Eighteen

THE G-MACHINE

The heart of any graph reducer is the implementation of function application. In Chapter 12 we saw that a lambda abstraction can be applied to an argument by constructing an instance of the body of the abstraction with substitutions made for occurrences of the formal parameter. Unfortunately, this involved an inefficient traversal of the tree representing the body of the abstraction, and the presence of free variables seemed to make a more efficient implementation rather difficult.

With this in mind, we developed the supercombinator transformation in Chapter 13, which yielded particularly simple lambda abstractions (the supercombinators), which had no free variables. This simplified the process of instantiating the body of such an abstraction, but at the (minor) price of having to substitute for several variables at once. However, the principal incentive for developing the supercombinator transformation was the hope of compiling the body of a supercombinator to a fixed sequence of instructions which, when executed, would construct an instance of its body.

The payoff comes in this chapter, in which we will examine the G-machine, an extremely fast implementation of graph reduction based on supercombinator compilation. The G-machine was developed at the Chalmers Institute of Technology, Göteborg, Sweden, by Johnsson and Augustsson. This chapter and the subsequent three chapters draw heavily on the G-machine papers [Johnsson, 1984; Augustsson, 1984]. Many of the ideas in these chapters are theirs, and not all of them have appeared in the published literature.

The development of the G-machine is presented informally, but it would be an interesting exercise to give a formal proof of its correctness [Lester, 1985].

18.1 Using an Intermediate Code

Once we have decided to compile supercombinator bodies to a sequence of instructions we have to decide on the language in which the instructions should be written. It would be possible to produce, say, VAX machine code directly, but this approach suffers from two disadvantages. Firstly, we would have to start all over again if we want to generate code for some other machine, and, secondly, we would be in danger of mixing up the issues of how to compile supercombinators to a sequential code with issues of how best to exploit particular features of the VAX.

This is not a new problem, and a common solution is to define an *intermediate code*, which can be regarded as the machine code for an abstract sequential machine. Then the compilation process can be split into two parts: first generate the intermediate code, and then generate target code for a particular machine from the intermediate code. Changing the code generator to generate code for a different target machine is then relatively easy, and improvements made in the compilation to intermediate code benefit all such code generators. Examples of this approach include Pascal's P-code [Clark, 1981], BCPL's O-code [Richards, 1971] and Portable Standard Lisp's C-macros [Griss and Hearn, 1981].

18.1.1 G-code and the G-machine Compiler

For these reasons, the designers of the G-machine defined an intermediate code called G-code, into which supercombinator bodies are compiled. The compiler for the G-machine follows a sequence similar to that described in the first two parts of this book. In particular:

(i) The source language is a variant of ML with lazy evaluation semantics, called Lazy ML (or LML).
(ii) Early phases of the compiler perform type-checking, compile pattern-matching and do dependency analysis. At this stage the program has been translated to the lambda calculus (augmented with let and letrec).
(iii) A lambda-lifter transforms the program to supercombinator form. The full laziness optimization is not performed, but his feature could easily be added.
(iv) Now the supercombinators are compiled to G-code.
(v) Finally, machine code for the target machine is generated from the G-code.

Figure 18.1 shows the structure of the G-machine compiler.

Our description of the G-machine compiler falls into three parts:

(i) a description of the compilation algorithm which translates the source language into the intermediate code;
(ii) a description of the intermediate code itself, giving a precise description of what each instruction does;

