$$cons \equiv (EXT\_ENV\ ;EXT\_ENV\ ;ARG\ 0\ ;ARG\ 1\ ;CONS\ ;APPLY\ )$$

The instruction *CONS* produces a pair $<x.y>$ in $\mathbf{B}=\mathbf{E}\times\mathbf{E}$ from $x, y \in \mathbf{E}$.

$$(x\ y\ .S)\ \ E\ \ (CONS\ .C)\ \ D\ \ \rightarrow\ \ (<x.y>.S)\ \ E\ \ C\ \ D$$

The selector functions *head* and *tail* first evaluate the argument and take an appropriate component of the pair.

$$head \equiv (EXT\_ENV\ ;ARG\ 0\ ;EVAL\ ;CAR\ ;EVAL\ ;APPLY\ )$$
$$tail \equiv (EXT\_ENV\ ;ARG\ 0\ ;EVAL\ ;CDR\ ;EVAL\ ;APPLY\ )$$

where *CAR* and *CDR* are FLFM instructions:

$$(\langle x.y\rangle . S)\ \ E\ \ (CAR\ .C)\ \ D\ \ \rightarrow\ \ (x\ .S)\ \ E\ \ C\ \ D$$
$$(\langle x.y\rangle . S)\ \ E\ \ (CDR\ .C)\ \ D\ \ \rightarrow\ \ (y\ .S)\ \ E\ \ C\ \ D$$

Conditional Operation

We have not yet dealt with any conditional operations. In ordinary SECD machine, *SEL* and *JOIN* instructions take part in conditional execution. The compiler generating code for such an SECD machine treats *if* as a special form for conditionals. However, in lazy languages, a conditional expression of the form $(if\ e_1\ e_2\ e_3)$ is simply a combination and it can be compiled into

$$(LD\ e_3\ ; LD\ e_2\ ; LD\ e_1\ ; GLOB\ if\ ; EVAL\ ; APPLY\ )\ .$$

where $LD\ e_i$ represents the code for loading the value of expression e_i onto the stack.

The function *if* can be written using a conditional instruction *SELECT* as

$$if \equiv (EXT\_ENV\ ; EXT\_ENV\ ; EXT\_ENV\ ; ARG\ 2\ ; EVAL\ ; SELECT\ C_1\ C_2)$$

where

$$C_1 = (ARG\ 1\ ; EVAL\ ; APPLY\ )$$
$$C_2 = (ARG\ 0\ ; EVAL\ ; APPLY\ )\ .$$

The instruction *SELECT* selects either C_1 or C_2 according to the value at the top of the stack:

$$(\textbf{true}.S)\ \ E\ \ (SELECT\ C_1\ C_2)\ \ D\ \ \rightarrow\ \ S\ \ E\ \ C_1\ \ D$$
$$(\textbf{false}.S)\ \ E\ \ (SELECT\ C_1\ C_2)\ \ D\ \ \rightarrow\ \ S\ \ E\ \ C_2\ \ D$$

The Boolean values **true** and **false** are assumed to be in **B**.

7.4. Compilation rules

Rules for compiling expressions into FLFM code is simpler than those for compiling similar expressions into ordinary SECD machine.

Rules

Let

$$\rho = [u_0, u_1, \cdots, u_p]$$

stand for the *static environment* to lookup variables in lexical-addressing. Concatenation of static environments is represented as

$$[v_0,v_1,\cdots,v_q]\bullet[u_0,u_1,\cdots,u_p] = [v_0,v_1,\cdots,v_q,u_0,u_1,\cdots,u_p]\ .$$

We use the notation in [Henderson80]:

$$e\ *\ \rho$$

represents FLFM code for expression e with respect to the environment ρ, and

$$(s_1)\mid(s_2)\mid\cdots\mid(s_n)$$

stands for

$$(s_1\ s_2\ \cdots\ s_n)$$

and $(EXT\_ENV)^k$ to represent a sequence of EXT_ENV instruction repeated k times. The basic compilation rules follow.

(*1) Basic value notation $b \in$ **Bas**

$$b*\rho = (CONST\ \beta)\mid(APPLY)$$

where β is the value in **B** the expression b represents.

(*2) Identifier $x \in$ **Ide**

$$x*[u_0,u_1,\cdots,u_p] = \begin{cases} (ARG\ i)\mid(EVAL)\mid(APPLY) & \text{if } x=u_i \text{ for some } i \\ (GLOB\ x)\mid(EVAL)\mid(APPLY) & \text{otherwise} \end{cases}$$

(*3) Combination

$$(e_0\ e_1\ \cdots\ e_n)*\rho = e_n\dagger\rho\mid\cdots\mid e_1\dagger\rho\mid e_0*\rho$$

(*4) Function notation

$$(\mathbf{fn}\ x_1\cdots x_k.e)*\rho = (EXT\_ENV)^k\mid e*\rho$$

(*5) Expression with local dedinitions[1]

$$(e_0\ \mathbf{where}\ x_1=e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_m=e_m)*\rho = e_m\dagger\rho\mid\cdots\mid e_1\dagger\rho\mid(EXT\_ENV)^m\mid e_0*\rho'$$

where $\rho'=[x_m,\cdots,x_1]\bullet\rho$.

$$(e_0\ \mathbf{whererec}\ x_1=e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_m=e_m)*\rho =$$
$$(DUMMY\_ENV)\mid e_m\dagger\rho'\mid\cdots\mid e_1\dagger\rho'\mid(INS\_ENV)^{m-1}\mid(FILL\_ENV)\mid e_0*\rho'$$

where $\rho'=[x_m,\cdots,x_1]\bullet\rho$.

---

[1] Although expressions with **where**-clauses are not of the fully lazy normal form, such expression can be compiled into FLFM code. Since the rules may be applied to expressions not satisfying the condition of the fully lazy normal form, the rule is included here.

Expressions of which evaluation is suspended are transformed as follows:

(†1) Basic value notation $b \in \mathbf{Bas}$

$$b\,\dagger\rho = (\,CONST\ \beta\,)$$

where β is the value in **B** the expression b represents.

(†2) Identifier $x \in \mathbf{Ide}$

$$x\,\dagger[u_0,u_1,\cdots,u_p] = \begin{cases} (\,ARG\ i\,) & \text{if } x=u_i \text{ for some } i \\ (\,GLOB\ x\,) & \text{otherwise} \end{cases}$$

(†3-5)Combination, Function notation, and Expression with local definitions

$$e\,\dagger\rho = (\,CLOS\,) \mid e^*\rho$$

Finally, the code for evaluating the outermost expression e under the standard initial environment ρ_0 is

$$(\,CALL\,) \mid e^*\rho_0\ .$$

Optimization

The compilation rules described above do not use any specific information about primitive functions. If we had used such information, we could obtain better FLFM code.

Suppose that we have a term $(add\ e_1\ e_2)$. If we use the knowledge about the arity of *add* and its *strictness*, the environment consisting of e_1 and e_2 is not necessary. In such a case, we can generate FLFM code as

$$(add\ e_1\ e_2)^*\rho = (\,CALL\,) \mid e_2^*\rho \mid (\,CALL\,) \mid e_1^*\rho \mid (\,ADD\,) \mid (\,APPLY\,)\ .$$

The expression is otherwise compiled into

$$(\,CLOS\,) \mid e_2^*\rho \mid (\,CLOS\,) \mid e_1^*\rho \mid (\,GLOB\ add\,) \mid (\,EVAL\,) \mid (\,APPLY\,)$$

and the global function *add* creates an environment as specified in the previous section. Similar rules can be applied to other arithmetic and Boolean functions. It should be noted that such optimization cannot be taken unless sufficient number of arguments are accompanied with the primitive function.

For the list constructor *cons*, and for the selectors *head* and *tail*, we have

$$(cons\ e_1\ e_2)^*\rho = e_2\dagger\rho \mid e_1\dagger\rho \mid (CONS) \mid (\,APPLY\,)$$
$$(head\ e_1)^*\rho = (\,CALL\,) \mid e_1^*\rho \mid (CAR) \mid (\,EVAL\,) \mid (\,APPLY\,)$$
$$(tail\ e_1)^*\rho = (\,CALL\,) \mid e_1^*\rho \mid (CDR) \mid (\,EVAL\,) \mid (\,APPLY\,)$$

Given the conditional form ($if\ e_1\ e_2\ e_3$), we can optimize the term as

$$(if\ e_1\ e_2\ e_3)^*\rho = (\ CALL\ ) \mid e_1^*\rho \mid (\ SELECT\ ) \mid e_2^*\rho \mid e_3^*\rho\ .$$

7.5. Implementation

In this section, we look over an FLFM implementation on a conventional machine MC68000 in order to fill the gap between the virtual machine model FLFM and the actual machine. As we shall see in Chapter 8, we have already implemented FLFM on four kinds of basically different machines.

As mentioned in Section 7.3, there is no need to use the list structure to represent every object held by the registers S, E, C, and D. In the first place, the stack S can be implemented by usual hardware stack manipulated by auto-increment and -decrement addressing of MC68000. Moreover, the value at the top of S is always held in a data register instead of on the stack. The code C is a fixed code of MC68000 instructions, and controlled by the program counter. The dump D can be embedded in the stack using the frame pointer indicating stack frames for recursive activations of functions. To attain the sharing property of the environment E, it is reasonable to make the environment using the list structure as illustrated in figures in Sections 7.2 and 7.3.

Each value is represented by a 32 bit word of which first 8 bit byte is used for the *tag* part indicating value types. Remaining 24 bit field contains primitive (integer, character, or Boolean) value, or a pointer to a cell allocated in the *heap* store. Four of the address registers of MC68000 are devoted to maintaining S, E, D, and the heap store. Figure 7.5.1 illustrates the stack, the heap, and pointers. Tags are not shown in the figure. We use a standard technique for garbage collection. Two heap storages are provided. When we are facing with the situation where no more cells are available, active cells in the current heap are copied to the other heap and then the role of the heaps is exchanged. An instruction *CHECK* inspects the heap pointer for availability of new cells in the current heap.

Most of the FLFM instructions are expanded into MC68000 instructions. Commonly used instruction sequences like *EXT_ENV*, *APPLY*, etc., are supplied as run-time routines. An example of MC68000 code is shown in the Appendix A of this thesis. As for the performance, the code thus obtained runs as fast as about 4400 function calls per second on Sun-2 workstations. This exceeds the execution count, about

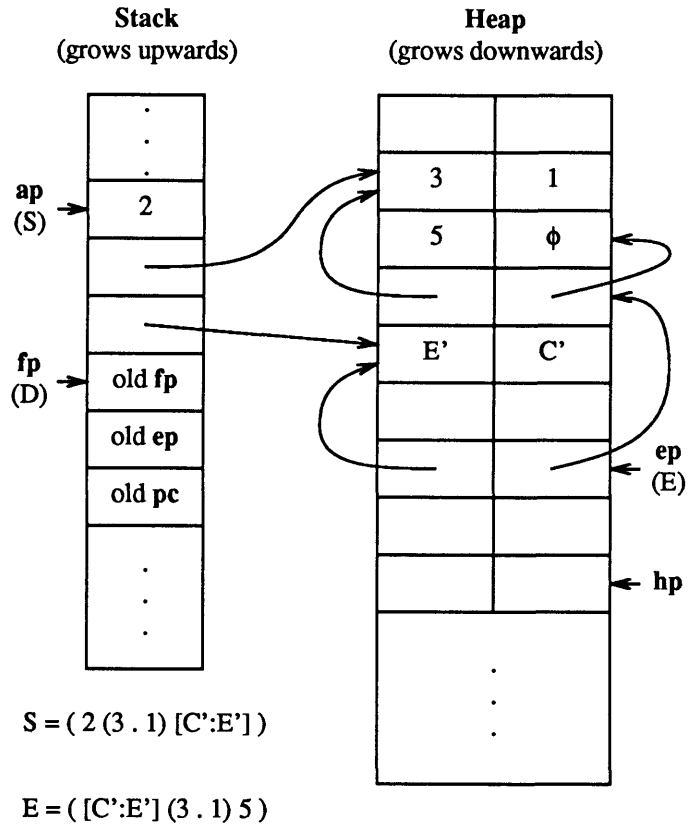


Figure 7.5.1. Implementation of FLFM

2000, by a micro-coded interpreter of the SECD machine for Lispkit Lisp on the Perq [Henderson83].

We shall leave further implementation details to Chapter 8.

7.6. Remarks

Our primary concern in this chapter is to develop compilation technique for fully lazy evaluation on conventional machines. Once we have obtained expressions of the fully lazy normal form, we can evaluate them in fully lazy way using ordinary lazy evaluation mechanism. It requires, however, that the evaluator allows function application of insufficient number of arguments, which is not the case of ordinary SECD machines. Our FLFM has been designed as a basic model of fully lazy evaluators. As shown above, code generation from FLFM code turns out to be much easier than from ordinary SECD code. This enables us to enhance portability of compiler systems.

Johnsson [Johnsson85] deals with a compilation scheme for ordinary lazy evaluation. It is, however, different from ours in several respects. Among others, it does not support full laziness, as is the basis of our method.

An advantageous feature of full laziness in practical problems has been investigated in Chapter 9. We believe that our compilation technique is extremely useful for exploiting the applicability of full laziness.

Chapter 8

Fully lazy implementation

There are many varieties of functional languages that have been developed to promote programming practices in a functional style. As for implementations of these languages, there has been a reputation for inefficiency. One of the reasons is that they were typically run interpretively rather than compiled. In this chapter, we shall provide an existence proof for both the feasibility and the viability of functional languages implemented on conventional computers and also explore the usefulness of the fully lazy normal form in compilers. We choose a compilation method based on the technique of *lambda-hoisting* (Chapter 6), and translate functional languages into a target language. Our target language is the code of the *Fully Lazy Functional Machine* (Chapter 7). The FLFM code is then transformed into machine code of several computers by a table-driven translator. This greatly enhances the portability of the compiler system. We shall demonstrate implementations on four different machines in Section 8.2.

Despite different appearances of functional languages, almost all share similar characteristics. More precisely the differences are superficial but the similarities are fundamental. Because the underlying concepts of functional languages are quite few, we provide the language implementer with a *common intermediate language* as the interface to fundamental features of functional programming. The common intermediate language is a simple functional language that can be compiled into machine code. Although we do not hope to add ill-conceived languages to the Tower of Babel of programming languages [Sammet69], the use of the common intermediate language makes a new implementation quite easy. In Section 8.3, we shall present an example implementation of a language designed for evaluating the expressive power of a set notation similar to that of Zermelo-Fraenkel set theory.

8.1. Compiler design

Examples of functional languages are pure Lisp [McCarthy62], Backus' FP [Backus78], ISWIM [Landin66], ML [Gordon79b, Milner84], HOPE [Burstall80], SASL [Turner76], KRC [Turner82], and Miranda [Turner85]. Although these languages vary in appearance from the simplest form of Lisp to a sophisticated notation of Miranda, they are built on a few fundamental concepts:

(1) A set of objects, e.g., primitive data values and primitive functions,

(2) Functional abstraction, e.g., **fn** $x.e$,

(3) Functional application, e.g., $(e_0\ e_1)$,

(4) Definition (possibly recursive) mechanism, e.g., $x{=}e$,

(5) Data structuring facilities such as constructing lists and pairs

(6) Lazy and non-lazy evaluation mechanisms.

Some languages do not allow functional abstraction and higher order functions; in which case predefined *functional forms* are used for combining terms [Backus78]. The other provides extended features such as exception handling and assignment [Gordon79b, Milner84]. The addition of a type discipline is common to modern functional languages [Milner84, Turner85]. See Chapter 10 of this thesis. Most of these differences can be treated separately before generating machine code, and programs can be embedded in a standard representation of fundamental concepts. It is believed that the large part of languages is common to each other. One reason of convincing on this point is the fact that functional programming allows only a few alternatives for introduction of new notions. Any representation for new idea would be translated into a crude but more primitive form by combining already existent ones using functional composition. Such extensibility is one of the advantageous features in functional programming. The situation is contrast to extensible procedural languages in the late sixties, e.g., Algol 68 [Wijngaarden69], which went into the great complexity. As far as functional languages are concerned, it seems better to design a translator for a *common intermediate language* that provides typical features (1)-(6) of many functional languages. The *language-dependent* part of the compiler is separated in this way as the *front-end.* The front-end of the compiler translates the source program of a particular language into the common intermediate language. The common intermediate language acts as an interface between the language-dependent part and the rest of the compiler.

Let us next consider the other end of the compiler. It is obviously desirable that the compiler is portable and can be implemented on many machines. One method is to define a *target language* which links the *machine-independent* and *machine-dependent* parts. The target language could also be thought of as an

abstract machine since the machine-independent part of the compiler see it like an object machine. The machine-independent part known as the *back-end* translates the target language into machine code of the object machine. There are a number of examples of such abstract machines for implementing procedural languages. BCPL [Richards71] has been implemented on many machines using OCODE as a target language. We had designed a back-end of OCODE and implemented BCPL on the Melcom 70/250 computer which is also an object machine of our functional languages. Some implementations of Pascal use a target language called P-code. Compilers generate P-code which is then usually interpreted. Further details of porting Pascal compilers are discussed in [Takeichi77]. A few abstract machines have been used for functional languages. The SECD machine [Henderson76] is an example, but the code is designed to be interpreted as P-code for Pascal is. Cardelli has designed a target language FAM (functional abstract machine) for ML [Cardelli84b] which could be implemented on many machines. We had ported ML on the Melcom 70/250 [Chujo84]. FAM reflects the design of ML in a number of ways. It does not provide any instruction for lazy evaluation. What is worse is that the back-end tends to become complex and porting the compiler to an essentially different machine is a laborious task. Making good use of our experiences on portable compilers, we have designed an abstract machine, *Fully Lazy Functional Machine* (FLFM), as described in Chapter 7. The target language FLFM is compact and provides instructions convenient for implementing (full) laziness. It will turn out that we can generate efficient code for various machines from the FLFM code.

Since we have defined a common intermediate language and a target language, the problem of implementing m languages on n machines would essentially involve writing $m+n$ pieces of translators, i.e., m language-dependent front-ends and n machine-dependent back-ends, rather than $m \times n$ complete compilers (Figure 8.1.1). The language- and machine-independent part of the compiler is called the *core translator.*

8.2. Compiler structure

We have seen in the previous section that our compiler comprises the core translator, a front-end translator for each source language and a back-end translator for each object machine. In this section we discuss in more detail the structure of the compiler. The main part of the compiler is the core translator that translates the common intermediate language into the target language from which the back-end translator

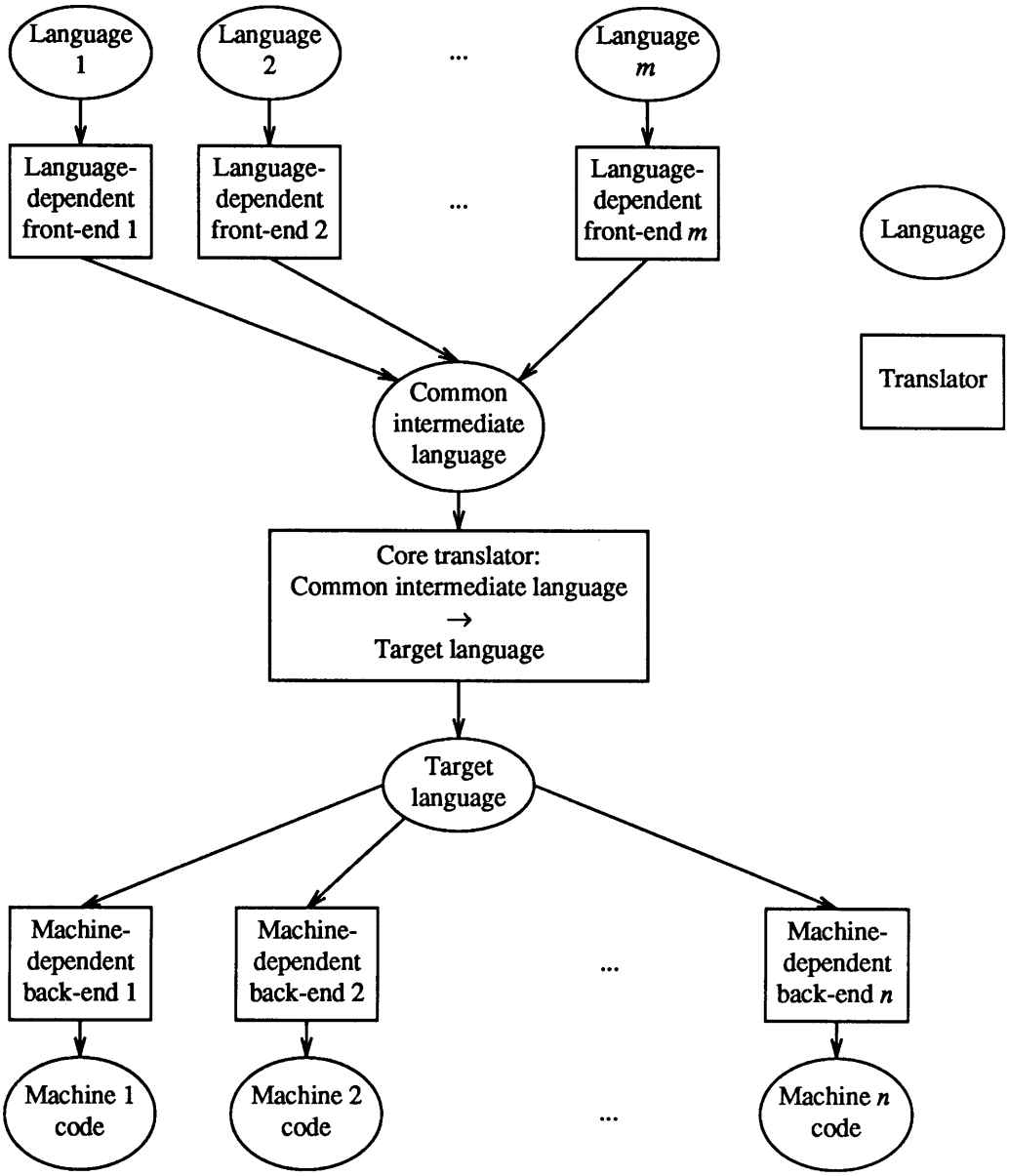
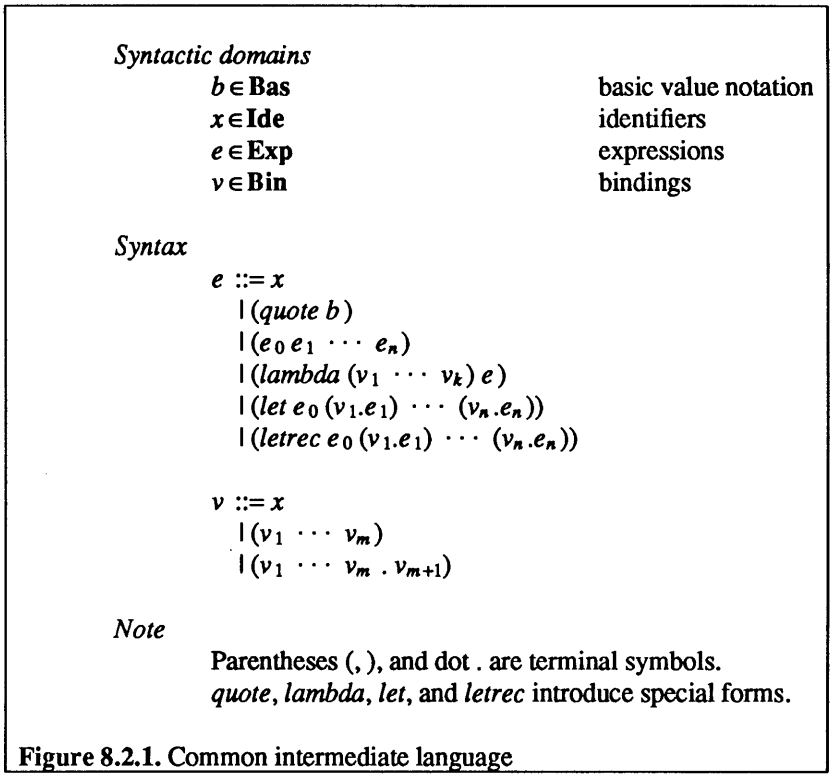


Figure 8.1.1. Compilers of m languages on n machines

generates machine instructions. A front-end translator will be demonstrated in the next section.