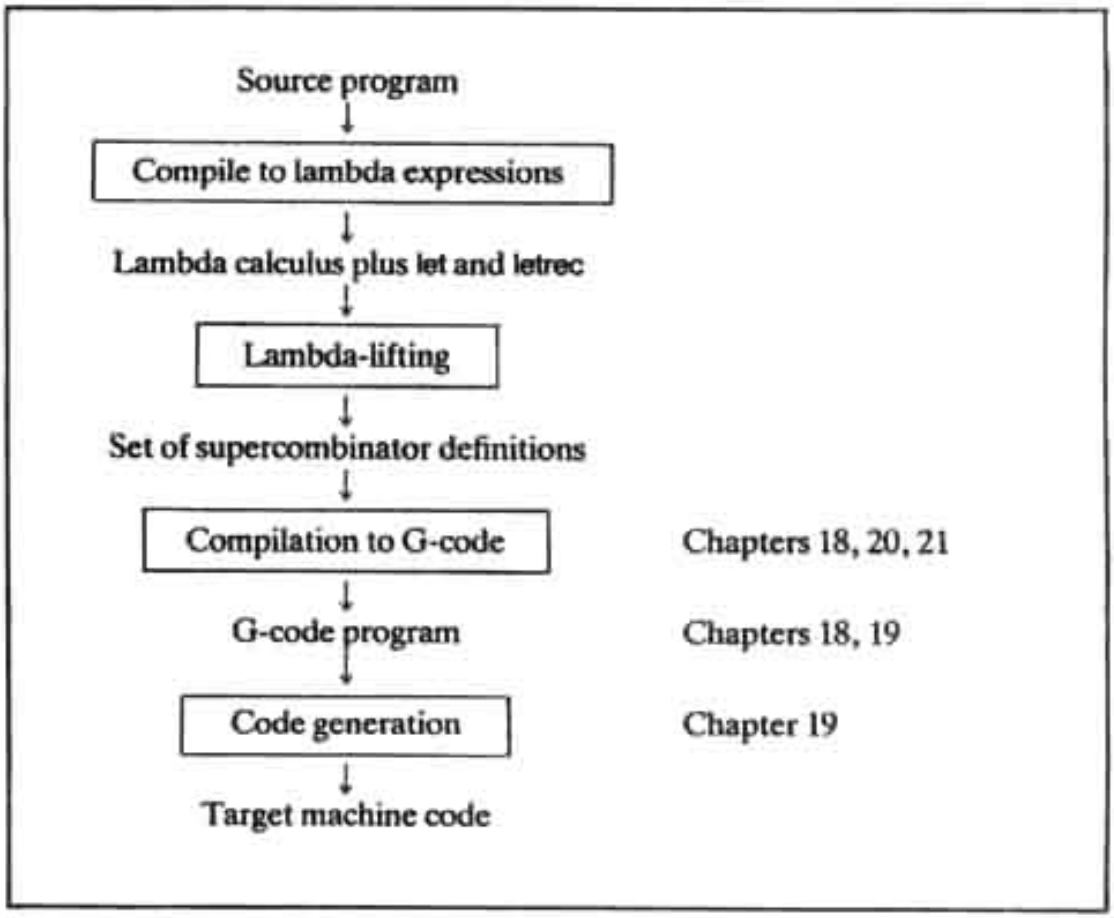


Figure 18.1 Structure of the G-machine compiler

(iii) a description of the code generator.

We will discuss the first of these parts in this chapter and the latter two in the next chapter. First, however, we will mention some related work.

18.1.2 Other Fast Sequential Implementations of Lazy Languages

The implementation of Ponder [Fairbairn, 1982], developed by Fairbairn and Wray, is based on a similar approach to the G-machine. The Ponder Abstract Machine (PAM) is at least as sophisticated as the G-machine, though they were developed independently, and is described in Fairbairn's thesis [Fairbairn, 1985; Fairbairn and Wray, 1986]. An interesting development of this work is a cross-compiler which compiles Ponder abstract machine instructions into SKIM microcode [Elworthy, 1985].

A related approach, though one which diverges from graph reduction, is to use a lexically scoped dialect of Lisp, such as Scheme [Steele and Sussman, 1978] or T [Rees and Adams, 1982], as an intermediate code. This takes advantage of the immense amount of effort which has been spent on building fast Lisp implementations, and is the approach taken by Hudak [Hudak and Kranz, 1984].

A fast VAX implementation of Hope [Burstall *et al.*, 1980] based on an intermediate code called FP/M has recently been developed at Imperial College [Field, 1985] (remember, however, that Hope is a strict language).

18.2 An Example of G-machine Execution

We begin with an example, to give the flavor of the G-machine. Consider the Miranda program

```
from n = n : from (succ n)
succ n = n+1
-----------------------------
from (succ 0)
```

It generates the infinite list [1,2,3,...,]. The functions from and succ are supercombinators already, so the lambda-lifting is trivial, yielding

```
$from n = CONS n ($from ($succ n))
$succ n = + n 1
$Prog = $from ($succ 0)
-----------------------------
$Prog
```

The G-machine uses a stack, and execution begins with a pointer to the initial graph on top of the stack (Figure 18.2(a)). The spine is then unwound, exactly as previously discussed in Section 11.6, without using pointer-reversal. The difference comes when the spine has been completely unwound, so that there is a pointer to $from on the stack (see Figure 18.2(b)). By following this pointer the machine extracts

(i) the number of arguments expected by $from (one in this case);
(ii) the starting address for the code for $from.

First it checks that there are enough arguments on the stack for $from to execute, and finds that there are. It then rearranges the top of the stack slightly (see the transition from (b) to (c) in Figure 18.2) and then jumps to the code for $from. The rearrangement of the top of the stack puts a pointer to the argument to $from on top of the stack. We will discuss the stack rearrangement in more detail later. Notice also that the machine *jumps* to $from rather than *calling* it. An instruction at the end of $from will complete evaluation of the graph after the $from reduction is done.