Common intermediate language

Our common intermediate language borrows its syntax from Lispkit Lisp [Henderson76] with a few extensions (Figure 8.2.1). Every program of the common intermediate language is an S-expression. New forms of bindings $v \in \mathbf{Bin}$ are included in the language; that is, *compound bindings* of the form



Syntactic domains

$b \in$ **Bas**	basic value notation
$x \in$ **Ide**	identifiers
$e \in$ **Exp**	expressions
$v \in$ **Bin**	bindings

Syntax

$e ::= x$
$\mid (quote\ b)$
$\mid (e_0\ e_1 \cdots e_n)$
$\mid (lambda\ (v_1 \cdots v_k)\ e)$
$\mid (let\ e_0\ (v_1.e_1) \cdots (v_n.e_n))$
$\mid (letrec\ e_0\ (v_1.e_1) \cdots (v_n.e_n))$

$v ::= x$
$\mid (v_1 \cdots v_m)$
$\mid (v_1 \cdots v_m\ .\ v_{m+1})$

Note

Parentheses (,), and dot . are terminal symbols.
quote, *lambda*, *let*, and *letrec* introduce special forms.

Figure 8.2.1. Common intermediate language

$(v_1 \cdots v_m)$ or $(v_1 \cdots v_m\ .\ v_{m+1})$

are allowed as well as simple variables $x \in$ **Ide**. The requirements (1)-(6) mentioned in the previous section should be examined.

(1) A primitive value is an integer, a Boolean, a character, or a distinguished value **nil**.

An integer value is represented by a quoted expression as

$(quote\ b)$

where $b \in$ **Bas** is a decimal notation of the number.

Boolean values are written using a constructor *bool* as

(*bool* (*quote* 0)) for **false**

and

(*bool* (*quote* 1)) for **true** .

These values are of a primitive data type *bool* unlike Lisp where **nil** and non-**nil** values act as Boolean.

Similarly a character is represented as

(*char* (*quote c*))

where c is the internal code of the character.

For simplicity, (*quote b*) is sometimes written using a quote symbol "' as "*b*'.

The distingushed value **nil** is identified by a special identifier *nil* or an empty combination ( ) according to the Lisp tradition.

The constructors

bool: **int** → **bool**

and

char: **int** → **char**

are predefined type transfer functions.

Primitive functions may vary depending upon the source language to be implemented. The compiler knows only a few functions such as *add*, *sub*, *eq*, *if*, etc., and makes machine-independent optimization on these functions as described in Section 7.4. Other language-dependent functions are assumed to be provided as library functions. All the optimization rules form a table describing how the compiler deals with special combinations of functions and arguments. The contents of the table may be changed to meet the rules of the source language.

(2) Functional abstraction is represented by an expression

(*lambda* (v_1 v_2 $\cdots$ v_k) e)

which is semantically equivalent to

(*lambda* (v_1) (*lambda* (v_2) $\cdots$ (*lambda* (v_k) e) $\cdots$)).

We may write this expression in our referential language being used throughout the thesis as

fn $v_1 \cdots v_k.e$.

The binding of the lambda form is extended in such a way that components of a structured argument may be specified by variables. See below for the details.

(3) Functional application is of the form

$(e_0\, e_1 \cdots e_n)$

which is equivalent to

$(e_0\, (e_1 \cdots e_n))$.

(4) An expression with local definition is either

$(let\ e_0\ (v_1.e_1) \cdots (v_n.e_n))$

or

$(letrec\ e_0\ (v_1.e_1) \cdots (v_n.e_n))$.

Corresponding expressions in our referential language are

e_0 **where** $v_1{=}e_1$ **and** $\cdots$ $v_n{=}e_n$

and

e_0 **whererec** $v_1{=}e_1$ **and** $\cdots$ $v_n{=}e_n$

respectively. However, the bindings v_i may be compound.

(5) There is a primitive data constructor *cons* for creating a pair structure. As in Lisp, there is no distinction between pairs and lists. These are distingushed only in the source language, if necessary. If we wish to handle a record structure as in Pascal

record *nodevalue*: *integer*; *left*, *right*: *pointer* **end**

the front-end should translate it into a structure using pairs

(*nodevalue* . (*left* . *right*)) or equivalently (*nodevalue left* . *right*)

for example, and produce selector functions

nodevalue ≡ (*lambda* ((*n l* . *r*)) *n*)

left ≡ (*lambda* ((*n l* . *r*)) *l*)

and

right ≡ (*lambda* ((*n l* . *r*)) *r*).

In order to represent an empty structure, **nil** is provided as a primitive value denoted by *nil*.

(6) The compiler compiles any program written in the common intermediate language to generate code that evaluate the original expression in a lazy (actually in a fully lazy) way. A special function *val* causes the argument expression be evaluated before function call. Consider for example

(*f* (*val* (*add n* 1)) (*sub n* 1)) .

The code sequence of this expression looks like

<evaluate (*add n* 1)>; <make closure (*sub n* 1)>; <call *f* >

or

(*ARG n*; *CONST* 1; *ADD*; *CLOS* (*ARG n*; *CONST* 1; *SUB*; *APPLY*); *CALL f*)

in FLFM code. That is, the function *val* may be used to indicate that the argument should be evaluated in a non-lazy way.

Compound bindings

Burstall [Burstall69] proposed several syntactic extensions of the ISWIM language [Landin66] which is the basis of our referential language. These enable us to write compact programs that manipulate data structures such as lists and tuples. The basic idea is that a single identifier, say *cons*, may serve for the data constructor, the destructor producing the components of the data structure, and the predicate. The role of the identifier can be distingushed from the context it appears. For example,

cons (*a* ,*b*)

in ISWIM means the pair structure with *a* and *b* as its components. If *cons* appears in a binding as

let *cons* (*x* ,*y*)=*p* **in** *e* ,

x and *y* are bound to the components of the pair *p* ; that is, it is equivalent to

let *x*=*car* (*p*) **and** *y*=*cdr* (*p*) **in** *e* .

The use of the constructor this way is very attractive for representing decomposition of the data structure, because it makes the selectors virtually useless and explicit use of the selector disappears.

The extension of bindings in the common intermediate language follows Burstall's idea. For example,

(*let e* ((*x*.*y*) . *p*))

means

(*let e* (*x* . (*head p*)) (*y* . (*tail p*)))

where *head* and *tail* are standard selector functions of the data structure produced by the constructor *cons*. Such a syntactic device allows us to write compact programs that manipulate data structures. Recent

functional languages provide mechanisms to define a function by giving several alternative equations distinguished by the use of different patterns in the parameters. In Miranda [Turner85],

reverse [ ] = [ ]
reverse (*a*:*x*) = *reverse x* ++ [*a*]

define a function that reverses a list. Such a function definition may be expressed in the common intermediate language as[1]

```
(reverse .
  (lambda (u)
    (if (null u) (quote nil)
      (let (append (reverse x) (cons a (quote nil))) ((a. x) . u)) ) ) )
```

where *u* is a fresh identifier. We shall discuss such translation rules of the front-end in the next section.

Elimination of compound bindings

The first phase of the core translator eliminates compound bindings of the form

$(v_1 \cdots v_m)$ or $(v_1 \cdots v_m \,.\, v_{m+1})$

by decomposing the structure using the standard selector functions *head* and *tail*. The translation rules are summarized in Figure 8.2.2.

For example,

(*lambda* ((*x*.*y*)) *e*)

is transformed as

(*lambda* ((*x*.*y*)) *e*) → (*lambda* (*u*) (*let V*[*e*] *S*[(*x*.*y*),*u*]))

where *u* is a fresh identifier and

S[(*x*.*y*),*u*] = *S*[*x*,(*head u*)] ! *S*[*y*,(*tail u*)] = ⌈(*x*. (*head u*)) (*y*. (*tail u*))⌉ .

Finally we have an expression without compound bindings

(*lambda* (*u*) (*let V*[*e*] (*x*. (*head u*)) (*y*. (*tail u*)))) .

It should be noted that local definitions are added to the original expression but nesting of the lambda expression remains unchanged. We could have used nested lambda forms instead of let- or letrec-expressions as

[1] This is not an expression, but a definition that may appear in a *letrec*-expression.

Syntactic domains

$b \in$ **Bas**	basic value notation
$x \in$ **Ide**	identifiers
$e \in$ **Exp**	S-expressions
$v \in$ **Bin**	bindings

Rules for elimination of compound bindings

V : **Exp** $\rightarrow$ **Exp**

$V[x] = \lceil x \rceil$

$V[(quote\ b)] = \lceil (quote\ b) \rceil$

$V[(e_0\ e_1\ \cdots\ e_n)] = \lceil (V[e_0]\ V[e_1]\ \cdots\ V[e_n]) \rceil$

$V[(lambda\ (v_1\ \cdots\ v_k)\ e)] =$
$\lceil (lambda\ (x_1\ \cdots\ x_k)\ (let\ V[e]\ S[v_1, x_1]\ |\ \cdots\ |\ S[v_k, x_k])) \rceil$
where x_i are fresh identifiers.

$V[(let\ e_0\ (v_1\ .\ e_1)\ \cdots\ (v_n\ .\ e_n))] =$
$\lceil (let\ (let\ V[e_0]\ S[v_1, x_1]\ |\ \cdots\ |\ S[v_n, x_n])\ (x_1\ .\ e_1)\ \cdots\ (x_n\ .\ e_n)) \rceil$
where x_i are fresh identifiers.

$V[(letrec\ e_0\ (v_1\ .\ e_1)\ \cdots\ (v_n\ .\ e_n))] =$
$\lceil (letrec\ V[e_0]\ S[v_1, x_1]\ |\ \cdots\ |\ S[v_n, x_n]\ (x_1\ .\ e_1)\ \cdots\ (x_n\ .\ e_n) \rceil$
where x_i are fresh identifiers.

Rules for decomposition of compound bindings

S : **Bin**$\times$**Exp** $\rightarrow$ **Exp***

$S[x,x] = \phi$ (empty)

$S[x,e] = \lceil (x\ .\ e) \rceil$

$S[(v_1\ v_2\ \cdots\ v_m),e] =$
$S[v_1,(head\ e)]\ |\ S[v_2,(head\ (tail\ e))]\ |\ \cdots\ |\ S[v_m,(head\ (tail^{m-1}\ e))]$

$S[(v_1\ v_2\ \cdots\ v_m\ .\ v_{m+1}),e] =$
$S[v_1,(head\ e)]\ |\ S[v_2,(head\ (tail\ e))]\ |\ \cdots$
$|\ S[v_m,(head\ (tail^{m-1}\ e))]\ |\ S[v_{m+1},(tail^m\ e)]$

Notations

Syntactic elements are quoted by $\lceil$ and $\rceil$.
Operator | concatenates S-expressions.
$tail^p$ stands for p-fold application of the function *tail*.

Figure 8.2.2. Elimination of compound bindings

$(lambda\ ((x.y))\ e) \rightarrow (lambda\ (u)\ (\ (lambda\ (x\ y)\ V[e\ ])(head\ u)(tail\ u))\ )$

It is, however, undesirable for our purpose. Extra lambda-bindings induce more operations required for passing parameters. It is from this reason that we have introduced the fully lazy normal form instead of using super-combinators as a basis of a compiler producing efficient code. See the discussion in Chapter 5.

Lambda-hoisting

The second phase of the core translator performs the lambda-hoisting procedure of Chapter 6 on the expression that has passed through the first phase.

FLFM code generation

The core translator finally generates FLFM code according to the rules described in Section 7.4. The compiler consults a table that specifies machine-independent optimization rules for generating efficient code. Although optimization rules for standard functions have been included in the table, language-dependent rules or heuristic rules might be added, if desired.

Machine code generation

The back-end translator of the compiler takes FLFM code as input and generates machine code as output. The task of the back-end is very simple; it repeatedly reads an FLFM instruction and looks up the corresponding machine instructions in the *code table*. It can be said that the back-end translator generates machine code in a table-driven way. Consequently we have only to make a code table for each object machine.

We have developed code tables and FLFMs for four different machines:

- Motorola MC68000, one of the most popular microprocessors,
- Melcom 70/250, an enhanced version of old Sigma 7,
- Melcom MX2000, a machine very similar to IBM 370, and
- Intel i8086, another popular microprocessor for small systems.

The computers on which the compilers and the FLFMs have actually been implemented are the Sun-2 workstation, the Melcom 70/250, the Melcom MX2000, and the NEC-PC9801.

It can be said that the core translator and several back-end translators form a portable functional system as shown in Figure 8.2.3.

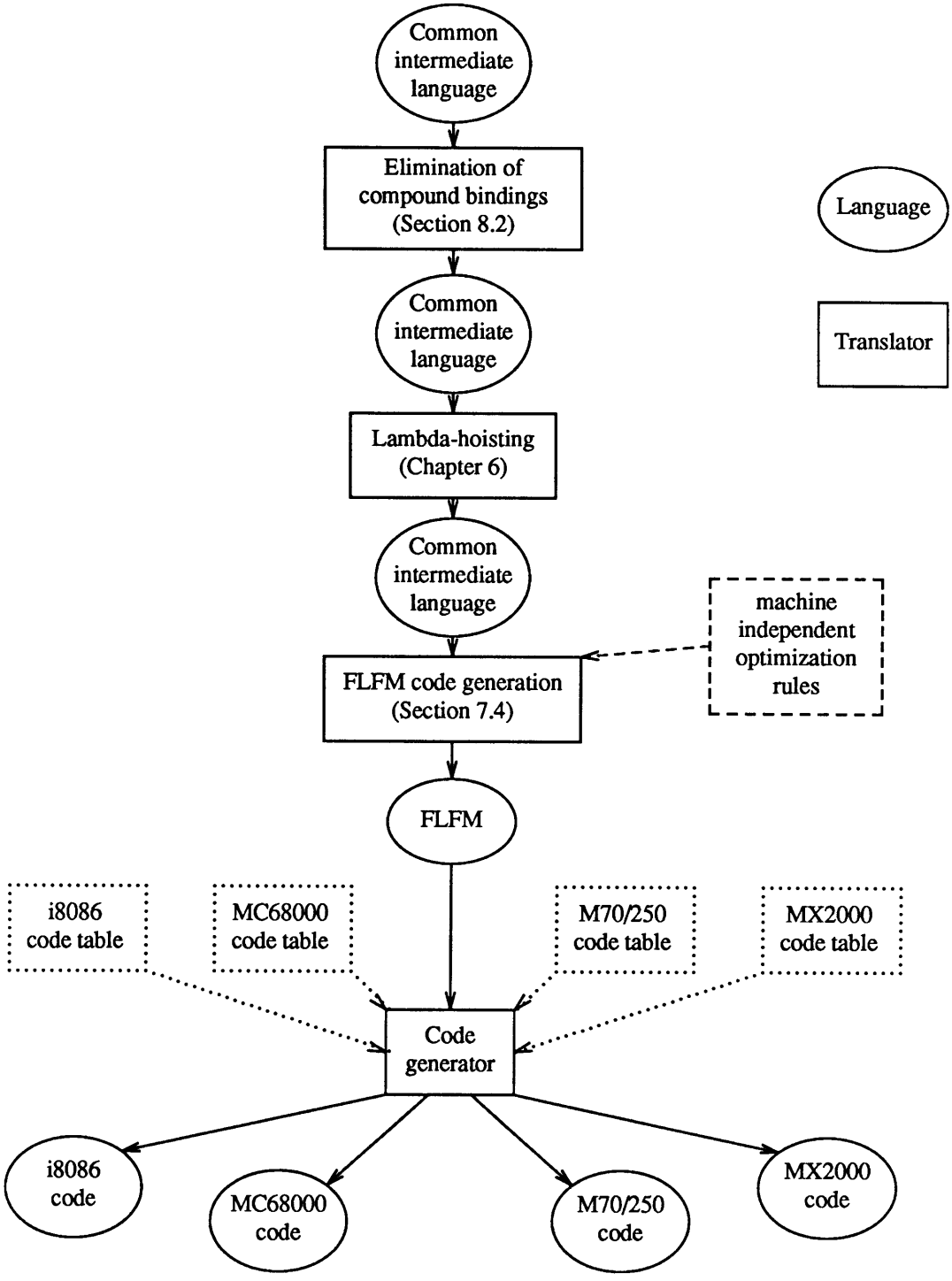


Figure 8.2.3. Core translator and back-ends

8.3. A functional language translator

We may write programs in the common intermediate language, and this is the only way of programming until more sophisticated languages are made. As described in Section 8.1, our compiler for a new language can be implemented by prefixing its front-end translator to the core translator. This section provides an informal introduction to a functional language *uc* (pronounced *you see*) and explains how the front-end translator is constructed[2]. It is not claimed that the syntax or semantics of the language *uc* are more elegant than those of other functional languages. The *uc* language rather borrows many ideas from Miranda [Turner85].

Translation rules are presented on the basis of abstract syntax of *uc*. The concrete syntax of *uc* is found in Appendix B of this thesis. For any form α of *uc*,

$\alpha \rightarrow$ S-expression

means that α should be translated into the right-hand expression of the common intermediate language. We sometimes write

$\lceil \alpha \rceil$

to represent the corresponding S-expression. For example,

if e_1 **then** e_2 **else** $e_3 \rightarrow$ (*if* $\lceil e_1 \rceil$ $\lceil e_2 \rceil$ $\lceil e_3 \rceil$)

is the rule for conditional expressions.

The language is purely functional. A *uc* program is an expression which represents a value we are interested in. Every program has a unique name that may be referred by other programs. That is, the name of a program is *global* in the sense that it denotes the expression common to all the programs in that computing environment. From the viewpoint of implementation these names are used for combining separately compiled modules to form a larger program. Such a global name g appeared in some expression should be compiled to an FLFM instruction *GLOB* g. Other names are local in a program and must be bound by **fn**-abstraction or local declaration. Here is a simple program fac:

---

[2] The language *uc* was used in previous chapters for explaining ideas in functional programming.

f **whererec**
$f\ n\ =$ **if** $n==0$ **then** 1 **else** $n*f\,(n-1)$.

Then we can write a program fac 10 that uses the expression denoted by fac as

$fac\ 10$.

This program is equivalent to

$f\ 10$
whererec
$f\ n\ =$ **if** $n==0$ **then** 1 **else** $n*f\,(n-1)$.

Note that the identifier f in the program fac has a meaning inside that program, but the name fac is global. The name fac actually denotes the value of f which is defined by means of a **whererec**-clause.

Any identifier x in a *uc* program is translated into itself, i.e.,

$x \rightarrow x$.

Primitive values are those provided by the common intermediate language. An integer constant is denoted by the usual decimal notation δ.

$\delta \rightarrow$ (*quote* δ) .

Boolean constants are represented by **true** and **false** both of which are reserved word.

true $\rightarrow$ (*bool* (*quote* 1))
false $\rightarrow$ (*bool* (*quote* 0)) .

A character constant is written as 'c'.

'c' $\rightarrow$ (*char* (*quote* δ))

where δ is a decimal representation of internal code for the character c. A string constant has the form "$c_1c_2\cdots c_n$" that represents a list structure of characters (See below).