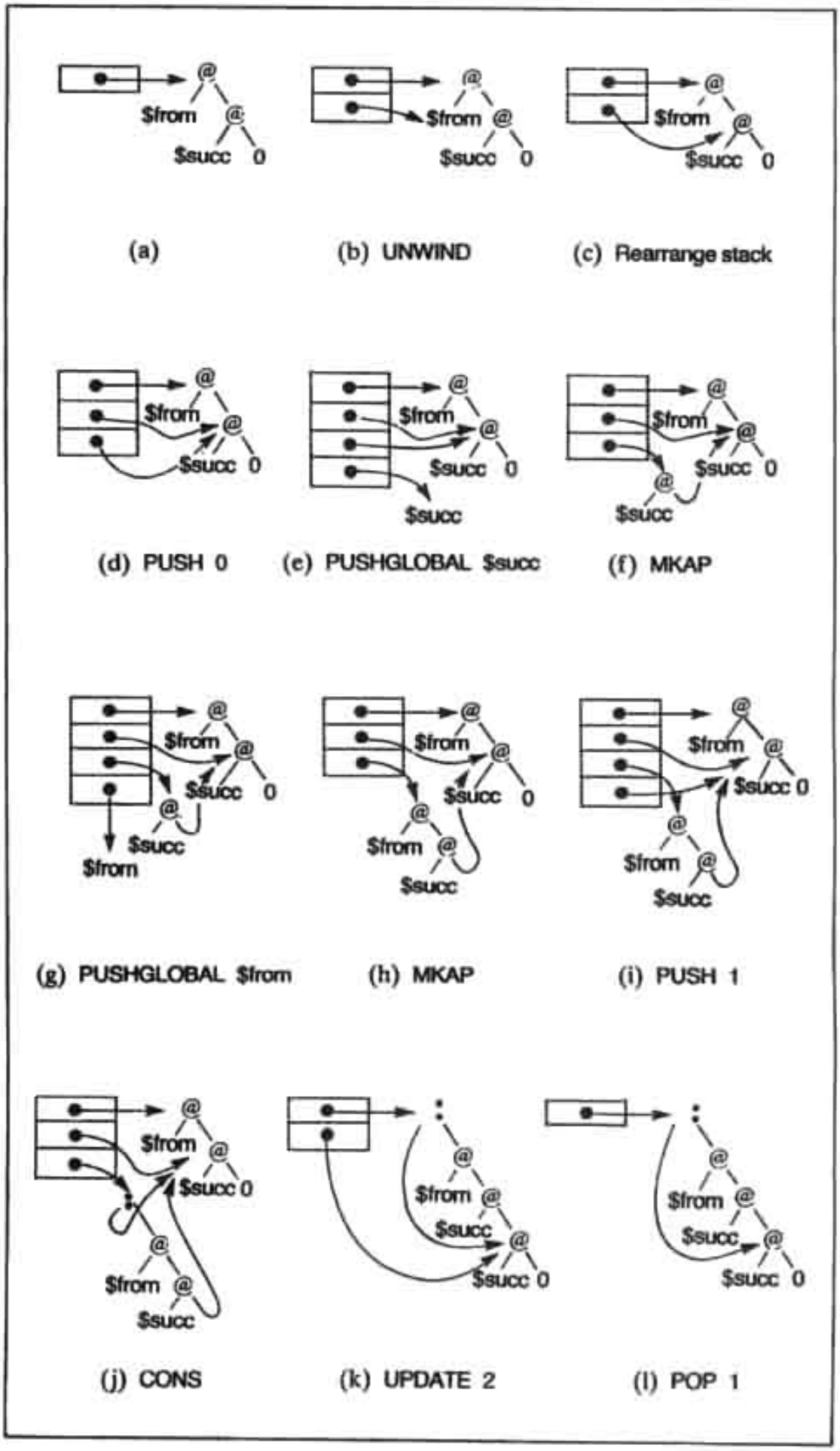


Figure 18.2 Evaluation of ($from ($succ 0))

Control has now passed to $from. A G-machine compiler will produce the following G-code for $from, which is executed in sequence:

G-code for function $from

```
PUSH 0;                  Push n
PUSHGLOBAL $succ;        Push function $succ
MKAP;                    Construct ($succ n)
PUSHGLOBAL $from;        Push function $from
MKAP;                    Construct ($from ($succ n))
PUSH 1;                  Push n
CONS;                    Construct (n : ($from ($succ n)))
UPDATE 2;                Update the root of the redex
POP 1;                   Pop the parameter n
UNWIND;                  Initiate next reduction
```

The execution of $from is shown step by step in Figure 18.2. We can make several observations by examining the code given above:

(i) At the point of entry, the parameter n is on top of the stack, and a pointer to the root of the redex is immediately below it (Figure 18.2(c)).

(ii) Items which are not on top of the stack are addressed *relative to the top of the stack*, with the top element having offset zero. For example, the PUSH 1 instruction takes the element next to top in the stack, and pushes it onto the stack. Stack items cannot be addressed relative to the base of the stack because a reduction takes place at the tip of the spine, with an unknown number of vertebrae above. (An alternative would have been to assume a frame pointer, and relegate offset calculation to code generation time.)

(iii) Some instructions take their operands from the stack and put their result on the stack in the manner of a zero address machine. MKAP and CONS are examples of such instructions.

Apart from the last three instructions, the sequence simply constructs an instance of the body of $from (see Figure 18.2(l)).

The UPDATE 2 instruction updates the root of the redex with a copy of the root of the result (there is a slight inefficiency here, since the root of the result is discarded almost immediately it is constructed; we will address this efficiency question later). Notice that the G-machine updates the root of the redex using copying, rather than using indirection nodes (but this is not an inherent property of the G-machine – see Section 19.4.4).

The POP 1 instruction removes the parameters (only one in this case) from the stack, leaving a pointer to the reduced graph on top of the stack. Finally UNWIND examines the tag of the root node of the reduced graph. In this case it is a CONS cell, so evaluation is complete.

This concludes our example, for now. (Note: in order to reduce the number

of execution steps, the example contains some optimizations which we will not study until Chapter 20.)

We will now develop the G-machine in a stepwise fashion, beginning with a very simple implementation, and developing the compilation algorithm and the G-code together. First, however, we will specify the language from which we are compiling.

18.3 The Source Language for the G-compiler

The compilation to G-code begins with a program consisting of a number of supercombinator definitions of the form

$$\$S\ x_1\ x_2\ \ldots\ x_n = E$$

where E is an expression containing no lambdas, but which may contain lets and letrecs. Figure 18.3 gives a reminder of the syntax of expressions. Notice

```
<E> ::= <constant>
      | <identifier>
      | <E> <E>
      | let <identifier> = <E> in <E>
      | letrec <identifier> = <E>
                ...
               <identifier> = <E>
           in <E>
```

Figure 18.3 BNF for syntax of expressions

that the left-hand side of a definition in a let or letrec can consist only of a single variable; local function definitions have been removed by lambda-lifting. For example,

```
let f x = + x 1
in E
```

cannot occur. Notice also that we allow only one definition in a let. Multiple definitions can be handled by nested lets, and the restriction slightly simplifies the compiler.

It is worth having a formal description of the syntax, because our compiler will need to contain a case for each construct. Referring to the syntax enables us to confirm that all cases have been covered.

To save repetitive work in this chapter we will use a stripped-down set of built-in functions and constants, shown in Figure 18.4. The stripped-down set has been chosen to illustrate all the features of the compiler. The operators in the right-hand column behave exactly like those in the left-hand column.

Assuming that we implement lists with structure tag 1 for NIL and 2 for

Stripped-down set	*Others which behave similarly*
integer constants	boolean, character constants
NEG (unary negation)	NOT
+	−, *, /, REM <, ≤, =, ≥, >
IF	CASE-n
FATBAR	
CONS	PACK-SUM-d-r, PACK-PRODUCT-r
HEAD	TAIL, SEL-r-l, SEL-SUM-r-l

Figure 18.4 Built-in functions and constants

CONS, we use CONS, HEAD and TAIL as abbreviations for PACK-SUM-2-2, SEL-SUM-2-1 and SEL-SUM-2-2 respectively. These abbreviations are easier to remember, and are used in the G-machine papers.

We do not treat UNPACK, since it is eliminated by the transformation described in Chapter 6.

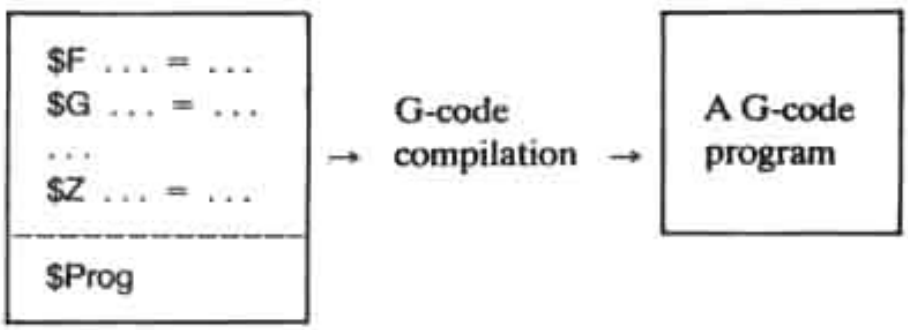
We will postpone a treatment of the FATBAR function until Chapter 20.

18.4 Compilation to G-code

For the rest of this chapter we will discuss the compilation of supercombinator definitions to G-code, leaving the code generation for the next chapter.

The compilation of a program to G-code and its execution by the G-machine are purely optimizations to the simpler template-instantiation implementation. We begin with the simplest possible G-machine, where the connection with template-instantiation is very direct. Later on, in Chapters 20 and 21, we will develop a number of optimizations which considerably speed up the operation of the machine.