"$c_1c_2\cdots c_n$" $\rightarrow$ $\lceil$ ['c_1' , 'c_2' , $\cdots$, 'c_n'] $\rfloor$.

The most commonly used data structure is the list which is written using brackets and commas as [1 , 2 , 3 , 5 , 7 , 11 , 13] .

[e_1 , e_2 , $\cdots$, e_n] $\rightarrow$ (*cons* $\lceil e_1 \rceil$ (*cons* $\lceil e_2 \rceil$ $\cdots$ (*cons* $\lceil e_n \rceil$ (*quote nil*)) $\cdots$))

where *cons* is a primitive data constructor provided in the common intermediate language. An empty list is denoted by a reserved word **nil** or [ ] both of which are translated into (*quote nil*).

There is a shorthand notation using '..' for lists whose elements form an arithmetic series of unit differences. For example, [1..n] means a list [1,2, $\cdots$,n]. Such an expression is translated into a combination using a library function *fromto*.

$$[e_1 .. e_2] \rightarrow (fromto\ \lceil e_1 \rceil\ \lceil e_2 \rceil).$$

The function *fromto* is defined as

$$fromto\ x\ y\ =\ \textbf{if}\ x>y\ \textbf{then}\ [\,]\ \textbf{else}\ x : fromto\ (x+1)\ y$$

where ':' is an infix operator for *cons* (See below).

Lazy evaluation makes it possible to write down infinite data structures. We may write a program, i.e., a global expression, *ones* that represents an infinite sequences of ones as

$$ones\ \ \textbf{whererec}\ ones = 1 : ones\ .$$

There is a modified form of the '..' notation for the infinite arithmetic series.

$$[e_1 ..\,] \rightarrow (from\ \lceil e_1 \rceil)$$

where the library function *from* is

$$from\ x\ =\ x : from\ (x+1).$$

The notation for function application is simply juxtaposition as is usual in many functional languages.

$$e_0\, e_1 \cdots e_n \rightarrow (\lceil e_0 \rceil\ \lceil e_1 \rceil\ \cdots\ \lceil e_n \rceil).$$

There are infix operators and unary oprerators for commonly used functions.

$$e_1\ ,\ e_2 \rightarrow (cons\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$$e_1\ +\!+\ e_2 \rightarrow (append\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$$e_1\ :\ e_2 \rightarrow (cons\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$$e_1\ \|\ e_2 \rightarrow (or\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$$e_1\ \&\&\ e_2 \rightarrow (and\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$$e_1 == e_2 \rightarrow (eq\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$$e_1\ !=\ e_2 \rightarrow (neq\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$$e_1 < e_2 \rightarrow (lt\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$$

$e_1 > e_2 \rightarrow (gt\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$e_1 <= e_2 \rightarrow (leq\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$e_1 >= e_2 \rightarrow (geq\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$e_1 + e_2 \rightarrow (add\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$e_1 - e_2 \rightarrow (sub\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$e_1 * e_2 \rightarrow (mul\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$e_1 / e_2 \rightarrow (div\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$e_1 \% e_2 \rightarrow (rem\ \lceil e_1 \rceil\ \lceil e_2 \rceil)$

$\tilde{}\ e_1 \rightarrow (neg\ \lceil e_1 \rceil)$

$!\ e_1 \rightarrow (not\ \lceil e_1 \rceil)$

It would be unnecessary to explain in detail, but some remarks should be made. The functions that appear in the right-hand sides are provided in the common intermediate language and most of them have thier counterparts of FLFM instructions. The function *cons* appears in two rules for operators ',' and ':'. Since the data structure available in the common intermediate language and FLFM is only one produced by *cons*, every structure in the source language has to be mapped into the single domain. It should also be noted that the unary minus operator is denoted by '~'. The reason of doing so will be clear in the next paragraph.

The language *uc* is a fully higher order language and functions are first class objects. It is therefore desirable to write standard functions listed above in some way using operator symbols. The language solves this problem as

$(\omega) \rightarrow \omega_f$

where ω is an operator and ω_f is the name of its function counterpart. For example,

$(+) \rightarrow add$
$(-) \rightarrow sub$
$(\tilde{}) \rightarrow neg$
$(\&) \rightarrow and$.

Many list processing functions can be obtained by partially parametrizing a library function *foldr*:

```
foldr whererec
    foldr op k = fn x. if x==[ ] then k else op a (foldr op k u)
                                    where (a:u)=x
```

as in

foldr (+) 0 for the sum of the elements,
foldr (*) 1 for the product of the elements.

Functional abstraction is written as **fn** $v_1 \cdots v_k$. e where v_i are *variable structures* that are simply translated into simple variables or *compound bindings* of the common intermediate language.

fn $v_1 \cdots v_k$. $e \rightarrow$ (*lambda* ($\lceil v_1 \rceil \cdots \lceil v_k \rceil$) $\lceil e \rceil$) .

A variable structure v may be a simple variable x or either of the form (v_1 , v_2) or $(v_1 : v_2)$. Symbols ',' and ':' correspond to ones for operator symbols. These are right associative so that $(a:b:c)$ means $(a:(b:c))$.

$x \rightarrow x$

$(v_1 , v_2) \rightarrow (\lceil v_1 \rceil . \lceil v_2 \rceil)$

$(v_1 : v_2) \rightarrow (\lceil v_1 \rceil . \lceil v_2 \rceil)$

Expressions with local definitions d are translated as

let d_1 **and** $\cdots$ **and** d_n **in** $e \rightarrow$ (*let* $\lceil e \rceil$ $\lceil d_1 \rceil \cdots \lceil d_n \rceil$)

letrec d_1 **and** $\cdots$ **and** d_n **in** $e \rightarrow$ (*letrec* $\lceil e \rceil$ $\lceil d_1 \rceil \cdots \lceil d_n \rceil$)

e **where** d_1 **and** $\cdots$ **and** $d_n \rightarrow$ (*let* $\lceil e \rceil$ $\lceil d_1 \rceil \cdots \lceil d_n \rceil$)

e **whererec** d_1 **and** $\cdots$ **and** $d_n \rightarrow$ (*letrec* $\lceil e \rceil$ $\lceil d_1 \rceil \cdots \lceil d_n \rceil$)

A definition d may be a variable definition or a function definition:

$v = e \rightarrow (\lceil v \rceil . \lceil e \rceil)$

$x\ v_1 \cdots v_k = e \rightarrow (x . (lambda (\lceil v_1 \rceil \cdots \lceil v_k \rceil) \lceil e \rceil))$.

One of the most powerful notations in *uc* is the one for set comprehension called *ZF-expression* after Zermero-Fraenkel set theory. A ZF-expression is of the form

$[e \mid q_1 ; \cdots ; q_n]$

where each qualifier q_i is either a *generator* of the form 'x<–e' or a *guard* which is a predicate p (a Boolean expression) used to restrict the ranges of the values of the variables. The translation rules for the ZF-expression is summarized in Figure 8.3.1.

A simple example of a ZF-expression is a list of odd numbers

$[\, e \mid q_1 ; \cdots ; q_n \,]$ ZF-expression with qualifiers q_i

Qualifier:

$x \leftarrow e$	generator
p	predicate (Boolean expression)

Rules for translation of ZF-expressions

[1] $[\, e \mid x \leftarrow e' \,] \rightarrow (map\ (lambda\ (x)\ \lceil e \rceil)\ \lceil e' \rceil)$

[2] $[\, e \mid x \leftarrow e' ; p \,] \rightarrow (map\ (lambda\ (x)\ \lceil e \rceil)\ (filter\ (lambda\ (x)\ \lceil p \rceil)\ \lceil e' \rceil))$

[3] $[\, e \mid q_1 ; \cdots ; q_k ; p ; p' \,] \rightarrow \lceil [\, e \mid q_1 ; \cdots ; q_k ; (p\ \&\&\ p') \,] \rceil$

[4] $[\, e \mid q_1 ; \cdots ; q_k ; x \leftarrow e' \,] \rightarrow < s \mid q_1 ; \cdots ; q_k >$
where $s = \lceil [\, e \mid x \leftarrow e' \,] \rceil$

[5] $[\, e \mid q_1 ; \cdots ; q_k ; x \leftarrow e' ; p \,] \rightarrow < s \mid q_1 ; \cdots ; q_k >$
where $s = \lceil [\, e \mid x \leftarrow e' ; p \,] \rceil$

Auxiliary rules

<0> $< s \mid > \rightarrow s$

<1> $< s \mid x \leftarrow e' > \rightarrow (concmap\ (lambda\ (x)\ s)\ \lceil e' \rceil)$

<2> $< s \mid x \leftarrow e' ; p > \rightarrow (concmap\ (lambda\ (x)\ s)\ (filter\ (lambda\ (x)\ \lceil p \rceil)\ \lceil e' \rceil))$

<3> $< s \mid q_1 ; \cdots ; q_k ; p ; p' > \rightarrow < e \mid q_1 ; \cdots ; q_k ; (p\ \&\&\ p') >$

<4> $< s \mid q_1 ; \cdots ; q_k ; x \leftarrow e' > \rightarrow < s' \mid q_1 ; \cdots ; q_k >$
where $s' = < e \mid x \leftarrow e' >$

<5> $< s \mid q_1 ; \cdots ; q_k ; x \leftarrow e' ; p > \rightarrow < s' \mid q_1 ; \cdots ; q_k >$
where $s' = < e \mid x \leftarrow e' ; p >$

Library functions

$map\ f\ x =$ **if** $x == [\,]$ **then** $[\,]$ **else** $f\ a : map\ f\ u$
where$(a:u)=x$

$filter\ p\ x =$ **if** $x == [\,]$ **then** $[\,]$ **else**
if $p\ a$ **then** $p\ a : filter\ p\ u$ **else** $filter\ p\ u$
where$(a:u)=x$

$concmap\ f\ x =$ **if** $x == [\,]$ **then** $[\,]$ **else** $f\ a +\!\!+ concmap\ f\ u$
where$(a:u)=x$

Figure 8.3.1. Translation of ZF-expression

$[\, 2*n+1 \mid n \leftarrow [0..] \,]$

which is translated into an S-expression of the common intermediate language as

$(map\ (lambda\ (n)\ (add\ (mul\ (quote\ 2)\ n)\ (quote\ 1)))\ (from\ (quote\ 0)))$.

Another ZF-expression of a list of odd numbers looks like

$[\, n \mid n \leftarrow [0..] ; odd\ n \,]$.

This expression is transformed into

(*map* (*lambda* (*n*) *n*) (*filter* (*lambda* (*n*) (*odd n*)) (*from* (*quote* 0))))

which should be simplified as

(*filter* (*lambda* (*n*) (*odd n*)) (*from* (*quote* 0))) .

A variant of the ZF-expression $\{ e \mid q_1 ; \cdots ; q_k \}$ represents a list of which duplicated elements are made single.

$\{ e \mid q_1 ; \cdots ; q_k \} \rightarrow$ (*mkset* $\lceil [e \mid q_1 ; \cdots ; q_k] \rceil$)

where *mkset* is a library function defined as

mkset x = **if** x==[ ] **then** [ ] **else** a : *filter* (*neq a*) (*mkset u*) **where** $(a:u)=x$.

A set of pairs of A and B can be written as

$\{ (x,y) \mid x \leftarrow A ; y \leftarrow B \}$

which is transformed into

(*mkset* (*concmap* (*lambda* (*x*) (*map* (*lambda* (*y*) (*cons x y*)) *B*)) *A*)) .

Example programs developed in Chapters 1 and 2 have been presented by such compact notations of *uc* borrowed from Miranda [Turner85].

The front-end translator for the language *uc* has been developed using the *yacc* parser generator and the *lex* scanner generator of UNIX operating systems[3]. It is an easy task to write a front-end for a new language if these development tools are available. An example of translation can be found in Appendix A.

8.4. Experimental results

Several programs have been written and benchmarked on four different systems described in Section 8.2. Some of them have been mentioned in the previous Chapters.

The nFib benchmark

We have used the *nFib* function [Henderson83] in Chapter 4 to estimate the efficiency of the evaluators based on combinator reduction.

nFib n = **if** $n \leq 1$ **then** 1 **else** *nFib* $(n-1)$+*nFib* $(n-2)$+1

The result of this function is the number of function calls done in the course of calculation. We can allocate

[3] Developed and Licensed by AT&T.

arbitrary number of cells for FLFM on the Sun-2 (MC68010, 10Mz), the Melcom 70/250, and the Melcom MX2000. The FLFM on the PC9801 (i8086, 8Mz) uses always 32K cells. We measured the timing with sufficient number of cells so that garbage collection does not take place on the first three machines. For the last one, we reduced the average time required for garbage collection from the total running time. The results are summarized in Table 8.4.1.

Table 8.4.1: Number of function calls per second by *nFib* benchmark

Machine	Sun-2	M70/250	MX2000	PC9801
Function calls per second	4378	3588	7108	2700

The combinator reducers have been experimented on the Melcom 70/250 and the fast reducer based on the fixed code scheme performs 2447 function calls per second. Our FLFM based code runs 50% faster than the combinator reducer on the same computer.

Finding the 30-th prime

As a typical example of using infinite lists implemented by lazy evaluation, the program for finding the 30-th prime number was written in *uc*.

```
nth 30 primes
    whererec {
      primes = sieve [2 .. ]
    and
      sieve (p:x) = p : sieve [n | n <- x; n%p !=0 ]
    and
      nth n (a:x) = if n==1 then a else nth (n-1) x
    }
```

To estimate the applicability of the FLFM based compilers, total running time including time for garbage collection was measured with different heap sizes on three machines. The results are shown in Table 8.4.2.

The results by the graph reduction evaluators are presented in Chapter 4. However, the run-time measured there does not include the time for garbage collection.

We also wrote a program in ML and processed by the compiler system on the Melcom 70/250 [Chujo84]. The language ML evaluates every expression in a *non-lazy* way, and the program is coded by

Table 8.4.2: Run-time in seconds for 30-th prime

Heap size	Sun-2	M70/250	MX2000
5K cells	0.7	1.65	0.66
10K cells	0.7	1.03	0.46
20K cells	0.5	0.92	0.40

producing closures to put off evaluation. The run-time was 4.18 seconds using the heap of 5K cells. FLFM runs about three times faster than the ML code.

Ramanujan's numbers

As an example of a sizable program, we measured the run-time and the amount of store claimed for printing first 10 Ramanujan's numbers. See Chapters 1 and 2 of this thesis for the specification of the problem. We had defined a program *ram* as

```
ram (sort_r 1)
  whererec {
      ram (x:(y:z)) =
        if sumcubes x == sumcubes y then (x,y) : ram(y:z) else ram(y:z)
    and
      sort_r k = (k,k) : merge_r [(k,b)| b<-[k+1..]] (sort_r (k+1))
    and
      merge_r (x:u) (y:v) =
        if sumcubes x <= sumcubes y then x : merge_r u (v:y) else y : merge_r (x:u) v
    and
      sumcubes (a,b) = a*a*a+b*b*b
  }
```

and wrote a program as

```
take 10 ram
```

where *take* is a library function which takes specified number of elements of the second argument.

The total number of cells allocated in the course of evaluation is 232111. The run-time varies with the heap sizes as shown in Table 8.4.3.

The FLFM on the MX2000 executes programs that caclulate integers very fast, but it is rather slow when programs manipulate data structures extensively. It is one of the anomalies of our FLFM on different machines.

Table 8.4.3: Run-time in seconds for 10 Ramanujan's numbers

Heap size	Sun-2	M70/250	MX2000
10K cells	27.3	35.4	35.2
20K cells	21.7	26.7	30.3
30K cells	19.5	25.9	29.0

In order to estimate the efficiency of the garbage collector of the FLFM, we have measured the time required for garbage collection on the Sun-2 using the above program. Accumulated number of active cells divided by the total time for garbage collection gives the average number of cells collected in a unit of time. The results are summarized in Table 8.4.4.

Table 8.4.4: Efficiency of garbage collection on the Sun-2 FLFM

Heap size (K cell)	Total time (second)	Number of GC (times)	Total time for GC (second)	Accumulated active cells	Collected cells per second	GC time ratio
10	27.3	40	10.2	165940	16269	37.3%
20	21.7	14	3.48	55928	16071	16.0%
30	19.5	8	1.98	33060	16697	10.2%
40	18.9	6	1.68	27693	16484	8.9%
50	18.4	4	1.14	17634	15468	6.2%

As mentioned above, the total number of cells required by the program is about 232 K cells and the net time, i.e., total time minus GC time, is approximately 17 seconds. Hence the average number of cells comsumed per second is 13 K cells, which is 80% of the collection capacity.

Chapter 9

Traversals of data structures

The use of higher order functions in functional programming opens up the possibility of defining functions by partial parametrization, and lazy evaluation brings out a new approach in programming methodology. This chapter describes a new transformation technique based on partial parametrization and fully lazy evaluation for eliminating multiple traversals of data structures. It uses no particular mechanisms in functional programming, whereas it transforms a wider class of programs into efficient ones than that proposed so far.

9.1. Motivation

One of the most important features in functional programming is the use of higer order functions as a powerful mechanism of abstraction. Many similar functions can be defined by parametrizing a higher order function that represents a common pattern of computation. The advantage of programming this way is that modularity can be achieved without introducing any new mechanism except for functional abstraction and application. Such a modular style of programming is of increasing importance in writing large-scale programs. However, programs consisting of these functions sometimes turn out to be inefficient to execute. The main source of the inefficiency lies in the way functions are defined in a program. Each function is usually defined independently by instantiating the common pattern with no reference to the other functions. For example, consider a higher order function that represents a traversal algorithm of a certain data structure. This function is supposed to have a parameter for the operation on each element of the data structure. Then we can define various functions that operate on the data structure by instantiating different operations for the parameter, and use them to construct a program. It is true that the program written in this way attains a high degree of modularity, but multiple traversals of the data structure are inevitable if the program contains multiple instances of the higher order traversal function.

Another important feature in functional programming is that programs can be improved by simple transformations. This is due to the principle of referential transparency of functional programs.

In this chapter, we illustrate how these features of functional programming bring unexpected gains in

efficiency. The purpose of the transformation technique proposed is, among others, directed to refinement of programs written in a modular style into efficient ones; programs that traverse a data structure more than once are transformed into ones that do so only once.

A similar transformation method by Bird deals with programs of a particular form of functional composition [Bird84]. It relies on lazy evaluation and local recursion to build a circular program structure. Our method can be applied to a wider class of programs than Bird's. It is based on partial parametrization and fully lazy evaluation in addition to local recursion assumed by Bird. Partial parametrization is a basic mechanism in modular programming as described above, and full laziness can be reduced to ordinary laziness by a simple transformation of programs. We can thus make many programs into efficient ones without any particular mechanisms.

We introduce the basic idea of our method through a simple example in Section 9.2. The new transformation technique is described in a general setting in Section 9.3. Further example programs are transformed in Section 9.4. More on the transformation rules are formulated in Section 9.5. Finally in Section 9.6 some remarks on related research are given with actual benchmarks using the *uc* language system described in Chapter 8.

9.2. Partial parametrization and fully lazy evaluation

Consider a program to find the average of the elements of an integer list x :

average x = *DIV* (*sum* x) (*length* x)
  whererec
    sum x = *IF* (*NULL* x) 0 (*PLUS* (*HEAD* x) (*sum* (*TAIL* x)))
  and
    length x = *IF* (*NULL* x) 0 (*PLUS* 1 (*length* (*TAIL* x)))

We use a ternary function *IF* for conditional expressions; *IF* **true** e_1 e_2=e_1 and *IF* **false** e_1 e_2=e_2 hold. The functions *PLUS* and *DIV* are binary arithmetic operations that return the sum and the quotient of two integers, respectively. *NULL* is a predicate that returns **true** when applied to a null list **nil**, and returns **false** otherwise. *HEAD* and *TAIL* are selectors for a nonnull list (*PREFIX* x y) with *HEAD* (*PREFIX* x y)=x, and *TAIL* (*PREFIX* x y)=y. Although predicates and selectors for data structures should have been defined more formally, we leave it for the later sections.

The above program *average* traverses the given list x twice; once for computing the *sum* of the elements and once for finding the *length* of the list. One way to make the program efficient is to combine the two functions *sum* and *length* into a function *sum-length*:

$$sum\text{-}length\ x = [sum\ x\ ,\ length\ x]$$

where $[a,b]$ represents a pair of a and b. A recursive definition of *sum-length* can be synthesized by the *unfold-fold* method [Burstall77] and the **where**-*abstraction*:

$$sum\text{-}length\ x = IF\ (NULL\ x)\ [0,0]\ [PLUS\ (HEAD\ x)s\ ,\ PLUS\ 1\ l]$$
$$\textbf{whererec}\ [s,l] = sum\text{-}length\ (TAIL\ x)$$

Having defined this function, we can rewrite the function *average* as

$$average\ x = DIV\ s\ l$$
$$\textbf{where}\ [s,l] = sum\text{-}length\ x$$
$$\textbf{whererec}\ sum\text{-}length\ x = \cdots$$

This program traverses the list x only once.

The above transformation method can be applied to programs of the form $h(f\ x)(g\ x)$, which we may call *S'-composition* after the S' combinator of combinator logic[1]. It is, however, largely dependent on the form of functional composition.

Bird [Bird84] extends this idea of the tupling and the unfold-fold methods to develop a technique for transforming programs of the *S-composition* form, i.e., $f\ x(g\ x)$, into ones that traverse a data structure x only once. Bird's transformation relies on *lazy evaluation* [Friedman76, Henderson76] and *local recursion* to build a circular program structure. We do not deal with such a program here, but explain the basic idea of our new transformation technique based on *partial parametrization* and *fully lazy evaluation* using the example program *average*. In later sections we will apply the technique to Bird's examples.

First of all, it should be observed that both the functions *sum* and *length* traverse the list x precisely in the same manner. Or rather, one can say that these definitions come out from a common pattern of computation. They differ only in the function that operates on the elements of the list. Hence, the uncommon operation having been made a parameter f, the common parts of these functions can be expressed as a

[1] Turner [Turner79] introduces the combinator S' for S' $h\ f\ g\ x = h(f\ x)(g\ x)$ as an extension to the standard S combinator for S $f\ g\ x = f\ x(g\ x)$. The combinator S' is also written as Φ in some books on combinatory logic. We prefer S' to Φ because the latter symbol is used differently in this chapter.

higher order function *accum* that accumulates the results of applying the function f to all the elements of the list x:

$$accum\ f\ x =$$
$$\quad IF\ (NULL\ x)\ 0\ (PLUS\ (f\ (HEAD\ x))\ (accum\ f\ (TAIL\ x)))$$

We can redefine the two functions using *accum*:

$$sum = accum\ \mathbf{I} \quad \textbf{where}\ \mathbf{I}\ x = x$$
$$length = accum\ (\mathbf{K}\ 1) \quad \textbf{where}\ \mathbf{K}\ x\ y = x$$

The use of higher order functions this way is so common that we can find many illustrative examples in the literatures on functional programming, e.g., [Burge75]. In fact, the function *accum* itself can be defined by instantiating another higher order function that represents a more general pattern of computation (See Section 9.4).

The transformation technique proposed in this chapter uses such a higher order function of which parameters are arranged so that the first one is the data structure x to be traversed. We accordingly use the following version of the accumulation function in place of *accum*:

$$accum'\ x\ f =$$
$$\quad IF\ (NULL\ x)\ 0\ (PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f))$$

and we rewrite the functions *sum* and *length* as

$$sum\ x = accum'\ x\ \mathbf{I}$$
$$length\ x = accum'\ x\ (\mathbf{K}\ 1)$$

Then, we can take advantage of an opportunity to make the common term $(accum'\ x)$ be shared by both of the functions. This term is a unary function obtained by parametrizing only the first argument of the binary function *accum'*. As has been done above, the higher order function that is derived from a commonly used one but takes the data structure as its first argument is denoted by the function name with a prime attached.

Using the above definitions, we can rewrite the original program as

$$average\ x = DIV\ (\xi\ \mathbf{I})\ (\xi\ (\mathbf{K}\ 1))$$
$$\quad \textbf{where}\ \xi = accum'\ x$$
$$\quad\quad \textbf{whererec}\ accum'\ x\ f = \cdots$$

Lazy evaluation with a call-by-need mechanism [Wadsworth71], or call-by-delayed-value [Vuillemin74] ensures that the term ξ appearing in both the arguments of *DIV* is evaluated only once.

Therefore, if the list x has been frozen in the function ξ=(*accum'* x) and no more reference to x occurs in evaluation of (ξf) for any function f, the new program *average* traverses the list x only once. If we were allowed to pre-compute ξ for a particular x, say [3;5], we obtain ξ in a single traversal of x as

$$\begin{aligned}
\xi &= accum'\ [3;5] \\
&= \lambda f.\ IF\ (NULL\ [3;5])0(PLUS\ (f\ (HEAD\ [3;5]))(accum'\ (TAIL\ [3;5])f\ )) \\
&= \lambda f.\ PLUS\ (f\ 3)(IF\ (NULL\ [5])0(PLUS\ (f\ (HEAD\ [5]))(accum'\ (TAIL\ [5])f\ ))) \\
&= \lambda f.\ PLUS\ (f\ 3)(PLUS\ (f\ 5)(IF\ (NULL\ [\,])0(\ \cdots\ ))) \\
&= \lambda f.\ PLUS\ (f\ 3)(PLUS\ (f\ 5)\ 0)
\end{aligned}$$

Using this value for ξ, we can compute the average of [3;5] according to the equation for *average* without traversing x any more. Precomputation is not satisfactory, however. It is costly to calculate (*accum'* x) as above each time the list x is given. Fortunately, the desired effect is achieved by the use of *fully lazy evaluation*; every expression is evaluated at most once after the variables in it have been bound [Hughes84]. In our particular case, fully lazy evaluation ensures that any expression containing the parameter x is evaluated at most once. With an algorithm similar to the one by Hughes for finding super-combinators [Hughes82], the definition of *accum'* can be rewritten as

$$\begin{aligned}
&accum'\ x = \Phi\ (IF\ (NULL\ x))\ (HEAD\ x)\ (accum'\ (TAIL\ x)) \\
&\quad \textbf{where}\ \ \Phi\ \beta\ \alpha_1\ \alpha_2 f = \beta\ 0\ (PLUS\ (f\ \alpha_1)\ (\alpha_2 f))
\end{aligned}$$

where Φ is a super-combinator, that is, a closed function with no free variables. The arguments of Φ are maximal terms dependent on x (See Section 9.3).

In lazy evaluation, the arguments of Φ are evaluated at most once as needed within the body of Φ. Once the argument has been evaluated, its result is retained in place for subsequent evaluation. We have thus attained full laziness by means of ordinary lazy evaluation with call-by-need semantics.

By way of illustration, we will trace the computational process for *average* [3;5]. In that, the function *DIV* forces both the operands be evaluated before division is taken, evaluation of the first operand (ξ **I**) proceeds as

$$\begin{aligned}
\xi\ \mathbf{I} &= accum'\ [3;5]\ \mathbf{I} \\
&= \Phi\ (IF\ (NULL\ [3;5]))\ (HEAD\ [3;5])\ (accum'\ (TAIL\ [3;5]))\ \mathbf{I} \\
&= (IF\ (NULL\ [3;5]))\ 0\ (PLUS\ (\mathbf{I}(HEAD\ [3;5]))\ (accum'\ (TAIL\ [3;5])\mathbf{I}))
\end{aligned}$$

The first term (*IF* (*NULL* [3;5])) is evaluated to become *IF-FALSE*. The function *IF-FALSE*=(*IF* **false**) satisfies *IF-FALSE* $e_1\ e_2$=e_2. Similarly, *IF-TRUE*=(*IF* **true**) and *IF-TRUE* $e_1\ e_2$=e_1. Once the result has been obtained, it supersedes the argument for later use. That is, ξ becomes

$$\xi = \Phi\ \textit{IF-FALSE}\ (\textit{HEAD}\ [3;5])\ (\textit{accum}'\ (\textit{TAIL}\ [3;5]))$$

and the term (ξ **I**) becomes

$$\xi\ \mathbf{I} = \textit{PLUS}\ (\mathbf{I}\ (\textit{HEAD}\ [3;5]))\ (\textit{accum}'\ (\textit{TAIL}\ [3;5])\ \mathbf{I})$$

Evaluation now proceeds to the arguments of *PLUS*. The first one is evaluated to be 3 and the second to be

$$\textit{accum}'\ (\textit{TAIL}\ [3;5])\ \mathbf{I} = \Phi\ (\textit{IF}\ (\textit{NULL}\ t))\ (\textit{HEAD}\ t)\ (\textit{accum}'\ (\textit{TAIL}\ t))\ \mathbf{I}$$

where t=*TAIL* [3;5] has not yet been evaluated. When that expression is evaluated, (*NULL* t) results in **false** and t becomes [5] as a side-effect. Thus we have

$$\xi = \Phi\ \textit{IF-FALSE}\ 3\ (\Phi\ \textit{IF-FALSE}\ (\textit{HEAD}\ [5])\ (\textit{accum}'\ (\textit{TAIL}\ [5])))$$

and

$$\xi\ \mathbf{I} = \textit{PLUS}\ (3\ (\Phi\ \textit{IF-FALSE}\ (\textit{HEAD}\ [5])\ (\textit{accum}'\ (\textit{TAIL}\ [5])\ \mathbf{I})))$$

Finally we obtain

$$\begin{aligned}\xi = {} & \Phi\ \textit{IF-FALSE}\ 3 \\ & (\Phi\ \textit{IF-FALSE}\ 5 \\ & \quad (\Phi\ \textit{IF-TRUE}\ (\textit{HEAD}\ [\,])\ (\textit{accum}'\ (\textit{TAIL}\ [\,]))))\end{aligned}$$

and

$$\xi\ \mathbf{I} = \textit{PLUS}\ 3\ (\textit{PLUS}\ 5\ 0) = 8$$

When the other argument of *DIV*, i.e., (ξ (**K** 1)), comes to be evaluated, the value ξ that has just been obtained is used. Note that the term

$$\Phi\ \textit{IF-TRUE}\ (\textit{HEAD}\ [\,])\ (\textit{accum}'\ (\textit{TAIL}\ [\,]))$$

is immediately reduced to 0 with no reference to *HEAD* or *TAIL* if some argument follows it. Since there is no more *NULL*, *HEAD*, or *TAIL* remaining in ξ, we get the value (ξ (**K** 1))=2 without any traversal of the list x.

As we have seen from the above example, our new transformation technique consists of the following novel ideas in functional programming:

(1) For the common expressions to share the result of partial parametrization of the higher order function.

(2) For the partially parametrized function to be evaluated recursively in a fully lazy way.

We assume hereafter that our functional language allows one to deal with partially parametrized functions as first class objects, and that evaluation is done lazily with the call-by-need mechanism. We have already demonstrated a technique of transforming a function definition into one that is evaluated in a fully lazy way using the ordinary lazy evaluation mechanism.

9.3. Transformation rules

Our transformation technique exemplified in the last section contains several stages. We describe them in a general setting to formulate the rules. Suppose that a naive functional program is given as

$f\ x = F$
whererec
$g_1\ x = \lambda v_1.\ G_1$
and
...
and
$g_k\ x = \lambda v_k.\ G_k$

where F contains terms $(g_i\ x)$ and constants. Each of G_i may well contain lambda variables v_i in addition to x and constants. In general, G_i may contain $(g_j\ x)$ for $i \neq j$, so the g's can be mutually recursive. We assume, however, in this section that G_i does not contain $(g_j\ x)$.

[Generalization]

The first thing to do in our transformation is to find a common recursive form for the traversal functions $g_1, ..., g_k$. The higher order function h with parameters $x, y_1, ..., y_k$ should be defined so that every traversal function concerned is a particular instance of h applied to x, and p actual arguments. Let h be defined recursively as

$$h\ x\ y_1 \cdots y_p = H$$

where H contains $h, x, y_1, ..., y_p$ and constants. Then we have

$$g_i\ x\ =\ \lambda v_i.\ h\ x\ a_1\ \cdots\ a_p \qquad \text{for } i = 1, \cdots, k$$

with $a_1, ..., a_p$ chosen appropriately.

[Substitution]

The next step is to replace all the terms $(g_i\ x)$ in F by

$$\lambda v_i.\ \xi\ a_1\ \cdots\ a_p$$

and to use **where**-abstraction getting

$$f\ x\ =\ F'$$
where
$$\xi = h\ x$$
whererec
$$h\ x\ y_1\ \cdots\ y_p = H$$

[Lambda-hoisting]

To realize full laziness by a call-by-need mechanism, transform the above definition for the function h by hoisting maximal free occurrences of expressions in H with respect to $y_1, ..., y_p$.

$$h\ x\ =\ \Phi\ b_1\ \cdots\ b_m$$
where
$$\Phi\ \beta_1\ \cdots\ \beta_m\ y_1\ \cdots\ y_p = H'$$

where b_i's stand for the expressions each of which occurs maximally free in H and contains only x, h and constants.

Among these stages, only the generalization is heuristic. The other two are done entirely mechanically. Generalization can be done by simple inspection for small programs, whereas general rules and algorithms are to be sought for sizable programs.

On the other hand, generalization may be rather straightforward when programs are constructed from a generic traversal function by means of partial parametrization. Constructing programs in such a style is, indeed, one of the advantageous features of functional programming. In our particular case, we have a special interest in traversal functions for certain data structures. As every traversal function necessarily depends on the concrete data structure, defining an abstract data type would be most appropriate. Using an

abstract type definition, the structure of data and the traversal functions on it can be encapsulated into a module. The data structure used in the previous section can be written in Standard ML [Milner84] as

```
abstype intlist = nil | PREFIX of int (int list)
  with
    val NULL nil = true | NULL (PREFIX _ _) = false
    and HEAD (PREFIX a x) = a
    and TAIL (PREFIX a x) = x
    and rec accum' x = Φ(IF (NULL x))(HEAD x)(accum' (TAIL x))
            where Φ β α1 α2 f = β 0 (PLUS (f α1) (α2 f))
end
```

We have used data constructors **nil** and *PREFIX* corresponding to [ ] and [;] in the previous example. It is true that there may be various kinds of traversal functions for a particular data structure, but generalization becomes trivial if all the $(g_i\ x)$ are expressed as instances of a generic traversal function in common. We will give some typical data structures and traversal functions in later sections.

The *lambda-hoisting* rule is much similar to the rule by Hughes [Hughes82] specifying how to convert a lambda expression into super-combinators[2]. It is slightly different, however, in the treatment of recursive definitions. We will give an algorithm for lambda-hoisting adapted to the definition for h specified as above. For simplicity, we assume here that the right hand side expression H is of the applicative form and it does not contain lambda expressions as its constituents. That is, an applicative expression is either

(1) *simple* and is either

  (1.1) a constant

  or

  (1.2) a variable

or

[2] The lambda-hoisting technique in more general setting has been fully developed in Chapter 6 of this thesis. Although the expression transformed by lambda-hoisting in this chapter is not of the *fully lazy normal form* (Chapter 5), there is no essential difference between the rules here and in Chapter 6.

(2) *compound* and is a combination $(e_1\ e_2)$ where both e_1 and e_2 are applicative expressions.

Here, we regard fixed functions like *IF*, *PLUS*, **I**, etc., as constants. We first define *free occurrences* of applicative expressions.

An occurence of an expression e in H is defined to be *free* with respect to $y_1, ..., y_p$ as follows.

(1) If e is simple and

(1.1) if e is a constant, e is not free in H,

or

(1.2) if e is a variable, and

(1.2.1) if e is the function variable h or the variable x, e is *free* in H with respect to $y_1, ..., y_p$,

or

(1.2.2) e is not free in H, otherwise.

(2) If e is compound and is a combination $(e_1\ e_2)$, and

(2.1) if either of e_1 or e_2 is free in H, and

(2.1.1) the other one is also free in H, or the other one is a constant, e is *free* in H with respect to $y_1, ..., y_p$.

or

(2.1.2) e is not free in H, otherwise.

(2.2) e is not free in H, otherwise.

Free occurrences of expressions that are not part of any larger free occurrence of an expression are called *maximal free occurrences* of expressions in H with respect to $y_1, ..., y_p$.

We assume that all the operations usually written in infix form are expressed by corresponding functions, and the conditional expression is also written using the function *IF* with three arguments. Although there is no problem arisen from this definition, it would be desirable to deal with conditionals as a special form. That is, even in the case that $(IF\ e_1\ e_2)$ happens to be a maximal free occurrence from the above

definition, we take both the parts $(IF\ e_1)$ and e_2 as maximal free occurrences.

We can now describe the algorithm for lambda-hoisting.

(Step1) Identify all the maximal free occurrences of expressions $b_1, ..., b_m$ in H with respect to $y_1, ..., y_p$.

(Step2) Replace each occurrence of b_i in H with a new variable β_i for $i=1, \cdots, m$ to obtain H'.

(Step3) Construct the definition of the form

$$h\ x\ =\ \Phi\ b_1\ \cdots\ b_m\ \textbf{where}\ \Phi\ \beta_1\ \cdots\ \beta_m\ y_1\ \cdots\ y_p = H'$$

If there exist common sub-expressions in $b_1, ..., b_m$, the expression $\Phi\ b_1\ \cdots\ b_m$ should be further transformed using **where**-abstraction so that they are replaced by the common variables bound to those expressions.

It is desirable, though not absolutely necessary, that identical expressions are not duplicated among $b_1, ..., b_m$ in (Step1). For example, consider the case where there are two maximal free occurrences of $(HEAD\ x)$ in H. If we take them to be distinctively, say b_1 and b_2, we cannot have the advantage of call-by-need evaluation at least as it is. After one of them has been evaluated to yield the value of $(HEAD\ x)$, the result cannot be used in evaluation of the other one. In fact, (Step3) ensures that such situation is properly dealt with for completing lambda-hoisting. The number of parameters is, of course, greater than the case where common expressions are made into one in (Step1). On the other hand, even if no duplicated b_i have been extracted in (Step1), (Step3) is indispensably necessary. This is because there may exist an expression b_i identical to some proper sub-expression of another expression b_j both of which occur maximally free in H.

As mentioned earlier, lambda-hoisting is intended to realize full laziness by ordinary lazy evaluation. The procedure essentially converts an expression into a new form of expression that uses functions with no free variables. In that, it is most important that maximal free occurrences of expressions are identified and they are moved out as arguments of the functions. This is the basic idea of super-combinators by Hughes. It should be noted, however, that the lambda-hoisting rule differs from that of Hughes for recursive definitions. According to [Hughes82], the definition of *accum'* in the previous section can be rewritten as

$$\begin{aligned} accum'\ x &= \lambda f.\ IF\ (NULL\ x)\ 0\ (PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f)) \\ &= IF\ (NULL\ x)\ 0\ (\lambda f.PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f)) \end{aligned}$$

$$= IF\ (NULL\ x)\ 0\ ((\lambda x.\lambda f.\ PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f))\ x)$$

and we have a super-combinator Ψ such that

$$accum'\ x = IF\ (NULL\ x)\ 0\ (\Psi\ x)$$
$$\textbf{whererec}\ \Psi\ x\ f = PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f)$$

In case of x=[3;5], we get

$$\xi\ \mathbf{I} = accum'\ [3;5]\ \mathbf{I}$$
$$= IF\ (NULL\ [3;5])\ 0\ (\Psi\ [3;5])\ \mathbf{I}$$
$$= \Psi\ [3;5]\ \mathbf{I}$$
$$= PLUS\ (\mathbf{I}\ (HEAD\ [3;5]))\ (accum'\ (TAIL\ [3;5])\ \mathbf{I})$$

and the computation proceeds as before to yield the value (ξ **I**)=8. But the value of ξ after evaluation of (ξ **I**) is simply (Ψ[3;5]), and no further reduction has been taken. This means that another traversal of the data structure is needed in evaluation of (ξ (**K** 1)).

Yet another transformation rule called *lambda-lifting* is presented in [Johnsson85]. The idea of identifying maximal free occurrences of expressions is not included in lambda-lifting. If we extrapolate the idea of lambda-lifting and move out maximal free occurrences of expressions instead of variables alone, we have

$$accum'\ x = \Phi'\ (IF\ (NULL\ x))\ (HEAD\ x)\ (TAIL\ x)$$
$$\textbf{whererec}\ \Phi'\ \beta\ \alpha_1\ \alpha_2\ f = \beta\ 0\ (PLUS\ (f\ \alpha_1)\ (accum'\ \alpha_2\ f))$$

This form of definition fails again to achieve our end. The situation is much similar to the case of Hughes' algorithm for super-combinators.

In summary, it is essential in lambda-hoisting that the variable h representing the traversal function and the variable x for the data structure should be taken to be free with respect to other variables as carefully described in the definition of free occurrences of expressions. As a matter of fact, we have made the data structure be the first argument of the traversal function h so that the combination of h and the expression for selecting some part of the data structure x become a single larger free expression.

9.4. Functions on some data structures

In this section we will illustrate how the transformation rules are applied to programs that traverse particular data structures. We have already presented our transformation technique for a program of the

S'-composition form in Section 9.2. We can equally transform programs of the S-composition form using the rules in the previous section. Example programs in this section are taken from [Bird84] for comparing our transformation technique with Bird's.

9.4.1. Functions on lists

To begin with, consider a definition of a general list type

```
abstype 'a list = nil | PREFIX of 'a ('a list)
  with
    val NULL nil = true | NULL (PREFIX _ _) = false
    and HEAD (PREFIX a x) = a
    and TAIL (PREFIX a x) = x
    and rec ···
end
```

Traversal functions to be defined in this abstract type are, among others, ones that scan a single list of the type *'a list*. We first define these functions in a commonly used form, and then derive the final equations by exchanging the order of parameters. Such functions are from [Burge75]:

$$list1\ a\ g\ f\ x = IF\ (NULL\ x)\ a\ (g\ (f\ (HEAD\ x))\ (list1\ a\ g\ f\ (TAIL\ x)))$$
$$list2\ a\ g\ f\ x = IF\ (NULL\ x)\ a\ (list2\ (g\ (f\ (HEAD\ x))\ a)\ g\ f\ (TAIL\ x))$$

The functions *list*1 and *list*2 are more general than the function *accum* used in Section 9.2. In fact, we have

$$accum = list1\ 1\ 0\ PLUS$$
$$accum = list2\ 0\ PLUS$$

It should be noted, however, that we can derive the recursion equation of *accum* from *list*1 by replacing the term (*list*1 *a g*) on the right hand side with *accum* itself, while we cannot do it from *list*2.