The G-code compilation algorithm behaves like this:



The compilation algorithm takes a set of supercombinator definitions,

together with a distinguished one ($Prog), and produces a G-code program. The G-code program will consist of the following parts:

(i) A segment of initialization code, which will perform any run-time initialization necessary.
(ii) A segment of G-code which evaluates the distinguished supercombinator $Prog and prints its value. This will probably follow immediately after (i).
(iii) A segment of G-code corresponding to each supercombinator definition. Each of these will be identified by an initial label.
(iv) Labelled segments of G-code corresponding to each built-in function (such as + or CONS). This constitutes the run-time library, since it is the same for all programs.

The code segments for (i) and (ii) can be fairly simple. All we need for (i) is a G-code instruction BEGIN which labels the beginning of the program and initializes anything necessary. Then to evaluate $Prog we will first push it onto the stack (using a G-code instruction PUSHGLOBAL), then evaluate it (using the EVAL instruction) and then print it (using the PRINT instruction). Here is a code sequence that could be generated to initialize the system and print $Prog:

```
BEGIN;                  Beginning of program

PUSHGLOBAL $Prog;       Push $Prog onto stack
EVAL;                   Evaluate it
PRINT;                  Print the result

END                     End of program
```

We have felt free to invent G-code instructions out of thin air to perform the steps of the program. We will continue to do this, and will wait until the next chapter before giving them a more precise meaning. The EVAL instruction is discussed in Section 18.8.1.

We now turn our attention to (iii), compiling code for supercombinators, leaving (iv) for Section 18.8.

18.5 Compiling a Supercombinator Definition

We may depict the compilation of a supercombinator definition like this:

We can regard the compiler as a *function* F, which takes a supercombinator definition as its argument, and returns the compiled G-code as its result. Using the ⟦ ⟧ notation:

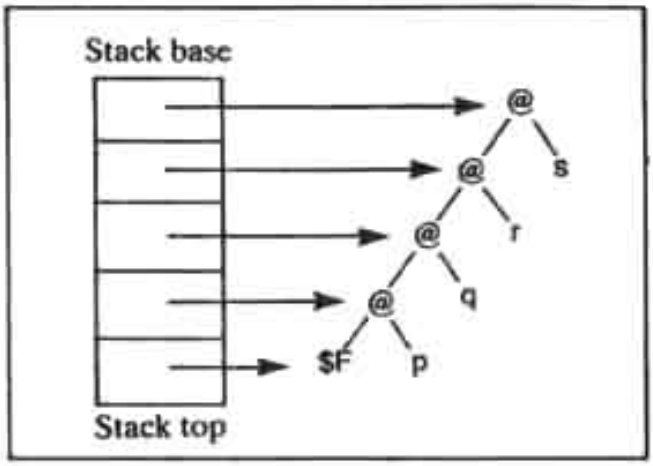
$$\mathcal{F}[\![\ \$F\ x_1\ x_2 = E\ ]\!] = \ldots \text{G-code for \$F} \ldots$$

We call the function F a *compilation scheme*, and we will use a number of other compilation schemes as auxiliary functions to F. Using this notation will allow us to express quite subtle compilation techniques in a compact and elegant way.

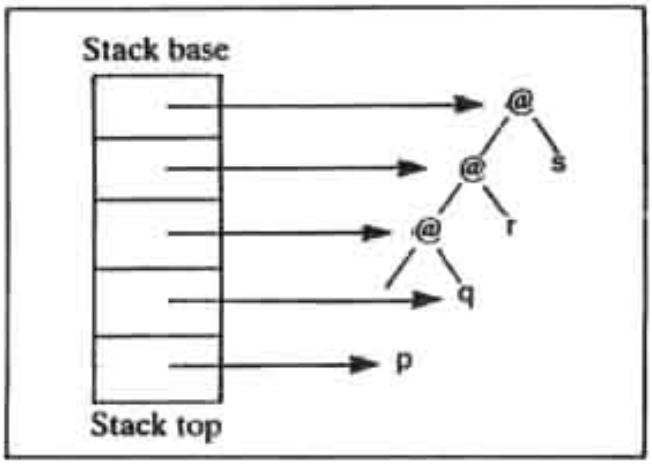
Now we will 'turn up the magnification' still more, and consider what the G-code for $F might look like. Before we can do this we must establish the context in which the code for $F will execute, and in particular the configuration of the stack which $F expects.

18.5.1 Stacks and Contexts

Suppose the G-machine was evaluating the expression ($F p q r s), and $F was a supercombinator of two arguments. After the spine of the graph has been unwound, the stack would look like this:



(In all the pictures the stack grows downwards.) This is not the most convenient configuration during execution of $F, because in order to access the arguments p and q it needs to do an indirect access via the vertebrae. The solution is to rearrange the stack after unwinding is complete, and before the supercombinator is executed, so that the elements on the stack point directly to the arguments, thus:



The rest of the spine is still there, of course, but it has not been drawn. Notice that we do retain a pointer to the root of the redex, because we will

subsequently need to update it. Now the arguments p and q are conveniently accessible. The supercombinator $F itself has been popped off, because this stack rearrangement is actually carried out by a prelude to the target code for $F.