These functions can be used to define many functions. For example, commonly used higher order functions *map* and *filter*:

$$map\ f\ x = IF\ (NULL\ x)\ \mathbf{nil}\ (PREFIX\ (f\ (HEAD\ x))\ (map\ f\ (TAIL\ x))$$

$$filter\ p\ x = IF\ (NULL\ x)\ \mathbf{nil}\ (IF\ (p\ u)\ (PREFIX\ u\ v)\ v)$$
$$\mathbf{whererec}\ u = HEAD\ x\ \mathbf{and}\ v = filter\ p\ (TAIL\ x)$$

can be defined using *list*1 as

$map = list\,1\ \mathbf{nil}\ PREFIX$

$filter\ p = list\,1\ \mathbf{nil}\ (\lambda uv.IF\ (p\ u)\ (PREFIX\ u\ v)\ v)\ \mathbf{I}$

And the function *reverse* that reverses the given list can be written using $list\,2$ as

$reverse = list\,2\ \mathbf{nil}\ PREFIX\ \mathbf{I}$

We need sometimes functions that operate two lists of the same length as in comparing corresponding elements of the lists. Hence, we next define such functions $zip\,1$ and $zip\,2$ that scan lists x and y of the same length. These are based on $list\,1$ and $list\,2$, respectively.

$$zip\,1\ a\ g\ f\ x\ y =$$
$$\quad IF\ (NULL\ x)\ a\ (g\ (f\ (HEAD\ x)\ (HEAD\ y))\ (zip\,1\ a\ g\ f\ (TAIL\ x)\ (TAIL\ y)))$$

$$zip\,2\ a\ g\ f\ x\ y =$$
$$\quad IF\ (NULL\ x)\ a\ (zip\,2\ (g\ (f\ (HEAD\ x)\ (TAIL\ y))\ a)\ g\ f\ (TAIL\ x)\ (TAIL\ y))$$

Here, the function g combines successively the results of f applied to corresponding elements of the lists x and y. The functions $list\,1$ and $list\,2$ are defined using $zip\,1$ and $zip\,2$ as

$$list\,1\ a\ g\ f\ x = zip\,1\ a\ g\ \bar{f}\ x\ DUMMY$$
$$list\,2\ a\ g\ f\ x = zip\,2\ a\ g\ \bar{f}\ x\ DUMMY$$

where

$$\bar{f}\ x\ y = f\ x$$

and *DUMMY* is a dummy expression never to be evaluated though its presence is necessary.

We now turn to defining the final equations for these functions to make the proposed transformation technique be applicable. To do so, we have only to promote the parameter x at the front of the parameter list. As noted in Section 9.2, we write such function names with primes.

$$list\,1'\ x\ a\ g\ f = IF\ (NULL\ x)\ a\ (g\ (f\ (HEAD\ x))\ (list\,1'\ (TAIL\ x)\ a\ g\ f))$$
$$list\,2'\ x\ a\ g\ f = IF\ (NULL\ x)\ a\ (list\,2'\ (TAIL\ x)\ (g\ (f\ (HEAD\ x))\ a)\ g\ f)$$

$$zip\,1'\ x\ a\ g\ f\ y =$$
$$\quad IF\ (NULL\ x)\ a\ (g\ (f\ (HEAD\ x)\ (HEAD\ y))\ (zip\,1'\ (TAIL\ x)\ a\ g\ f\ (TAIL\ y)))$$

$$zip\,2'\ x\ a\ g\ f\ y =$$
$$\quad IF\ (NULL\ x)\ a\ (zip\,2'\ (TAIL\ x)\ (g\ (f\ (HEAD\ x)\ (HEAD\ y))\ a)\ g\ f\ (TAIL\ y))$$

By lambda-hoisting, we get the equations

$list1'\ x = \Phi\ (IF\ (NULL\ x))\ (HEAD\ x)\ (list1'\ (TAIL\ x))$

where $\Phi\ \beta\ \alpha\ \eta\ a\ g\ f = \beta\ a\ (g\ (f\ \alpha)(\eta\ a\ g\ f))$

$list2'\ x = \Phi\ (IF\ (NULL\ x))\ (HEAD\ x)\ (list2'\ (TAIL\ x))$

where $\Phi\ \beta\ \alpha\ \eta\ a\ g\ f = \beta\ a\ (\eta\ (g\ (f\ \alpha)\ a)\ g\ f)$

$zip1'\ x = \Phi\ (IF\ (NULL\ x))\ (HEAD\ x)\ (zip1'\ (TAIL\ x))$

where $\Phi\ \beta\ \alpha\ \eta\ a\ g\ f\ y = \beta\ a\ (g\ (f\ \alpha\ (HEAD\ y))(\eta\ a\ g\ f\ (TAIL\ y)))$

$zip2'\ x = \Phi\ (IF\ (NULL\ x))\ (HEAD\ x)\ (zip2'\ (TAIL\ x))$

where $\Phi\ \beta\ \alpha\ \eta\ a\ g\ f\ y = \beta\ a\ (\eta\ (g\ (f\ \alpha\ (HEAD\ y))\ a)\ g\ f\ (TAIL\ y))$

We assume here that these definitions are included in the above abstract type definition of *'a list*.

As an example of list traversal programs, consider the palindrome problem in [Bird84]: Determine whether a given list of integers is palindromic; i.e., equal to its reverse.

A straightforward solution looks like

```
palindrome x = eqlist x (reverse x)
    whererec
      eqlist x y =
        IF (NULL x) true
          (AND (EQUAL (HEAD x) (HEAD y)) (eqlist (TAIL x) (TAIL y)))
    and
      reverse x = reverse' x nil
    and
      reverse' x z = IF (NULL x) z (reverse' (TAIL x) (PREFIX (HEAD x) z))
```

The auxiliary functions *eqlist* and *reverse* can be readily expressed using *zip*2' as

```
eqlist x y = zip2' x true AND EQUAL y
reverse x = zip2' x nil PREFIX K DUMMY
```

Note that the definition of *reverse* is derived from the one using *list*2 and the equation combining *list*2 and *zip*2. We have thus finished the generalization step. The final program can be obtained by substitution.

```
palindrome x = ξ true AND EQUAL (ξ nil PREFIX K DUMMY)
    where ξ = zip2' x
```

Lambda-hoisting of the function *zip*2' has been completed in the definition of the type *'a list*.

9.4.2. Functions on trees

We shall consider another data structure *'a tree*:

```
abstype 'a tree = TIP of 'a | FORK of ('a tree) ('a tree)
  with
    val ISTIP (TIP _) = true | ISTIP (FORK _ _) = false
    and TIPVAL (TIP a) = a
    and LEFT (FORK l r) = l
    and RIGHT (FORK l r) = r
    and rec
      btree' x = Φ (IF (ISTIP x)) (TIPVAL x) (btree' (LEFT x)) (btree' (RIGHT x))
        where Φ β v η ζ g f = β (f v) (g (η g f) (ζ g f))
end
```

Here, the function *btree'* is based on a familiar form of tree traversal functions

```
btree g f x =
    IF (ISTIP x) (f (TIPVAL x)) (g (btree g f (LEFT x)) (btree g f (RIGHT x)))
```

that applies the function f to each tip of the tree x and combines the results of doing so on left and right subtrees by the function g over the tree.

We can solve the tree replacement problem in [Bird84] using the traversal function *btree'*. The problem is to change a given binary tree into one identical in shape to the first, but with all the tip values replaced by the minimum of the tip values of the first. For example, the tree of Figure 1 is changed into that of Figure 2. (Figure 3 will be referred in Section 9.5.)

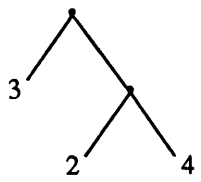


Figure 1

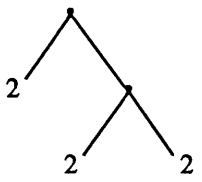


Figure 2

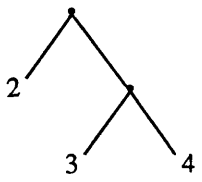


Figure 3

A straightforward solution to this problem would be

$transform\ x = replace\ x\ (tmin\ x)$

whererec

$replace\ x\ m =$

$IF\ (ISTIP\ x)\ (TIP\ m)\ (FORK\ (replace\ (LEFT\ x)\ m)\ (replace\ (RIGHT\ x)\ m))$

and

$tmin\ x = IF\ (ISTIP\ x)\ (TIPVAL\ x)\ (MIN\ (tmin\ (LEFT\ x))\ (tmin\ (RIGHT\ x)))$

where *MIN* yields the minimum of the two integers.

We can define the auxiliary functions *replace* and *tmin* in terms of *btree'* as

$replace\ x\ m = btree'\ x\ FORK\ (\lambda u.TIP\ m)$

$tmin\ x = btree'\ x\ MIN\ \mathbf{I}$

By substitution, we get

$transform\ x = \xi\ FORK\ (\lambda u.TIP\ (\xi\ MIN\ \mathbf{I}))$

where $\xi = btree'\ x$

It is easy to see that this program traverses the tree x only once as in the case of list traversal. It is because our transformation technique does not depend on the data structure itself, but on the traversal function. There are many programs on tree structures that can be transformed similarly.

The original *transform* is of the S-composition form dealt with by Bird. We sketch here Bird's transformation method for comparison. To apply the method to this problem, we define a new function that produces a pair from a tree and an integer

$repmin\ t\ m = PAIR\ (replace\ t\ m)\ (tmin\ t)$

Once this function has been defined properly, the function can be rewritten as

$transform\ t = FST\ p$

whererec $p = repmin\ t\ (SND\ p)$

where $FST\ (PAIR\ x\ y)=x$ and $SND\ (PAIR\ x\ y)=y$. The reader should refer to [Bird84] for the reasoning by which this has been derived from the definition of *repmin*. The recursive definition of *repmin* can be obtained using the standard unfold-fold method [Burstall77].

repmin t m =

IF (*ISTIP t*) (*PAIR* (*TIP m*) (*TIPVAL t*)) (*PAIR* (*FORK t*1 *t*2) (*MIN m*1 *m*2))

whererec

$t1 = FST\ r1$ **and** $m1 = SND\ r1$ **where** $r1 = repmin\ (LEFT\ t)\ m$

and

$t2 = FST\ r2$ **and** $m2 = SND\ r2$ **where** $r2 = repmin\ (RIGHT\ t)\ m$

Using this function, we can carry out the tip replacement in a single traversal of the tree.

Table 9.4.1 contains the number of primitive operations performed to print out the resulting tree of Figure 2 when applying the function *transform* to the tree of Figure 1. The effect of transformations is clearly seen from it.

Table 9.4.1: Number of primitive operations by *transform*

Operation	Program		
	Straightforward	Ours	Bird's
IF	10	5	5
IF-TRUE	6	6	3
IF-FALSE	4	4	2
ISTIP	10	5	5
TIPVAL	3	3	3
LEFT	4	2	2
RIGHT	4	2	2
MIN	2	2	2
FORK	2	2	2
TIP	3	1	3
PAIR	-	-	5
FST	-	-	5
SND	-	-	5

9.5. More on transformation rules

We have excluded so far the case where the auxiliary traversal functions are mutually recursive. It is, however, too restrictive in practice as illustrated in the next example.

Consider a problem in [Hughes85]: Find the set of the deepest tips in a tree, where the set is represented by a list. A naive solution can be

```
deepest x =
    IF (ISTIP x) (PREFIX (TIPVAL x) nil)
      (IF (GREATER (depth l) (depth r)) (deepest l)
        (IF (LESS (depth l) (depth r)) (deepest r)
          (APPEND (deepest l) (deepest r))))
      whererec
        l = LEFT x  and  r = RIGHT x
      and
        depth x =
          IF (ISTIP x) 0 (PLUS 1 (MAX (depth (LEFT x)) (depth (RIGHT x))))
```

The function *APPEND* concatenates two lists and the function *MAX* gives the maximum of the two arguments. Here we can see that the function *deepest* does not only depend on itself, but also on the function *depth*. We have not yet established the rules for such cases.

There can be several possibilities of extending the idea of our transformation technique to deal with such cases. One way to do this would be to define general computational rules for the tuple $[g_1, \cdots, g_k]$ of the auxiliary functions. This may raise a new problem of defining the meaning of functional application $([g_1, \cdots, g_k]\, x)$ consistently. So, we shall deal with mutual recursion of the auxiliary functions differently. Although this approach is applicable to any data structure in general, we focus here on the tree structure.

Consider the general form of tree traversal programs.

```
f x = F
    whererec
      g1 x = IF (ISTIP x) (ψ1 (TIPVAL x)) s1
    and
        ...
    and
      gk x = IF (ISTIP x) (ψk (TIPVAL x)) sk
```

Each function s_i may contain g_j $(j \neq i)$ as well as g_i. We can assume here, however, that only the terms (*LEFT* x) and (*RIGHT* x) are the arguments of g_i and g_j in s_i, and no other terms do not contain the variable x. This implies that there is no term dependent on x directly nor indirectly except (*LEFT* x) and (*RIGHT* x). Since the auxiliary function g_i traverses the data structure x, such assumption is very natural in practice. Hence, we can define a higher order function common to all the g_i's using the following rules.

[Generalization with Lambda-hoisting]

Rewrite each s_i of the auxiliary function

$$g_i\ x = IF\ (ISTIP\ x)\ (\psi_i\ (TIPVAL\ x))\ s_i$$

as

$$\begin{aligned} s_i = \sigma_i\ & (g_i\ l)\ (g_i\ r) \\ & (g_1\ l)\ (g_1\ r) \\ & \cdots \\ & (g_{i-1}\ l)\ (g_{i-1}\ r) \\ & DUMMY\ DUMMY \\ & (g_{i+1}\ l)\ (g_{i+1}\ r) \\ & \cdots \\ & (g_k\ l)\ (g_k\ r) \end{aligned}$$

where

$$l = LEFT\ x \ \textbf{and}\ r = RIGHT\ x$$

where σ_i is a combinator, that is, a closed lambda term composed of only lambda variables, and constants. The first pair of the arguments $(g_i\ l)$ and $(g_i\ r)$ corresponds to the self-recursive terms within s_i. The other arguments represent mutual recursive terms. Two *DUMMY* arguments appear at i-th position for σ_i. This process can be done using a variant of the lambda-hoisting technique. The arity of the combinator is common to all the σ_i, and equal to $2k+2$.

Then the common recursive form can be written as

$$\begin{aligned} & h\ x\ \psi\ \sigma = \\ & \quad IF\ (ISTIP\ x)\ (\psi\ (TIPVAL\ x)) \\ & \quad\quad (\sigma\ (\eta\ \psi\ \sigma)\ (\zeta\ \psi\ \sigma) \\ & \quad\quad\quad (\eta\ \psi_1\ \sigma_1)\ (\zeta\ \psi_1\ \sigma_1)\ \cdots\ (\eta\ \psi_k\ \sigma_k)\ (\zeta\ \psi_k\ \sigma_k)\ ) \\ & \quad\quad \textbf{where} \\ & \quad\quad\quad \eta = h\ (LEFT\ x)\ \textbf{and}\ \zeta = h\ (RIGHT\ x) \end{aligned}$$

[Substitution]

Replace all the terms $(g_i\ x)$ in F by

$$\xi\ \psi_i\ \sigma_i \quad \textbf{where}\ \ \xi = h\ x$$

getting F'.

[Lambda-hoisting]

By applying the lambda-hoisting procedure to the function h above, we obtain the final form

$f\ x = F'$
  where
    $\xi = h\ x$
      whererec
        $h\ x = \Phi\ (IF\ (ISTIPx))\ (TIPVAL\ x)\ (h\ (LEFT\ x))\ (h\ (RIGHT\ x))$
          where
            $\Phi\ \beta\ \nu\ \eta\ \zeta\ \psi\ \sigma =$
              $\beta\ (\psi\ \nu)$
                $(\sigma\ (\eta\ \psi\ \sigma)\ (\zeta\ \psi\ \sigma)$
                  $(\eta\ \psi_1\ \sigma_1)\ (\zeta\ \psi_1\ \sigma_1)\ \cdots\ (\eta\ \psi_k\ \sigma_k)\ (\zeta\ \psi_k\ \sigma_k))$

Although the function h can be defined in the abstract type *'a tree*, it would be less general than the higher order functions defined so far. Firstly, the function h is dependent on the number k of the auxiliary functions. And it is observed that only some part of the terms $(\eta\ \psi_i\ \sigma_i)$ and $(\zeta\ \psi_i\ \sigma_i)$ are actually needed as will be shown in the examples. In fact, redundant terms should not be included to keep the number of arguments as small as possible. Omission of unnecessary terms makes gains in execution time and space. Therefore, we will give a particular definition of h adapted for each problem.

We now try to transform the program *deepest* presented at the beginning of this section. Observe that the auxiliary functions *deepest* and *depth* can be rewritten as

deepest x =
  IF (*ISTIP x*) (ψ_1 (*TIPVAL x*)) (σ_1 (*deepest l*) (*deepest r*) (*depth l*) (*depth r*))
    where
      $\psi_1\ \nu$ = *PREFIX* ν **nil**
    and
      $\sigma_1\ \eta\ \zeta\ \alpha_1\ \alpha_2 =$
        IF (*GREATER* $\alpha_1\ \alpha_2$) η (*IF* (*LESS* $\alpha_1\ \alpha_2$) ζ (*APPEND* $\eta\ \zeta$))

$depth\ x =$

$IF\ (ISTIP\ x)\ (\psi_2\ (TIPVAL\ x))\ (\sigma_2\ (depth\ l)\ (depth\ r)\ DUMMY\ DUMMY)$

where

$\psi_2\ v = 0$

and

$\sigma_2\ \eta\ \zeta\ \alpha_1\ \alpha_2 = PLUS\ 1\ (MAX\ \eta\ \zeta)$

where l=$(LEFT\ x)$ and r=$(RIGHT\ x)$. We have assigned *deepest* to g_1 and *depth* to g_2. According to the general form of s_i, s_1 and s_2 in this case should have been

$s_1 = \sigma_1\ (deepest\ l)\ (deepest\ r)\ DUMMY\ DUMMY\ (depth\ l)\ (depth\ r)$

$s_2 = \sigma_2\ (depth\ l)\ (depth\ r)\ (deepest\ l)\ (deepest\ r)\ DUMMY\ DUMMY$

But as mentioned earlier, s_2 does not contain any term on *deepest* and the arguments $(deepest\ l)$ and $(deepest\ r)$ of σ_2 are redundant. Hence we have omitted the third and fourth arguments of both σ_1 and σ_2. From the above definitions, we get the final program

$deepest\ x = \xi\ \psi_1\ \sigma_1$

where

$\xi = h\ x$

whererec

$h\ x = \Phi\ (IF\ (ISTIP\ x))\ (TIPVAL\ x)\ (h\ (LEFT\ x))\ (h\ (RIGHT\ x))$

where

$\Phi\ \beta\ v\ \eta\ \zeta\ \psi\ \sigma =$

$\beta\ (\psi\ v)\ (\sigma\ (\eta\ \psi\ \sigma)\ (\zeta\ \psi\ \sigma)\ (\eta\ \psi_2\ \sigma_2)\ (\zeta\ \psi_2\ \sigma_2))$

where

$\psi_2\ v = \cdots$ **and** $\sigma_2\ \eta\ \zeta\ \alpha_1\ \alpha_2 = \cdots$

and

$\psi_1\ v = \cdots$ **and** $\sigma_1\ \eta\ \zeta\ \alpha_1\ \alpha_2 = \cdots$

Our next example is again a tree replacement problem: Transform a binary tree into one of the same shape of which tip values are those of the original tree arranged in increasing order. For example, the tree of Figure 1 is to be transformed into that of Figure 3.

A solution with refinement by transformational programming is given in [Bird84].

$transform\ x = replace\ x\ (sort\ (tips\ x))$

whererec

$replace\ x\ z =$

$IF\ (ISTIP\ x)\ (TIP\ (HEAD\ z))$

$(FORK\ (replace\ (LEFT\ x)\ (take\ (size\ (LEFT\ x))\ z))$

$(replace\ (RIGHT\ x)\ (drop\ (size\ (LEFT\ x))\ z)))$

and

$size\ x = IF\ (ISTIP\ x)\ 1\ (PLUS\ (size\ (LEFT\ x))\ (size\ (RIGHT\ x)))$

and

$tips\ x = ntips\ x$ **nil**

and

$ntips\ x\ z =$

$IF\ (ISTIP\ x)\ (PREFIX\ (TIPVAL\ x)\ z)\ (ntips\ (LEFT\ x)\ (ntips\ (RIGHT\ x)\ z))$

The functions *take* and *drop* over list structures are defined as

$take\ n\ z = IF\ (EQUAL\ n\ 0)$ **nil** $(PREFIX\ (HEAD\ z)\ (take\ (MINUS\ n\ 1)(TAIL\ z)))$

$drop\ n\ z = IF\ (EQUAL\ n\ 0)\ z\ (drop\ (MINUS\ n\ 1)\ (TAIL\ z))$

Although there is much room for improvement on these functions, we confine ourselves to transformation for eliminating multiple traversals of the binary tree. We assume that an efficient sorting function *sort* exists.

Here, we illustrate the transformation process in another way; define first the function h, and then rewrite the auxiliary functions in terms of h. This is the way of programming with higher order generic functions. In this case, we need to include only the term $(size\ (LEFT\ x))$ as an argument to the combinator σ in the general form. Taking this into account, we have a common recursive form for the program *transform*

$h\ x = \Phi\ (IF\ (ISTIP\ x))\ (TIPVAL\ x)\ (h\ (LEFT\ x))\ (h\ (RIGHT\ x))$

where

$\Phi\ \beta\ \nu\ \eta\ \zeta\ \psi\ \sigma =$

$\beta\ (\psi\ \nu)\ (\sigma\ (\eta\ \psi\ \sigma)\ (\zeta\ \psi\ \sigma)\ (\eta\ \psi_2\ \sigma_2))$

where ψ_2 and σ_2 come from *size*.