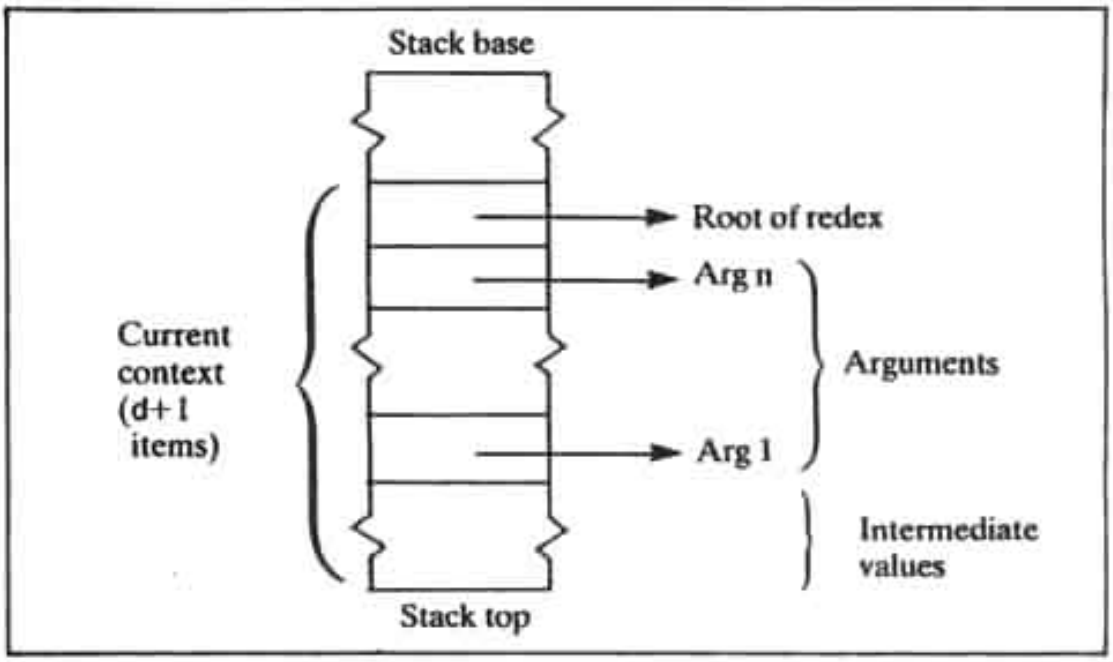


Figure 18.5 The stack during G-code execution

We see, therefore, that during the execution of the G-code for a supercombinator, the stack looks like Figure 18.5. The section at the top of the stack, including the pointer to the root of the current redex, the arguments and the intermediate values, is called the *current context*. It always sits at the top end of the stack, but there may be other stack elements between the stack base and the base of the current context. At the end of the execution of a function, the root of the redex will be updated and all the items in the context will be popped, leaving only the pointer to the root of the redex.

To summarize, here are two ground rules, which will hold throughout:

(1) When execution of (the code corresponding to) a supercombinator body begins, the arguments are on top of the stack, and underneath them is a pointer to the root of the redex.
(2) When execution of the supercombinator completes, only the pointer to the reduced graph remains on the stack. The reduced graph is not necessarily in WHNF, so the last instruction in the supercombinator initiates the next reduction.

During compilation of a supercombinator the compiler needs to maintain a model of what the stack looks like. In particular, it needs to know where the value of each variable is held, relative to the top of the stack. For all our compilation functions this information will be held as:

(i) ρ, a function which takes an identifier and returns a number giving the offset of the corresponding argument from the base of the current context, counting the bottom element of the context as having an offset of

0. The pointer to the root of the current redex therefore has an offset of 0, and the last argument has an offset of 1 (see Figure 18.5).

(ii) d, the depth of the current context minus one.

From these we can calculate the offset of a variable, x, from the top of the stack as (d − ρ x), counting the top element of the stack as having an offset of 0.

(Note: the G-machine paper [Johnsson, 1984] uses 'r' instead of 'ρ' and 'n' instead of 'd'. It also uses slightly different conventions for n and r (n = d+1 and r x = 1 + ρ x).)

For example, consider the context shown in Figure 18.6. The depth of the context is 5, so d=4. The function ρ maps the variable x to 2 and y to 1, and we write

$$\rho = [x=2,\ y=1]$$

The offset of the value of x from the top of the stack is

$$(d - \rho\ x) = (4 - 2) = 2$$

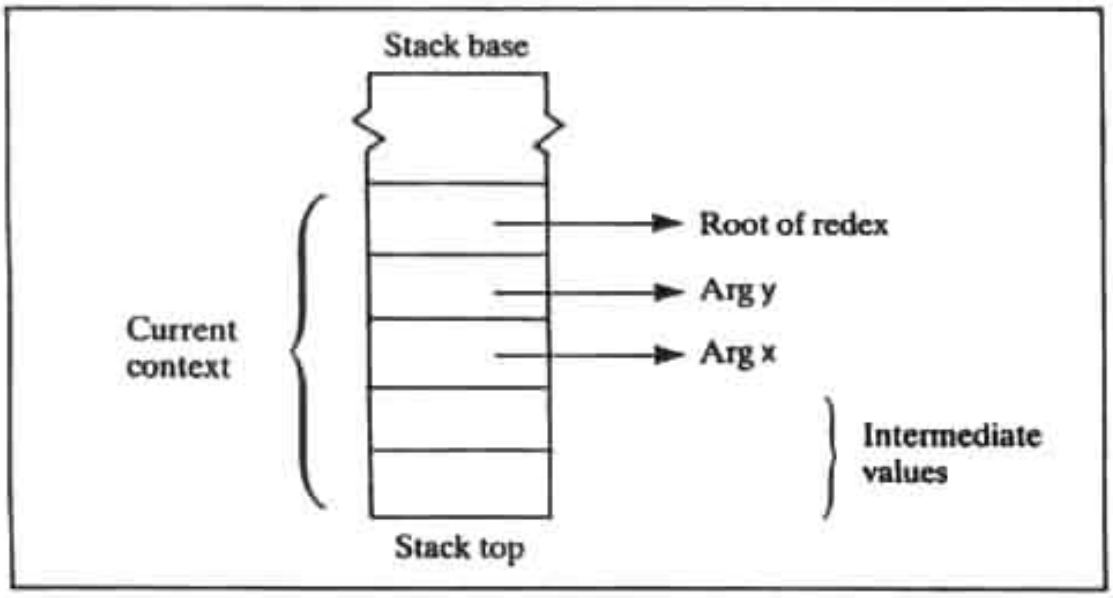


Figure 18.6 An example context

18.5.2 The R Compilation Scheme

We can now give the complete definition of the compilation scheme F we referred to above:

$$\mathcal{F}[\![\ f\ x_1\ x_2\ \ldots\ x_n = E\ ]\!]$$
$$= \text{GLOBSTART } f,n;\ \mathcal{R}[\![\ E\ ]\!]\ [x_1=n,\ x_2=n-1,\ \ldots,x_n=1]\ n$$

where f stands for a supercombinator name. The 'GLOBSTART f,n' is a G-code pseudo-instruction which labels the beginning of a function called f, which takes n arguments. Then F calls a function R to compile code for the body, E, of the supercombinator, passing it the correct ρ and d (in that order).

We must now describe what R does. As we saw in our example, the code for a supercombinator has to do four things:

(i) construct an instance of the supercombinator body, using the parameters on the stack;
(ii) update the root of the redex with a copy of the root of the result (note: there are the usual complications if the body consists of a single variable, which we deal with later);
(iii) remove the parameters from the stack;
(iv) initiate the next reduction.

This translates directly into a compilation scheme for R:

```
R[[ E ]] ρ d = C[[ E ]] ρ d; UPDATE (d+1); POP d; UNWIND
```

We use another auxiliary function, C (for Construct Instance), which produces code to construct an instance of E and put a pointer to it on the stack, which constitutes step (i). The UPDATE instruction overwrites the root of the redex (which is now at offset (d+1) from the top of the stack) with the newly created instance, which is currently on top of the stack (step (ii)); UPDATE then pops it from the stack. Then the POP instruction pops the arguments (step (iii)), and the UNWIND instruction initiates the next reduction (step (iv)). Figure 18.7 summarizes the F and R compilation schemes.

Warning: while it will give the correct results, the code generated by R may give bad performance for projection functions, such as

```
f x y z = y
```

where the body of the function consists of a single variable. The reasons for this were explained in Section 12.4. As given, the UPDATE instruction generated by the R scheme will copy the root of the argument y, without first evaluating it. This risks duplicating the root of a redex, which would lose laziness. We will fix this problem in the next version of R, at the beginning of Chapter 20.

All we have left to do is to describe the C compilation scheme.