Using the function h, we can rewrite the auxiliary functions:

$replace\ x = h\ x\ \psi_1\ \sigma_1$

where

$\psi_1\ v = \lambda u.\ TIP\ (HEAD\ u)$

and

$\sigma_1\ \eta\ \zeta\ \alpha = \lambda u.\ FORK\ (\eta\ (take\ \alpha\ u))\ (\zeta\ (drop\ \alpha\ u))$

$size\ x = h\ x\ \psi_2\ \sigma_2$

where

$\psi_2\ v = 1$

and

$\sigma_2\ \eta\ \zeta\ \alpha = PLUS\ \eta\ \zeta$

$ntips\ x = h\ x\ \psi_3\ \sigma_3$

where

$\psi_3\ v = \lambda u.\ PREFIX\ v\ u$

and

$\sigma_3\ \eta\ \zeta\ \alpha = \lambda u.\ \eta\ (\zeta\ u)$

By substitution we obtain the final program

$transform\ x = \xi\ \psi_1\ \sigma_1\ (sort\ (\xi\ \psi_3\ \sigma_3\ \mathbf{nil}))$

where

$\xi = h\ x$

whererec

$h\ x = \Phi\ (IF\ (ISTIP\ x))\ (TIPVAL\ x)\ (h\ (LEFT\ x))\ (h\ (RIGHT\ x))$

where

$\Phi\ \beta\ v\ \eta\ \zeta\ \psi\ \sigma =$

$\beta\ (\psi\ v)\ (\sigma\ (\eta\ \psi\ \sigma)\ (\zeta\ \psi\ \sigma)\ (\eta\ \psi_2\ \sigma_2))$

where

$\psi_2\ v = \cdots$ **and** $\sigma_2\ \eta\ \zeta\ \alpha = \cdots$

and

$\psi_1\ v = \cdots$ **and** $\sigma_1\ \eta\ \zeta\ \alpha = \cdots$

and

$\psi_3\ v = \cdots$ **and** $\sigma_3\ \eta\ \zeta\ \alpha = \cdots$

The transformation rules above can be applied in a systematic way; no heuristic process employed.

9.6. Remarks

We wrote two programs for the *deepest* problem in the previous section; a straightforward program and a transformed program. Both programs were written in the language *uc* and compiled as described in Chapter 8. We measured the run-time and the store claimed to find the number of the deepest leaves of the balanced tree with 1024 leaves. Table 9.6.1 shows the results obtained on the FLFM on Sun-2. No garbage collection were taken.

Table 9.6.1: Run-time and store claimed for the *deepest* programs

Program	Straightforward	Transformed
Run-time in seconds	16.9	12.2
(ratio)	(1)	(0.72)
Store claimed in cells	317477	330787
(ratio)	(1)	(1.04)

The effect of the transformation is clearly demonstrated; about 30% improvement on running time with a little increase in allocated cells.

Although our transformation technique has been applied to small programs in this chapter, there is no problems encountered on applying it to practical programs if we establish the transformation rules for the data structure that the program deals with. Examples of such rules have been shown in Sections 9.4 and 9.5. The transformation rules for various kind of data structures lead to the possibility of an automatic transformation system for optimizing functional programs. Transformed programs have to be evaluated in a lazy way; the argument expression is never evaluated until required, and when it is evaluated the argument is replaced by its result. Hence a partially parametrized function shared by several positions in an expression effectively distributes information obtained by the first visit to every data item. In fact it is memo-ized as a function closure which is bound to the shared variable corresponding to the argument.

It should be noted that our technique imposes no restriction on the forms of functional composition; Bird [Bird84] deals with the form $f\ x\ (g\ x)$. As shown by the examples, our rules do not depend on the form of functional composition, but do only on the traversal function. As for the language issue, it is assumed that our functional language allows the higher order function and its partial parametrization in addition to lazy evaluation and local definition mechanisms assumed by Bird. In fact these features,

including partial parametrization of higher order functions, are particularly powerful and useful tools in functional programming.

Chapter 10

Checking types in functional specifications

It is a desirable feature of a programming language that there should be some simple discipline of types. It ensures that only valid function applications may occur in any expression. The user of traditional typed languages is required to declare the type of variables and procedures. For a fragment of a Pascal program

```
var x: integer; y: char;
...
begin ··· x + y ··· end
```

the compiler can detect an error 'illegal type' of operands of the operator + which expects integers or reals using the information on variable types declared. A more flexible type system is adopted in several functional languages. Its basic idea is to deduce the type of a construct from the types of its constituents. Given is an expression $x+y$ and the type of the operator +:**int**×**int**→**int**, the type of that expression is deduced as **int** and both operands x and y must be of type **int**. Milner proposed such a type system [Milner78] and successfully used in ML which is the metalanguage of the LCF proof system [Gordon79b]. The Milner type discipline also permits *polymorphic* functions which is similar to the concept of *generic* procedures in traditional languages. The polymorphic type system marks a step forward in programming language design. It enables the compiler to deal with polymorphic functions without loss of error-detecting abilities. If it were not used, polymorphism is supported only by typechecking at run time.

In this chapter, we present an application of the polymorphic type system in the field of semantics description of programming languages. In describing denotational semantics of programming languages, injection operations into sum domains are conventionally omitted for the sake of brevity. This in turn leads to difficulties for semantic processing systems which accept denotational specifications as input and mechanically calculate them for debugging the semantics. We deal with an algorithm for inserting injection operations to denotational specifications as part of the typechecking process.

10.1. Insertion of injection operations

The *Semantics Implementation System* (SIS) of Peter Mosses [Mosses79] can be considered as the first system which generates a compiler or an interpreter from syntactic and semantic specifications written

in the style of denotational semantics. Experience with SIS led us to designing an experimental system [Ohira84] which includes a typechecker for which SIS lacks, and a translator to convert denotational description into functional programs. Since it had no interface to parser generator, only a few example specifications and programs were processed. More recently, a *Semantic Prototyping System* (SPS) of Mitchell Wand has been built [Wand84]. Most of the inefficiencies of SIS pointed out in [Bodwin82] have been improved in SPS using the tools such as Yacc and Scheme 84 available in UNIX operating systems[1]. Wand has also implemented a typechecker and insists on its importance from his experience with sizable examples. He makes an interesting remark that most common error detected by the typechecker is failure to inject operands into corresponding sum domains. The principal cause underlying such failure is to be sought in the fact that we usually take a value both as an element of a sum domain and as one of its summand domain when we write down denotational specifications. This kind of convention is widely adopted in the literature on denotational semantics [Gordon79a, Stoy77] for avoiding a tedious task of balancing types of operands, and for making the specification more readable. As mentioned in [Gordon79a], such convention might be taken as a conversion analogous to the coercion operation of programming languages. The necessary operations for retaining the type consistency could be inserted in such a way that conversion operations between integers and reals are generated by compilers. This would serve for a concise notation to be accepted by the semantic processing system.

We deal with denotational specifications written in a tentative semantics description language.

10.2. A semantics description language

Major components of SIS are a parser generator and an evaluator. The parser generator accepts concrete syntax of the language and generates a parser to analyze the program written in that language into an abstract syntax tree. The abstract syntax tree corresponds to an element of the syntactic domain which is represented by concrete data structure manipulated by the evaluator. Semantics of the language is written using *Denotational Semantics Language* (DSL), which is an extension of lambda notation. The DSL description is then converted into a simpler form to be evaluated when the parse tree is given.

---

[1] Developed and Licensed by AT&T.

The SPS of Wand consists of similar components. In addition, included is a typechecker for guaranteeing type consistency. Yacc and Scheme 84 take the part of the parser generator and the evaluator, respectively. Semantics is described using Scheme 84 functions. According to [Wand84], programs compiled by SPS run much faster than those by SIS. Wand concludes that the efficient interpreter provided by Scheme 84 means a great deal. And the Yacc parser generator adopted by SPS seems to do much to increase the efficiency of syntactic processing.

Our *Sementics Description Language* (SDL) is independent of the parser generator, while the interface to it should be assumed. We expect to use the state-of-the-art software tools such as Yacc and Lex. We also assume that the SDL description can be directly evaluated, or translated into certain functional language for execution. In our previous work [Ohira84], we chose ML [Gordon79b, Chujo84] as an implementation language of the evaluator. Typechecking performed by ML becomes redundant if the consistency of the SDL specification is checked separately. We could write a front-end translator of SDL for the portable functional system described in Chapter 8 of this thesis. In this case the front-end should perform typechecking.

SDL consists of the facilities to define domains and functions; no syntactic definitions are written in SDL. We refrain from giving a complete definition of SDL in this thesis. Instead, we will informally introduce a small set of primitives. And expected facilities to be implemented by the evaluator will be mentioned where appropriate.

10.2.1. Domains

In the semantics processing system, every domain has to be implemented in some way; that is, every element of defined domains should be represented as a *value* to be calculated by the evaluator. In other words, each domain should be represented by a concrete data type of the language with which the evaluator is to be implemented. We do not deal with *polymorphic values* in our evaluator, while do with *polymorphic functions* in describing semantic functions. Hence we make the assumption that every value in denotational description lies in certain data type of the implementation language, and a correspondence between them exists.

In this section, we use the term *type* to refer to the domain when its representation is in mind.

The way of defining domains in SDL follows standard textbook. We leave the details to Section 3.3 of [Gordon79a].

Let $D, D_1, D_2, \cdots$ stand for arbitrary domains, and *Int*, *Bool*,*?* for primitive domains.

(D1) Primitive Domains

(D1.1) Standard domains: *Int* of integer values, and *Bool* of Boolean values.

(D1.2) Singleton domains: *?* of a distinguished element **?**, and any domain with a single element represented by a symbol.

(D1.3) Abstract domains: Domains of which structures and values are not specified in SDL but are to be provided by the evaluator.

(D2) Function Domains

$D_1 \rightarrow D_2$ of functions with the source D_1 and the target D_2.

(D3) Product Domains

$D_1 \times D_2 \times \cdots \times D_n$ of n-tuples of elements from $D_1, D_2, \cdots, D_n$.

(D4) Sequence Domains

D^* of finite sequences of elements from D.

(D5) Sum Domains

$t_1[D_1]+t_2[D_2]+\cdots+t_n[D_n]$ of union of $D_1, D_2, \cdots, D_n$ with tags $t_1, t_2, \cdots, t_n$.

Domain equations are used to define recursive domains:

$$D_1 = G_1[D_1, \cdots, D_m]$$
$$\cdots$$
$$D_m = G_m[D_1, \cdots, D_m]$$

where each G_i is a domain expression constructed from $D_1, \cdots, D_m$ and primitive domains using the domain constructors $\rightarrow$, $\times$, $*$, and + described above.

Although it would be unnecessary to explain each of these in detail, several points should be noted.

The domain *?* of (D1.2) is intended to be one consisting of a single element **?** representing the *semantically nonsensical* value. In our typechecking algorithm, failure in matching the type of an expression with required type results in assigning **?** to that expression (See the next section). Other singleton domains will be used to define finite domains in combination with operation + of (D5). For example, we can define a finite domain

$$Finalstate = abort\,[Error\,] + normal\,[Stop\,]$$

where *Error* and *Stop* are singleton domains of which values are *error* and *stop*, respectively.

Implementation of abstract domains (D1.3) remains open in the sense of the abstract type in programming languages. This is similar to *holes* of the type system of SPS. We require for these domains only that all the necessary functions and their types are to be given in the part of expression definitions. For example, the domain *Ide* of identifiers commonly used in specifications of programming languages might be treated as an abstract domain; we are not interested in how the element of *Ide* is represented. If we use a function *equal_ide* to check the equality of two identifiers, all that we need in typechecking is the type of that function, i.e., $(Ide \times Ide) \rightarrow Bool$.

Rule (D5) for construction of the sum domain differs slightly from that of [Gordon79a]. Every summand of a sum domain must have a unique tag which is used to discriminate among summands and to inject the summand into the sum domain. The usage of the tag will be explained in the next section.

As mentioned earlier, we make the assumption that the element of the syntactic domain is to be represented by an abstract syntax tree generated by the parser. DSL of SIS has a particular construction rule for syntactic domains. In contrast to this, our syntactic domains are constructed using general rules for sums and products of syntactic domains themselves. For example, a syntactic domain *Cmd* of commands

$$cmd \;::=\; cmd\;;\,cmd \;\mid\; ide := exp$$

can be defined as

$$Cmd = cmd\_seq\,[Cmd \times Cmd\,] + cmd\_asg\,[Ide \times Exp]$$

where *Ide* and *Exp* are syntactic domains for identifiers and expressions[2]. We do not distinguish between

[2] It may be claimed that the use of terminal symbols ought to be allowed in SDL to write *Cmd* = [*Cmd* ";" *Cmd*] + [*Ide* ":=" *Exp*] as an abbreviation to *Cmd* = "Cmd;Cmd"[*Cmd*×*Cmd*] + "Ide:=Exp"[*Ide*×*Exp*]. Improvements should be made on SDL for the user to write the specification more comprehensible, though it is not our main concern here.

syntactic and semantic domains in domain construction.

10.2.2. Expressions

Expressions used in defining semantic functions are usually written in a particular version of lambda notation. An expression in SDL is either

(E1) a constant of type integer or Boolean,

(E2) a variable,

(E3) a combination, or an applicative expression of the form

$$f\ e_1 \cdots e_n$$

where $n \geq 1$, f is a variable, and $e_1, \cdots, e_n$ are expressions. Note that f must be a variable; expression of other form is not allowed. (This is motivated later in the next section).

(E4) a lambda expression of the form

$$\lambda v.e,$$

or

(E5) a case expression of the form

$$\textbf{case}\ e_0\ \{\ t_1[v_1].e_1\ |\ \cdots\ |\ t_n[v_n].e_n\ \}$$

where $e_0, e_1, \cdots, e_n$ are expressions, $t_1, \cdots, t_n$ are tags of a sum domain, and $v_1, \cdots, v_n$ are bindings.

The binding in (E4) and (E5) is an extension to lambda notation. It is either

(B1) a variable x,

or

(B2) a structured binding of the form

$$\gamma\, v_1 \cdots v_n$$

where $n \geq 1$, γ is a constructor, and $v_1, \cdots, v_n$ are bindings.

The constructor in bindings may be any curried function which creates a structured data of a particular type from its components. Examples are

$prefix : \alpha \rightarrow \alpha^* \rightarrow \alpha^*$

$pair : \alpha \rightarrow \beta \rightarrow \alpha \times \beta.$

Constructors *prefix* and *pair* are *polymorphic* in the sense that they are applicable to values of any types α and β. We will shortly discuss about polymorphism in our typechecking algorithm.

The case expression is another extension to conventional lambda notation. It tests a value of a sum domain and binds it to variables in the binding of corresponding summand. To gain a better understanding of the usage of the case expression, consider the primitive operations over sum domains in [Gordon79a]. For a sum domain

$$D = t_1[D_1] + \cdots + t_n[D_n],$$

there are primitive functions

$$isD_i\ d = \begin{cases} \textbf{true} & \text{if } d \text{ comes from summand } D_i \\ \textbf{false} & \text{otherwise} \end{cases} \qquad \text{(Test)}$$

$$outD_i\ d = \begin{cases} d_i \text{ in } D_i & \text{if } (isD_i\ d) \text{ holds} \\ ? & \text{otherwise} \end{cases} \qquad \text{(Projection)}$$

and

$$inD_i\ d = d \text{ in } D \qquad \text{(Injection)}$$

Note that these functions have types

$isD_i : D \rightarrow Bool$

$outD_i : D \rightarrow D_i$

$inD_i : D_i \rightarrow D.$

As mentioned in the previous section, we use tag t_i to inject values of D_i into D; that is, $(inD_i\ d)$ is written as $(t_i\ d)$ in SDL. The test isD_i and the projection $outD_i$ can be written using the case notation

case d { $t_1[v_1]$.**false** | $\cdots$ | $t_i[v_i]$.**true** | $\cdots$ }

and

case d { $t_1[v_1]$.? | $\cdots$ | $t_i[v_i].v_i$ | $\cdots$ },

respectively. As will be shown in following examples, the case notation is more useful than primitive operations like tests and projections.

As far as typechecking is concerned, special forms like infix notation for arithmetic operations are not relevant and are excluded from SDL. The conditional expression

if e **then** e_1 **else** e_2

could be defined by a function

$$if : Bool \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

or

$$if : (\alpha \times \alpha) \rightarrow Bool \rightarrow \alpha$$

as appropriate. Polymorphic functions appear again. In fact polymorphism eliminates the need to consider special functions one by one, and makes our typechecking algorithm versatile and independent of the implementation language of the evaluator. Polymorphic values are, however, not included in SDL domains. That is, functions like *prefix* and *pair* can be used to describe semantics such a way that they appear at the function position of combinations and the constructor position of structured bindings. We can say that polymorphic functions are second class objects in the sense that they are not allowed to become arguments of functions.

We now extend the domain expression by adding

(D0) Type variables $\alpha, \beta, \cdots$

for specifying types of polymorphic functions.

The expression definition is a system of recursive equations

$$E_1 : T_1 = F_1[E_1, \cdots, E_n]$$
$$\cdots$$
$$E_n : T_n = F_n[E_1, \cdots, E_n]$$

where each E_i is a variable, T_i is a domain expression constructed using (D0)-(D4), and each F_i is an expression built from (E1)-(E5). Note that the domain construction rule (D5) is not allowed here. This effectively eliminates the possibility of defining sum domains of which summands contain type variables, i.e., polymorphic sum domains.

Expression $F_i[E_1, \cdots, E_n]$ may be omitted; in this case $E_i : T_i$ simply states that E_i is of type T_i. Otherwise, E_i is defined by the right hand side of the equation. The type may be polymorphic in the first case, but must not be polymorphic in the second case.

10.3. Types and typechecking

In order to ensure consistent treatment of the type, we follow a clear exposition of polymorphic typechecking scheme by Cardelli [Cardelli84a]. We start with discussion of types with relation to domains described in the previous section.

10.3.1. Types

Correspondence between domains and types is rather straightforward. Types are *structures* given by domain definitions. Given a domain equation

$$D_i = G_i[D_1, \cdots, D_m]$$

the type specified by D_i is the structure defined by $G_i[D_1, \cdots, D_m]$. That is, we are concerned with the structure of the domain.

A type can be either a type variable or a type operator. Type variable stands for an arbitrary type. Type operator corresponds to one of the primitive domains or the domain constructors. An operator standing for primitive domains like *Int*, *Bool* or *?*, or an abstract domain is nullary. Parametric operators like $\rightarrow$, $\times$, *, and + take one or more types as arguments. Types containing type variables are *polymorphic* and called *polytypes*. Other types are *monomorphic* and called *monotypes*. Recall that types corresponding to domain expressions in the domain definition are monomorphic. In SDL, only a function can be polymorphic.

Thus, a type is either

(T1) an atomic type,

(T1.1) a type variable,

(T1.2) a primitive type operator among **int**, **bool**, **?**, $\cdots$, corresponding to a primitive domain,

(T2) $T_1 \rightarrow T_2$, where T_1 and T_2 are types,

(T3) $T_1 \times T_2 \times \cdots \times T_n$, where $T_1, T_2, \cdots, T_n$ are types,

(T4) T^*, where T is a type,

or

(T5) $t_1[T_1]+t_2[T_2]+\cdots+t_n[T_n]$, where $t_1, t_2, \cdots, t_n$ are tags, and $T_1, T_2, \cdots, T_n$ are monotypes.

As mentioned earlier, constituent types of a sum type must be monomorphic.

The correspondence between domains and types is straightforward. For example, for a set of domain equations

$$D_0 = t_1[D_1] + t_2[D_2]$$
$$D_1 = Int \times D_0$$
$$D_2 = Bool$$

a type σ corresponding to D_0 is: $\sigma = t_1[\mathbf{int}\times\sigma] + t_2[\mathbf{bool}]$ where **int** and **bool** are primitive types corresponding to the primitive domains *Int* and *Bool*.

10.3.2. Typechecking and injection operations

Typechecking is a process of checking whether every term, or subexpression, of an expression has a type consistent with ones of other terms. In particular, we are concerned with the consistency of types of combinations. Let f be of type $\sigma_1\rightarrow\sigma$, and e of type σ'_1. Then what condition should be satisfied for the combination $(f\ e)$ being meaningful ? A sufficient condition would be $\sigma_1=\sigma'_1$, getting the type σ for $(f\ e)$. In another case where σ_1 is a sum type and σ'_1 is its summand with tag t_1, we can transform the term into acceptable one $(f\ (t_1\ e))$ using injection operator $t_1:\sigma'_1\rightarrow\sigma_1$. We will make a generalization of this idea in our typechecking algorithm.

In SDL, and in many textbooks, the type of the expression is given in its definition as described in the previous section. Although naming convention such as e for a variable of type *Exp* might be applied, there is no declaration of types for locally declared variables. This is very contrast to conventional typed language like Pascal. Typechecking in SDL is, therefore, the process of checking whether every $F_i[E_1, \cdots, E_n]$ does or does not have a type consistent with T_i under the condition that each E_j has type T_j for $j=1, \cdots, n$. Note that T_i should be monomorphic in the definition of the form

$$E_i : T_i = F_1[E_1, \cdots, E_n]$$

No types for local variables are specified in SDL[3].

---

[3] The accepted usage of global declarations, e.g., e :*Exp* for variable e and decorated variables e_1, e', etc. being of type *Exp*, may help typechecking. We do not adopt such declarations in SDL because the types of local variables can be inferred from the context as shown in the algorithm. However, there remains the possibility of using such global

Our typechecker does not only check the consistency of types, but also inserts necessary injection operations to SDL specifications. Let **P** be the typechecking procedure producing SDL specifications with injection operations inserted for any SDL specifications. Then, for any x, $x'=\mathbf{P}(x)$ is also an SDL specification and $x'=\mathbf{P}(x')$ is expected. That is, **P** has the *idempotent* property $\mathbf{P}^2=\mathbf{P}$.

10.3.3. Typechecking algorithm

Our typechecking algorithm partly relies on the polymorphic type system of ML [Gordon79b, Milner78]. It is different, however, in that ours determines types of constituents of an expression from the type of that expression, while the type of an ML expression is inferred from known types of its constituents. That is, types are propagated downwards from an expression to its constituent subexpressions in our algorithm. This enables us to insert injection operations into subexpressions properly to keep the type of the larger expression unaffected.

We now introduce some notations for describing the algorithm.

We use the notation

$$[\sigma' \rightarrow \sigma]$$

to represent an injection operation. For monotypes σ' and σ other than *?*, $[\sigma' \rightarrow \sigma]$ is the function that injects every element of type σ' into one of type σ. In particular, it is simply the identity function if σ' and σ are identical. If σ' or σ is polymorphic, or either of them is *?*, it stands for the function $\lambda x.?$ which maps any value into the nonsensical element **?** of type *?* .

The *environment* ρ *for variables* associates variables with their types. The initial environment ρ_0 for the definitions

$$E_i : T_i = F_1[E_1, \cdots, E_n] \quad \text{for } i=1,2,\cdots,n$$

is

$$\rho_0 = \{ E_1 \rightarrow T_1; \cdots ; E_n \rightarrow T_n \}$$

that represents the function

$$\rho_0 x = \begin{cases} T_i & \text{if } x=E_i \\ \textit{undefined} & \text{otherwise} \end{cases}$$

---

declarations to locate erroneous specifications of types.

An environment ρ is updated by $\{x \rightarrow \sigma\}$ to yield a new environment $\rho+\{x \rightarrow \sigma\}$:

$$(\rho+\{x \rightarrow \sigma\})\, y = \begin{cases} \sigma & \text{if } x=y \\ \rho\, y & \text{otherwise} \end{cases}$$

Similarly, the *environment π for type variables* maps type variables to monotypes. Substitution τ/π of type variables in a type expression τ with monotypes under the type environment π is defined as

(1) For an atomic type

(1.1) a type variable α, $\alpha/\pi = \pi\alpha$

(1.2) a primitive type operator o, $o/\pi = o$

(2) For a functional type $\tau_1 \rightarrow \tau_2$, $(\tau_1 \rightarrow \tau_2)/\pi = (\tau_1/\pi) \rightarrow (\tau_2/\pi)$

(3) For a product type $\tau_1 \times \tau_2 \times \cdots \times \tau_n$,

$$(\tau_1 \times \tau_2 \times \cdots \times \tau_n)/\pi = (\tau_1/\pi) \times (\tau_2/\pi) \times \cdots \times (\tau_n/\pi)$$

(4) For a sequence type τ^*, $(\tau^*)/\pi = (\tau/\pi)^*$

(5) For a sum type σ, which should be a monotype, $\sigma/\pi = \sigma$.

Finally, we denote the expression obtained by inserting injection operations to the expression e in the context requiring type σ under the environment ρ by

$$<e : \sigma \mid \rho>$$

where σ is a monotype.

The basic algorithm **P** can be described as follows. We use σ, σ', σ'', σ_1, $\cdots$ to represent monotypes, and τ, τ_1, $\cdots$ to represent polytypes (including monotypes).

(A1) If a constant i of type **int** appears in the context requiring type σ, injection function ψ:**int**$\rightarrow\sigma$, if any, is inserted to obtain $(\psi\, i)$ of type σ. That is,

$$<i : \sigma \mid \rho> = [\mathbf{int} \rightarrow \sigma]\, i$$

Similarly for a constant b of type **bool**,

$$<b : \sigma \mid \rho> = [\mathbf{bool} \rightarrow \sigma]\, b$$

The type *?* is considered to be universal; it is consistent with any type.

$$<? : \sigma \mid \rho> = ?$$

(A2) If a variable x of type σ' appears in the context requiring type σ, injection function $\psi:\sigma'\rightarrow\sigma$, if any, is inserted to obtain $(\psi\, x)$ of type σ. Types of variables are assigned by expression definitions or bindings in expressions of the form (E4) or (E5). The type of each variable is maintained as an environment ρ.

$$<x : \sigma \mid \rho> = [\rho x \rightarrow \sigma]x$$

(A3) For a combination $(f\ e_1 \cdots e_n)$ in the context requiring type σ, assume that variable f is assigned a functional type $\rho f = \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$, which can be polymorphic with type variables $\alpha_1, \cdots, \alpha_m$. Find a type environment π for type variables α_j by unifying τ with σ or its summands. Then replace type variables α_j in $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ by monotypes using the type environment π just obtained to yield a monotype $\sigma'_1 \rightarrow \cdots \rightarrow \sigma'_n \rightarrow \sigma'$. That is, $\sigma' = \tau/\pi$ and $\sigma'_i = \tau_i/\pi$ for $i = 1, \cdots, n$. Finally make a new combination

$$<(f\ e_1 \cdots e_n) : \sigma \mid \rho> = [\sigma' \rightarrow \sigma](f\ e'_1 \cdots e'_n)$$

where each e'_i is a transformed expression of e_i under the context of required type σ'_i, i.e., $e'_i = <e_i : \sigma'_i \mid \rho>$.

If f does not have a functional type of the above form,

$$<(f\ e_1 \cdots e_n) : \sigma \mid \rho> = ?$$

(A4) For a lambda expression $\lambda v.e$ in the context requiring type σ, find a functional type $\sigma' \rightarrow \sigma''$ from σ itself or its summands with injection function $\psi:(\sigma' \rightarrow \sigma'') \rightarrow \sigma$. Update the environment ρ to reflect the lambda variables in binding v of type σ' as $\rho + \{\sigma/v\}$ (See below). Then check and transform e in the context of required type σ'' with the new environment to obtain e'. Finally make a combination of ψ and a new lambda term $\lambda v.e'$ getting $\psi(\lambda v.e')$. That is,

$$<\lambda v.e : \sigma \mid \rho> = [(\sigma' \rightarrow \sigma'') \rightarrow \sigma](\lambda v.<e : \sigma'' \mid \rho + \{\sigma'/v\}>)$$

If no functional type is found from σ or its summands,

$$<\lambda v.e : \sigma \mid \rho> = ?$$

(A5) For a case expression

$$e = \textbf{case}\ e_0\ \{t_1[v_1].e_1 \mid \cdots \mid t_n[v_n].e_n\}$$

in the context requiring type σ, find a functional type $\sigma' \rightarrow \sigma''$ from σ itself or its summands with

injection function $\psi:(\sigma'\rightarrow\sigma'')\rightarrow\sigma$. Assume that σ' is a sum type

$$\sigma' = t_1[\sigma_1]+ \cdots +t_n[\sigma_n] \ .$$

Transform e_0 into e'_0 of type σ':

$$e'_0 = <e_0 : \sigma' \mid \rho>$$

For each e_i, find the new environment $\rho+\{\sigma_i/v_i\}$ for variables which reflects binding v_i of type σ_i, and transform e_i with respect to type σ'' to obtain e'_i under that environment:

$$e_i = <e_i : \sigma'' \mid \rho+\{\sigma_i/v_i\}> \quad \text{for } i=1,2,\cdots,n$$

Then make a new expression

$$<e : \sigma \mid \rho> = [(\sigma'\rightarrow\sigma'')\rightarrow\sigma](\textbf{case } e'_0 \{t_1[v_1].e'_1 \mid \cdots \mid t_n[v_n].e'_n\})$$

of type σ.

If no functional type is found from σ or its summands, or if the source type σ' of the functional type is not a sum type,

$$<e : \sigma \mid \rho> = ?$$

The injection function may well be the identity function. The typechecking procedure **P** will report type inconsistencies to make the term be reduced to **?** if the conditions mentioned in each case are not satisfied. It should be noted that polymorphic types are allowed only for the function f in (A3).

In (A4) and (A5), the environment ρ of variables needs to be updated. Bindings are either a variable or a composite variable structure.

(AB1) If the binding is a variable x and the type given to it is σ, then the new environment is one updated as x has type σ. That is,

$$\rho+\{\sigma/x\} = \rho+\{x\rightarrow\sigma\}$$

(AB2) If the binding is of the form

$$\gamma\, v_1 \cdots v_n$$

and the given type is σ, assume that the constructor γ has type $\tau_1\rightarrow\cdots\rightarrow\tau_n\rightarrow\tau$, which can be polymorphic with type variables $\alpha_1, \cdots, \alpha_m$. Note that σ must be monomorphic. Then find type environment π for type variables α_j by unifying τ with σ. Finally replace type variables α_j in $\tau_1\rightarrow\cdots\rightarrow\tau_n\rightarrow\tau$ by monotypes using the type environment π to obtain $\sigma'_1\rightarrow\cdots\rightarrow\sigma'_n\rightarrow\sigma$.

That is, $\sigma=\tau/\pi$, and $\sigma'_i=\tau_i/\pi$ for $i=1,\cdots,n$. The new environment is obtained by applying this algorithm recursively:

$$\rho+\{\sigma/(\gamma v_1 \cdots v_n)\} = \rho+\{\sigma'_1/v_1\}+\cdots+\{\sigma'_n/v_n\}$$

If the constructor γ does not have the type as above,

$$\rho+\{\sigma/(\gamma v_1 \cdots v_n)\} = \rho+\{v_1\rightarrow ?\}+\cdots+\{v_n\rightarrow ?\}$$

The *unification algorithm* [Robinson65] is used in stages (A3) and (AB2) to find particular instances of polymorphic types.

To illustrate how the types of subexpressions are determined, consider a simple term (*pair x y*) in the context of required type $Val=val_1$[**int**]$+val_2[Num\times$**bool**$]$. Assume that $Num=num_1$[**int**]$+num_2[?]$, and the environment for variables gives types **int** and **bool** to x and y, respectively, i.e., $\rho=\{x\rightarrow$**int**; $y\rightarrow$**bool**$\}$. To begin with, Rule (A3) is applied to (*pair x y*). The type of *pair* is polymorphic, i.e., $\alpha\rightarrow\beta\rightarrow\alpha\times\beta$. By unifying $\alpha\times\beta$ with the second summand of *Val*, a type environment $\pi=\{\alpha\rightarrow Num; \beta\rightarrow$**bool**$\}$ is obtained. And $\sigma' = (\alpha\times\beta)/\pi = Num\times$**bool**. Then the injection operation for that term is $[\sigma'\rightarrow Val]=val_2$. The next step is to check types of subexpressions x and y, which must be transformed to have type *Num* and **bool**, respectively. We can apply Rule (A2) to both terms. From the environment ρ, $\rho x =$ **int**, which is injected into *Num* by num_1, i.e., $[\rho x\rightarrow$**int**$] = num_1$. And y remains as it is. Thus, we obtain a term

$$<(pair\ x\ y) : Val \mid \rho> = (val_2(pair(num_1\ x)\ y))$$

of type *Val*.

A small but complete example of typechecking a specification in Figure 10.3.1 is shown in the Appendix C of this thesis. The specification is written in the form of S-expression of Lisp, which should be considered as an internal form of SDL. A more sophisticated style of specifications will be allowed in the final definition of SDL.

10.4. Remarks

We have not specified how to choose a summand from possible candidates in the clauses "by unifying τ with σ or its summands" in (A3), and "from σ itself or its summands" in (A4) and (A5). The most

Abstract syntax

i	integer notation
b	truth value notation
cmd∈ **Cmd**	commands
exp∈ **Exp**	expressions

cmd ::= cmd ; cmd | **output** exp
exp ::= *i* | *b*

Semantic domains

Int	integers
Bool	truth values
State	states
Cc = State→Ans	commamd continuations
Ec = Val→Cc	expression continuations
Ans = {*error*}+**Val**×**Ans**	answers
Val = Int+Bool	values

Semantic functions

C : **Cmd→Cc→Cc**
E : **Exp→Ec→Cc**

C[cmd1;cmd2] c = *C*[cmd1](*C*[cmd2]c)
C[**output** exp] c = *E*[exp](λvs.(v, c s))
E[*i*] k = k *i*
E[*b*] k = k *b*

Figure 10.3.1. Example language definition

general solution would be one to try *all* the possible cases using backtracking. However, it would be impractical for sizable specifications. Since the main purpose of our algorithm is still guaranteeing the type consistency of specifications, no backtracking is implemented in our typechecker; actually only the first possible summand is taken. The validity of inserting injection operations could be confirmed to some degree by applying the procedure **P** to the resulting specification again to check whether the idempotent property is satisfied. Of course, it does not follow that the transformation is correct even if that property holds. In either case, the resulting specification has to be inspected whether it is or is not the intended one. If not, the necessary injection operation should be inserted to make the specification unambiguous.

To find out the cause of ambiguity, the dependency graph for domains would be helpful. The dependency graph consists of nodes containing type operators, and of arcs directed to constituent types. Each node is associated with a type of which structure is given by the maximal subgraph containing that node. The domains defined in the specification correspond to types associated with the dependency graph. Figure

10.4.1 illustrates the graph for domains defined in Figure 10.3.1. Summands with the identical structure do not exist in this case. If several descendants of a + node have the same kind of operators among $\rightarrow$, $\times$, $*$, or primitive types, care should be taken to make the specification unambiguous.

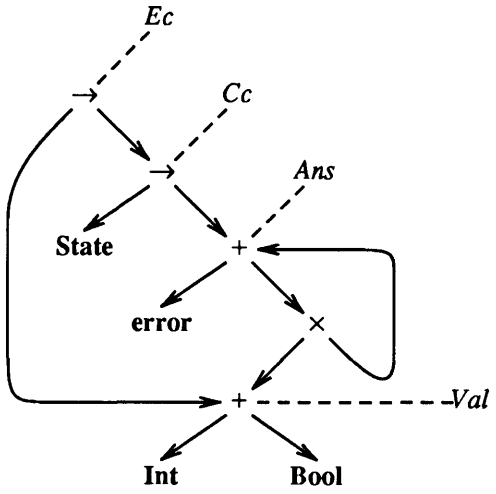


Figure 10.4.1. Dependency graph for domains

Although there is a limitation of our typechecker, injection insertion turns out to be useful for practical problems. In fact, there is little chance of failure in injection by the above procedure.

We have devised an algorithm for inserting injection operations to denotational specifications. Although local binding mechanisms such as "**let** $\cdots$ **in** $\cdots$", or " $\cdots$ **where** $\cdots$" are not included in SDL described in this chapter, extension of our algorithm to such expressions is straightforward.

In a sense, our algorithm is based on a compromise reached by restricting the class of acceptable expressions to ones described in Section 10.1 at the cost of generality. Although the rule (E3) seems too restrictive at first sight, it turns to be no practical problems. If more general form of combination (f e) had been allowed, the types of f and e could not be deduced from the type of (f e) alone; a functional type should be produced for $f=\lambda v.e'$ from little knowledge of the type of f. This can be done if we simply check the type consistency as in ML, but it is not the case for our purpose. From this observation, we have chosen a slightly restricted class of expressions for our specification language.

Mitchell [Mitchell83] deals with polymorphic typechecking with automatic coersions between types.

Allowable coercion there should be of the form " α is coercible to β" where α and β are *atomic* types. This is too restrictive for our purpose.

As the final remark, we have to state about the limitation of our algorithm. Although the value for practical usage has been demonstrated in this chapter, we do not claim that our algorithm is complete or optimum. On the contrary, it fails easily for artificial problems as described in the previous section.

It is hoped, however, that this research has made a step toward a complete design of SDL and a more sophisticated semantics implementation system based on SDL. Another area of research is in extending this work to coercion insertion in functional languages. The applicability of our algorithm and the relations to other approaches, e.g.,[Futatsugi84], should be studied further.

Chapter 11

Conclusions

11.1. Contributions of the thesis

The major contributions of the thesis are as follows:

- *New transformation techniques for functional specifications.* It is a great advantage of functional programming that it allows us to specify problems in a highly abstract way. In practice, however, the specification must be transformed into a form suitable for execution. Program transformation in this context means source-to-source conversion of programs. The technique described in Chapter 9 is based on higher order functions and partial parametrization, both of which are particularly powerful and useful tools in functional programming. Unlike other transformation work, our transformation is *algorithmic* rather than *heuristic*. The idea of *lambda-hoisting* in Chapter 6 has been successfully used for that purpose.
- *Fully lazy evaluators.* Among many ideas on evaluation mechanisms of functional programs, fully lazy evaluation turns out to be optimal in the sense that *every* expression is evaluated *at most once* after the variables in it have been bound. It would be intuitively clear that full laziness is better than ordinary laziness at least in principle because only the arguments of functions are evaluated at most once in lazy evaluation. It is, however, necessary to provide an existence proof of efficient evaluators which perform fully lazy evaluation. The combinator reducers in Chapter 4 and the fully lazy functional machine in Chapter 7 are such evaluators.
- *Portable compilers.* We have demonstrated a portable compiler system in Chapter 8. Successful implementations on four different computers prove that our approach is correct to produce efficient code on different machines. The lambda-hoisting technique has been used for obtaining expressions of the fully lazy normal form in the course of compilation.
- *Typechecking with operator insertion. Polymorphic types* are particularly useful in checking types of functional specifications. The polymorphic type discipline allows us to specify *generic* functions of which argument can be of any type. It enables the compiler to deal with polymorphic functions

without loss of ditecting abilities of type inconsistencies. We have developed an extended typechecking algorithm in Chapter 10 that may insert type transfer functions like *coercion* in conventional languages during the typechecking process. Unfortunately, insertion of such operators cannot be uniquely determined in nature. This does, however, serve for using compact and rigorous notations in functional specification.

11.2. Future work

The work may raise some questions for future research.

- *Deciding the parameter order for optimal evaluation.* Since partial parametrization depends on the order of parameters, maximal free subexpressions may change according to that order. It would be desirable to discover an algorithm for determining an optimal order of parameters so that full laziness takes great effect. The use of algebraic laws such as commutativity and associativity of arithmetic operations should also be considered.
- *Automatic transformation systems.* An approach to transformation systems based on the technique described in Chapter 9 would be promising. If we develop some mechanisms for defining abstract data types and associated higher order functions, we can transform a wide class of programs into more efficient ones. As described above, the method is algorithmic and transformation is straightforward. What we need is the highly abstract definition of data types and traversal functions.
- *Designing higher level notations.* We have demonstrated in Section 8.3 that the Zermero-Fraenkel set notation can be transformed into primitive functional notation. The use of the ZF-notation in functional language is an excellent idea of Turner and we have borrowed his idea. Other convenient notations for specifying problems should be sought and implemented. A candidate for such notations is *functional arrays* similar to arrays of conventional languages. Of course, functional arrays do not permit destructive updating which is the main purpose of assignment of procedural programming. Although the array notation is not very high level, great many algorithms have been deviced using conventional arrays. They are expressed in a natural way using arrays and very comprehensible Some of them can be functional algorithms on functional arrays. An efficient method of implement-

ing functional arrays should be studied.

- *Concurrent execution.* Functional programs contain implicit parallelism which makes execution on parallel processors most attractive. The idea of full laziness does not cause any problem in parallel evaluation. We have not discussed parallelism in this thesis, there are many problems unsolved. Is parallel processing really effective? How can communication overhead be minimized? Further research should be expected.

References

[Aho77] Aho, A.V., and Ullman, J.D.: *Principles of Compiler Design.* Addison-Wesley, 1977.

[Augustsson84] Augustsson, L.: A compiler for lazy ML. *Proc. 1984 ACM Symp. LISP and Functional Programming*, 218-227 (1984)

[Backus78] Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* **21**, 613-641 (1978)

[Backus81] Backus, J.: Functional programs as mathematical objects. *Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architecture*, 1-10 (1981)

[Bird84] Bird, R. S.: Using circular programs to eliminate multiple traversals of data. *Acta Informatica 21*, 239-250 (1984)

[Bodwin82] Bodwin, J., Bradley, L., Kanda, K., Litle, D., and Pleban, U.: Experience with an experimental compiler generator based on denotational semantics. *Proc. 1982 ACM Symp. on Compiler Construction, SIGPLAN Notices Vol. 17, No. 6*, 216-229 (1982)

[Burge75] Burge, W. H.: *Recursive Programming Techniques.* Addison-Wesley, 1975

[Burstall69] Burstall, R.M.: Proving properties of programs by structural induction. *Computer Journal* **12**, 41-48 (1969)

[Burstall77] Burstall, R.M., and Darlington, J.: A transformation system for developing recursive programs. *J. ACM* **24**, 44-67 (1977)

[Burstall80] Burstall, R.M., MacQueen, D.B., and Sannella, D.T.: HOPE, an experimental applicative language. *Proc. of 1980 ACM Lisp Conference*, 136-143 (1980)

[Burton82] Burton, F.W.: A linear space translation of functional programs to Turner combinators. *Information Processing Letters* **14**, 201-204 (1982)

[Cardelli84a] Cardelli, L.: Basic polymorphic typechecking. *Polymorphism Vol. 2, No. 1*,1984.

[Cardelli84b] Cardelli, L.: Compiling a functional language. *Proc. 1984 ACM Symp. LISP and Functional Programming*, 208-217 (1984)

[Chujo84] Chujo, H., and Takeichi, M.: Porting ML on a New Machine (In Japanese). *Proc. 28-th Symp. of Inf. Proc. Japan*, 427-428 (1984)

[Church41] Church, A.: *The Calculi of Lambda-conversion.* Princeton University Press, 1941.

[Clarke80] Clarke, T.J.W., Gladstone, P.J.S., MacLean, C.D., and Norman, A.C.: SKIM - The S, K, I Reduction Machine. *Proc. of 1980 ACM Lisp Conference*, 128-135 (1980)

[Friedman76] Friedman, D.P., and Wise, D.S.: Cons should not evaluate its arguments. *Automata, Languages and Programming*, 257-284. Edinburgh University Press, 1976.

[Futatsugi84] Futatsugi, K., Goguen, J.A., Jouannaud, J.-P., and Meseguer, J.: Principles of OBJ2. *Proc. 12-th ACM Symp. on Principles of Prog. Lang.*, 52-66 (1984).

[Gordon79a] Gordon, M.J.C.: *The Denotational Description of Programming Languages.* Springer-Verlag, 1979

[Gordon79b] Gordon, M.J., Milner, R., and Wadsworth, C.P.: *Edinburgh LCF.* LNCS 78, Springer-Verlag, 1979

[Henderson76] Henderson, P., and Morris, J.M.: A lazy evaluator. *Proc. 3rd Symp. on Principles of Programming Languages*, 95-103 (1976)

[Henderson80] Henderson, P.: *Functional Programming: Application and Implementation.* Prentice-Hall, 1980.

[Henderson83] Henderson, P., Jones, G.A., and Jones, S.B.: *The Lispkit Manual.* Technical Monograph PRG-32, Oxford University Computing Laboratory, 1983.

[Hikita84] Hikita, T.: On the average size of Turner's translation to combinator programs. *Journal of Information Processing* **7**, 164-169 (1984)

[Hindley72] Hindley, J.R., Lercher, B., and Seldin, J.P.: *Introduction to Combinatory Logic.* Cambridge University Press, 1972.

[Hughes82] Hughes, R. J. M.: Super-combinators: a new implementation method for applicative languages. *Proc. 1982 ACM Symp. Lisp and Functional Programming*, 1-10 (1982)

[Hughes84] Hughes, R. J. M.: *The design and implementation of programming languages.* D.Phil. thesis. Oxford University, 1984.

[Hughes85] Hughes, R. J. M.: Lazy memo-functions. *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201**, 129-146, Springer-Verlag, 1985.

[Ida85] Ida, T., and Konagaya, A.: Comparison of closure reduction and combinatory reduction schemes. *RIMS Symposia on Software Science and Engineering II*, Lecture Notes in Computer Science **220**, 261-291, Springer-Verlag, 1985.

[Ingerman61] Ingerman, P.Z.: Thunks. *Comm. ACM* **4**, 55-58 (1961)

[Johnsson84] Johnsson, T.: Efficient compilation of lazy evaluation. *Proc. SIGPLAN '84 Symp. on Compiler Construction*, 58-69 (1984)

[Johnsson85] Johnsson, T: Lambda-lifting: Transforming programs to recursive equations. *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201**, 190-203, Springer-Verlag, 1985.

[Jones82] Jones, N. D., and Muchnick, S. S.: A fixed-program machine for combinator expression evaluation, *Proc. 1982 ACM Symp. Lisp and Functional Programming*, 11-20 (1982)

[Kennaway82] *The complexity of a translation of λ-calculus to combinators.* School of Computing Studies and Accountancy, Univ. of East Anglia, Norwich, 1982.

[Landin66] Landin, P.J.: The next 700 programming languages. *Comm. ACM*, **9**, 157-164 (1966)

[McCarthy62] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I.: *The LISP 1.5 Programmer's Manual.* MIT Press, 1962.

[Milner78] Milner R.: A theory of type polymorphism. *J. Comput. Syst. Sci. 17*, 348-375 (1978)

[Milner84] Milner, R.: A proposal for Standard ML. *Proc. 1984 ACM Symp. LISP and Functional Programming*, 184-197 (1984)

[Mitchell83] Mitchell, J.C.: Coercion and type inference (Summary). *Proc. 11-th ACM Symp. on Principles of Prog. Lang.*, 175-185 (1983)

[Morris80] Morris, J.H., Schmidt, E., and Wadler, P.L.: Experience with an applicative string processing language. *Proc. 7-th ACM Symp. on Principles of Programming Languages*, 32-46 (1980)

[Morris82] Morris, J.H.: Real programming in functional languages. *Functional Programming and its Applications* (eds. J. Darlington, P. Henderson, and D. A. Turner), Cambridge University Press, 1982.

[Mosses79] Mosses, P.: *SIS - Semantic Implementation System: Reference Manual and User Guide.* DIAMI MD-30, Dept. of Computer Science, University of Aarhus, Denmark, 1979

[Noshita85a] Noshita, K.: Translation of Turner combinators in $O(n \log n)$ space. *Information Processing Letters* **20**, 71-74 (1985)

[Noshita85b] Noshita, K., and Hikita, T.: The BC-chain method for representing combinators in linear space. *RIMS Symposia on Software Science and Engineering II*, Lecture Notes in Computer Science **220**, 291-306, Springer-Verlag, 1985.

[Ohira84] Ohira, T., and Takeichi, M.: A Language Development System (In Japanese). *Proc. 28-th Symp. of Inf. Proc. Japan*, 329-330 (1984)

[Peyton-Jones82] Peyton-Jones, S.L.: An investigation of the relative efficiencies of combinators and lambda expressions. *Proc. 1982 ACM Symp. LISP and Functional Programming*, 150-158 (1982)

[Peyton-Jones85] Peyton-Jones, S.L.: Yacc in Sasl - an exercise in functional programming. *Software-Practice and Experience* **15**, 807-820 (1985)

[Richards71] Richards, M.: The portability of the BCPL compiler. *Software - Practice and Experience* **1**, 135-146 (1971)

[Robinson65] Robinson, J.A.: A machine-oriented logic based on the resolution principle. *J. ACM* **12**, 23-49 (1965)

[Sammet69] Sammet, J.: *Programming Languages, History and Fundamentals.* Prentice-Hall, 1969.

[Stoy77] Stoy, J.E.: *Denotational Semantics*. MIT Press, 1977.

[Takeichi77] Takeichi, M: Pascal - implementation and experience. *Journal of Faculty of Eng., University of Tokyo (Series B)* **34**, 129-136 (1977)

[Takeichi82a] Takeichi, M.: Name identification for languages with explicit scope control. *Journal of Information Processing* **5**, 45-49 (1982)

[Takeichi82b] Takeichi, M.: Consistent annotations for scope rules. *Journal of Information Processing* **5**, 106-112 (1982)

[Takeichi84] Takeichi, M.: Evaluation of combinator expressions. *Proc. 1st JSSST Annual Conference*, 213-222 (1984). In Japanese.

[Takeichi85] Takeichi, M.: An Alternative Scheme for Evaluating Combinator Expressions. *Journal of Information Processing* **7**, 246-253 (1985)

[Takeichi86a] Takeichi, M.: A Functional Machine for Fully Lazy Evaluation (Extended Abstract). *Proc. RIKEN Symp. on Functional Programming*, 54-60 (1986)

[Takeichi86b] Takeichi, M: Lambda-hoisting: a transformation technique for fully lazy evaluation of functional programs (Extended abstract). *Proc. 3rd JSSST Annual Conference*, 113-116 (1986)

[Takeichi86c] Takeichi, M: Inserting injection operations to denotational specifications. *New Generation Computing* **4**, 365-381 (1986)

[Takeichi87] Takeichi, M.: Partial Parametrization Eliminates Multiple Traversals of Data Structures. To appear in *Acta Informatica*.

[Turner76] Turner, D. A.: *SASL language manual*, St. Andrew's University Technical Report No. CS/75/1, 1976.

[Turner79] Turner, D. A.: A New Implementation Technique for Applicative Languages, *Software - Practice and Experience*, 39-49 (1979)

[Turner81a] Turner, D. A.: *Aspects of the implementation of programming languages*. D.Phil. thesis. Oxford University, 1981.

[Turner81b] Turner, D.A.: The semantic elegance of applicative languages. *Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architecture*, 85-92 (1981)

[Turner82] Turner, D. A.: Recursion equations as a programming language. *Functional Programming and its Applications* (eds. J. Darlington, P. Henderson, and D. A. Turner), 1-28, Cambridge University Press, 1982.

[Turner84] Turner, D.A.: Functional programs as executable specifications. *Mathematical Logic and Programming Languages* (eds. C.A.R. Hoare and J.C. Shepherdson), 29-54, Prentice-Hall, 1984.

[Turner85] Turner, D. A.: Miranda: A non-strict functional language with polymorphic types. *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201**, 1-16, Springer-Verlag, 1985.

[Vuillemin74] Vuillemin, J: Correct and optimal implementations of recursion in a simple programming language. *J. Comp. Sys. Sci.* **9**, 332-354 (1974)

[Wadler85] Wadler, P.L.: *Listlessness is Better than Laziness.* Ph.D thesis. Carnegie-Mellon University, 1985.

[Wadsworth71] Wadsworth, C. P.: *Semantics and Pragmatics of the Lambda-Calculus.* D.Phil. thesis. Oxford University, 1971.

[Wand84] Wand, M.: A Semantic Prototyping System. *Proc. 1984 ACM Symp. on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6*, 213-221 (1984)

[Wijngaarden69] Wijngaarden V.(Ed.): Report on the algorithmic language ALGOL 68. *Numerische Mathematik* **14**, 79-218 (1969)

[Wirth73] Wirth, N.: *Systematic Programming/ An introduction.* Prentice-Hall, 1973.

Appendix A

An example of program translation

This appendix contains

primes.uc: Program *primes* written in *uc* (Appendix B).

primes.lk: Program in the common intermediate language. It has been obtained by translating *primes.uc* as described in Section 8.3 and eliminating compound bindings as in Section 8.2.

primes.flk: Program in the common intermediate language, which has been obtained by lambda-hoisting *primes.lk* as described in Chapter 6.

primes.s: Target code for the MC68000. It runs on the Sun-2 workstation using run-time routines provided as the FLFM library.

primes.uc

```
# prime numbers
primes
whererec {
        primes= sieve [2..]
and
        sieve (p:x)= p : sieve [n | n<-x; n%p !=0]
}
```

primes.lk

```
(letrec primes
     (primes sieve (from (quote 2)))
     (sieve lambda
          (@00004)
          (let (cons p
                    (sieve
                     (map (lambda (n)
                               n)

                          (filter (lambda (n)
                                        (neq (rem n p) (quote 0)))

                               x))))
               (p head @00004)
               (x tail @00004))))
```

primes.flk

```
(letrec g00004
    (g00005 lambda
        (@00004)
        (letrec (cons g00006
                    (g00005
                     (map (lambda (n)
                              n)

                          (filter (lambda (n)
                                      (neq (rem n g00006)
                                        (quote 0)))

                          g00007))))
                (g00007 tail @00004)
                (g00006 head @00004)))
    (g00004 g00005 (from (quote 2))))
```

primes.s

```
    .text
TAG =0xff000000
ADDR    =0x00ffffff
T_DATA  =0x80000000
T_CONS  =0x80000000
T_INT   =0xc0000000
T_BOOL  =0xa0000000
T_CHAR  =0x90000000
T_UNIT  =0x88000000
FALSE   =T_BOOL
TRUE    =FALSE+1
NIL =T_UNIT
    .globl _primes
    .globl __primes
    .data
_primes:    .long 0
    .long __primes
    .text
__primes:
L00004:
    jbsr  _CHECK
    jbsr  _DUMMY_ENV
    movl a5,d0
    movl a4,a5@+
    movl #L00005,a5@+
    movl d0,a2@-
    movl a5,d0
    movl a4,a5@+
    movl #L00007,a5@+
    movl d0,a2@-
    jsr   _INS_ENV
    jbsr  _FILL_ENV
    movl a4@(4),d0
    jbsr  _EVAL
    jra   _APPLY
L00005:
    jbsr  _CHECK
    movl a5,d0
    movl a4,a5@+
    movl #L00006,a5@+
    movl d0,a2@-
    movl a4@,a0
    movl a0@(4),d0
    jbsr  _EVAL
    jra   _APPLY
L00006:
    jbsr  _CHECK
    movl #T_INT+2,d0
    movl d0,a2@-
    movl #_from,a0
    jbsr  _GLOB
    jbsr  _EVAL
    jra   _APPLY
L00007:
    jbsr  _CHECK
    jsr   _EXT_ENV
    jbsr  _DUMMY_ENV
    movl a5,d0
    movl a4,a5@+
    movl #L00008,a5@+
    movl d0,a2@-
    movl a5,d0
    movl a4,a5@+
    movl #L00009,a5@+
    movl d0,a2@-
    jsr   _INS_ENV
    jbsr  _FILL_ENV
    movl a5,d0
    movl a4,a5@+
    movl #L00010,a5@+
    movl d0,a2@-
    movl a4@(4),d0
    movl d0,a2@-
    jbsr  _CONS2
    jra   _APPLY
L00008:
    jbsr  _CHECK
    movl a4@,a0
    movl a0@,a0
    movl a0@(4),d0
    jbsr  _EVAL
    jbsr  _CAR
    jbsr  _EVAL
    jra   _APPLY
L00009:
    jbsr  _CHECK
    movl a4@,a0
    movl a0@,a0
    movl a0@(4),d0
    jbsr  _EVAL
    jbsr  _CDR
    jbsr  _EVAL
    jra   _APPLY
L00010:
    jbsr  _CHECK
    movl a5,d0
    movl a4,a5@+
    movl #L00011,a5@+
    movl d0,a2@-
    movl a4@,a0
    movl a0@,a0
    movl a0@,a0
    movl a0@,a0
    movl a0@(4),d0
    jbsr  _EVAL
    jra   _APPLY
L00011:
    jbsr  _CHECK
    movl a5,d0
    movl a4,a5@+
    movl #L00012,a5@+
    movl d0,a2@-
    movl a5,d0
    movl a4,a5@+
    movl #L00015,a5@+
    movl d0,a2@-
    movl #_map,a0
    jbsr  _GLOB
    jbsr  _EVAL
    jra   _APPLY
L00012:
    jbsr  _CHECK
    movl a4@,a0
    movl a0@(4),d0
    movl d0,a2@-
    movl a5,d0
    movl a4,a5@+
    movl #L00013,a5@+
    movl d0,a2@-
    movl #_filter,a0
    jbsr  _GLOB
    jbsr  _EVAL
    jra   _APPLY
L00013:
    jbsr  _CHECK
    jsr   _EXT_ENV
    movl #T_INT+0,d0
    movl d0,a2@-
    movl #L00014,a0
    jbsr  _CALL
    jbsr  _NEQ
    jra   _APPLY
L00014:
    jbsr  _CHECK
    movl a4@,a0
    movl a0@(4),d0
    jbsr  _EVAL
    movl d0,a2@-
    movl a4@(4),d0
    jbsr  _EVAL
    jbsr  _MOD
    jra   _APPLY
L00015:
    jbsr  _CHECK
    jsr   _EXT_ENV
    movl a4@(4),d0
    jbsr  _EVAL
    jra   _APPLY
```

Appendix B

Syntax of *uc*

This appendix provides the concrete syntax of the *uc* language which has been extracted from the *yacc* specification used in the front-end translator.

- Operators are arranged in order of increasing binding power, and the associativity L, R, or N is shown in the comment field:

  L: left associative, R: right associative, N: non-associative

- Character constants *CHAR_CONST* and string constants *STRING_CONST* are denoted as in language C.

- Only the decimal notation is allowed for integer constants *INT_CONST*.

- Identifiers *IDENTIFIER* are [a-zA-Z][a-zA-Z_0-9]* in the *lex* notation.

- Tokens including reserved words are enclosed by ' '.

```
prog    :   expr
        ;
expr    :   expr1                       simple expression
        |   'fn' vars_s '.' expr        functional abstraction
        |   'let' decl_l 'in' expr      expr. with non-recursive decl.
        |   'letrec' decl_l 'in' expr   expr. with recursive decl.
        |   expr1 'where' decl_b        alternative for 'let ...'
        |   expr1 'whererec' decl_b     alternative for 'letrec ...'
        ;
expr1   :   expr2
        |   'if' s_expr 'then' expr1 'else' expr1
        ;
expr2   :   s_expr
        |   expr2 ',' expr2             pair (R)
        ;
s_expr  :   a_expr
        |   s_expr '++' s_expr          append (L)
        |   s_expr ':' s_expr           list cons (R)
        |   s_expr '||' s_expr          logical or (R)
        |   s_expr '&&' s_expr          logical and (R)
        |   s_expr '==' s_expr          equal to (N)
        |   s_expr '!=' s_expr          not equal to (N)
        |   s_expr '<' s_expr           less than (N)
        |   s_expr '>' s_expr           greater than (N)
        |   s_expr '<=' s_expr          less than or equal to (N)
        |   s_expr '>=' s_expr          greater than or equal to (N)
        |   s_expr '+' s_expr           sum (L)
```

```
            |   s_expr '-' s_expr                  difference (L)
            |   s_expr '*' s_expr                  product (L)
            |   s_expr '/' s_expr                  quotient (L)
            |   s_expr '%' s_expr                  remainder (L)
            |   '~' s_expr                         negation (N)
            |   '!' s_expr                         logical not (N)
            ;
a_expr      :   var                                variable
            |   var arg_l                          functional application
            |   b_expr                             basic expression
            ;
b_expr      :   'nil'                              list nil
            |   'true'                             Boolean true
            |   'false'                            Boolean false
            |   INT_CONST                          integer constant
            |   CHAR_CONST                         character constant
            |   STRING_CONST                       string constant
            |   '[' l_expr ']'                     enumeration of list elements
            |   '(' expr ')'                       parenthesized expression
            |   '(' operator ')'                   operators as function names
            ;
operator    :   '++' | ':' | '||' | '&&' | '==' | '!=' | '<' | '>' | '<=' | '>=' | '+' | '-' | '*' | '/' | '%' | '~' | '!'
            ;
arg_l       :   b_expr                             argument list
            |   var
            |   arg_l b_expr
            |   arg_l var
            ;
l_expr      :   /* empty */                        nil
            |   el_expr                            element list
            |   dot_expr                           dotdot expr.
            |   zf_expr                            Zermelo-Fraenkel expr.
            ;
el_expr     :   s_expr
            |   s_expr ',' el_expr
            ;
dot_expr    :   s_expr '..'                        from 's_expr' to infinity
            |   s_expr '..' s_expr                 interval
            ;
zf_expr     :   s_expr '|' qual_l                  qualified zf-expression
            ;
qual_l      :   qual
            |   qual_l ';' qual_l                  qualifier list
            ;
qual        :   expr1                              guard, i.e., filter
            |   var '<-' expr1                     generator
            ;
decl_l      :   decl
            |   decl_l 'and' decl_l                declaration list
            ;
decl_b      :   decl
            |   '{' decl_l '}'                     for 'where' and 'whererec'
            ;
decl        :   vars '=' expr                      simple binding
```

```
        |   var vars_s '=' expr          function definition
        ;
var     :   IDENTIFIER
        ;
vars_s  :   vars                        variable structure, i.e., compound binding
        |   vars vars_s
        ;
vars    :   var
        |   '(' vars_l ')'              structured variable
        ;
vars_l  :   vars
        |   vars_l ',' vars_l           pair
        |   vars_l ':' vars_l           list cons
        ;
```

Appendix C

An example of inserting injection operations

This appendix presents an example specification of a simple language shown in Figure 10.3.1, and the result of insertion of injection operations to that specification.

SDL specification

```
; Domains
((? (?))                                                    ; ?={?}
 (Int (Int))                                                ; Standard domains Int
 (Bool (Bool))                                              ; and Bool
 (State (State))                                            ; Abstract domain State
 (Cmd (+ (cmd-seq (* Cmd Cmd)) (cmd-output Exp)))           ; Cmd
 (Exp (+ (exp-int Int) (exp-bool Bool)))                    ; Exp
 (Cc (-> State Ans))                                        ; Command continuations
 (Ec (-> Val Cc))                                           ; Expression continuations
 (Ans (+ ("error") ((* Val Ans))))                          ; Answers
 (Val (+ (Int) (Bool))))                                    ; Values
; Functions
((pair (-> (0) (-> (1) (* (0) (1)))))                       ; pair : α→β→α×β
 (C (-> Cmd (-> Cc Cc))                                     ; Semantic function C
   (lambda (cmd c)
       (case cmd
           (cmd-seq ((pair cmd1 cmd2)) (C cmd1 (C cmd2 c)))
           (cmd-output (exp) (E exp (lambda (v s)
                                      (pair v (c s)))
                             ))))
  )
 (E (-> Exp (-> Ec Cc))                                     ; Semantic function E
   (lambda (exp k)
       (case exp (exp-int (e) (k e)) (exp-bool (e) (k e))))
  ))
```

Transformed SDL specification

```
((? (?))
 (Int (Int))
 (Bool (Bool))
 (State (State))
 (Cmd (+ (cmd-seq (* Cmd Cmd)) (cmd-output Exp)))
 (Exp (+ (exp-int Int) (exp-bool Bool)))
 (Cc (-> State Ans))
 (Ec (-> Val Cc))
 (Ans (+ (Ans1 (error)) (Ans2 (* Val Ans))))                ; Summand tags Ans1, Ans2, Val1,
 (Val (+ (Val1 Int) (Val2 Bool))))                          ; and Val2 have been generated.
((pair (-> (0) (-> (1) (* (0) (1)))))
 (C (-> Cmd (-> Cc Cc))
   (lambda (cmd)                                            ; Lambda-binding has been normalized.
       (lambda (c)
           (case cmd
               (cmd-seq ((pair cmd1 cmd2)) (C cmd1 (C cmd2 c)))
               (cmd-output (exp)
                       (E exp
                         (lambda (v)
                             (lambda (s)
```

```
                        (Ans2                                   ; Ans2 : Val×Ans→Ans
                         (pair v (c s))))
                    )
                ))))
    )
)
(E (-> Exp (-> Ec Cc))
  (lambda (exp)
      (lambda (k)
          (case exp
              (exp-int (e) (k (Val1 e)))                        ; Val1 : Int→Val
              (exp-bool (e) (k (Val2 e)))))                     ; Val2 : Bool→Val
      )
))
```