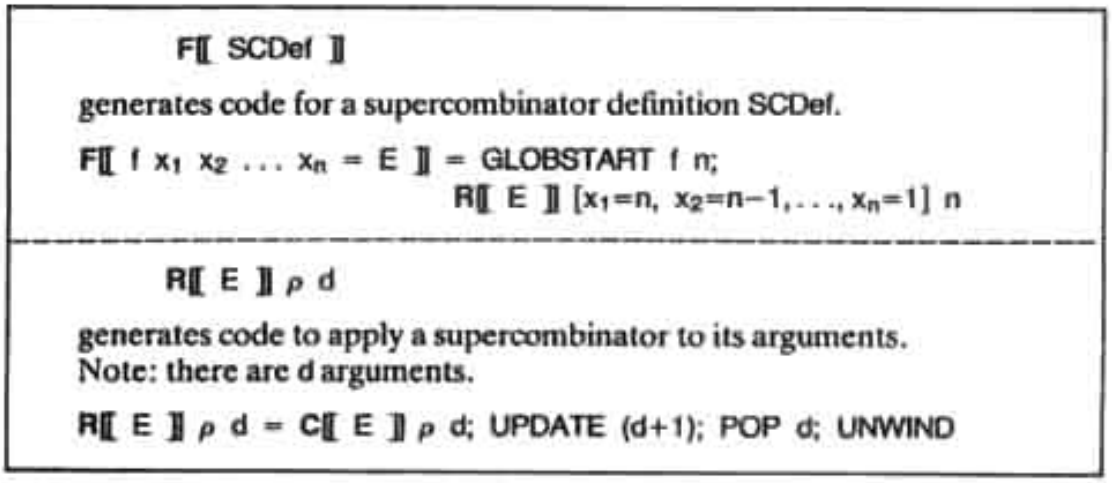


Figure 18.7 The R compilation scheme

18.5.3 The C Compilation Scheme

The C compilation scheme compiles code to construct an instance of an expression. It is a function with the following behavior:

(i) *Arguments*: the expression to be compiled, plus ρ and d, which specify where the arguments of the supercombinator are to be found in the stack.
(ii) *Result*: a G-code sequence which, when executed, will construct an instance of the expression, with pointers to the supercombinator arguments substituted for occurrences of the corresponding formal parameters, and leave a pointer to the instance on top of the stack.

To define C fully, we must specify the result of the call

C⟦ E ⟧ ρ d

for every possible expression E. The expression E can take a number of forms (see Figure 18.3), and we define C by specifying it separately for each form of E. The cases are described in the following sections.

18.5.3.1 E is a constant

There are actually two cases to consider here. First, suppose E is an integer, i (or a boolean, or other built-in constant value). All we need do is to push a pointer to the integer onto the stack (or the integer itself in an unboxed implementation), an operation which is carried out by the G-code instruction

```
PUSHINT i
```

We may write the compilation rule like this:

C⟦ i ⟧ ρ d = PUSHINT i

Secondly, suppose E is a supercombinator or built-in function, called f. We must push a pointer to the function onto the stack, using the G-code instruction

```
PUSHGLOBAL f
```

We write the rule in the same way as before:

C⟦ f ⟧ ρ d = PUSHGLOBAL f

18.5.3.2 E is a variable

The next case to consider is that of a variable, x. The value of the variable is in the stack, at offset (d − ρ x) from the top, and the G-code instruction

PUSH (d − ρ x)

will copy this item onto the top of the stack. Hence we may write the rule

C⟦ x ⟧ ρ d = PUSH (d − ρ x)

18.5.3.3 E is an application

If E is an application (E_1 E_2), where E_1 and E_2 are arbitrary expressions, then the expression to be constructed is the application of E_1 to E_2. It is easy to do this: first construct an instance of E_2 (leaving a pointer to the instance on top of the stack), then construct an instance of E_1 (likewise), then make an application cell from the top two items on the stack, and leave a pointer to the application cell on top of the stack. This can be achieved by the following rule:

$$C[\![\ E_1\ E_2\ ]\!]\ \rho\ d = C[\![\ E_2\ ]\!]\ \rho\ d;\ C[\![\ E_1\ ]\!]\ \rho\ (d+1);\ MKAP$$

Notice that the current context is one deeper during the second call to C, so we passed it (d+1) instead of d.

MKAP is an instruction which takes the top two items on the stack, pops them, forms an application node in the heap, and pushes a pointer to this node onto the stack. If MKAP took its arguments in the other order, we could construct first E_1 and then E_2. This might seem to be a more logical order, but we will see later that it is more convenient to construct E_2 first.

18.5.3.4 E is a let-expression

Next, consider the rule for let-expressions

$$C[\![\ \text{let}\ x = E_x\ \text{in}\ E_b\ ]\!]\ \rho\ d$$

where x is a variable and E_x, E_b are expressions (we consider only the case of a single definition). We recall that a let in a supercombinator body is just a way of describing a graph (with sharing) rather than a tree. We can deal with let in a very straightforward way.

(i) First we construct an instance of E_x, leaving a pointer to it on the stack.
(ii) Then we augment ρ to say that x is to be found at offset (d+1) from the base of the context (which is true, since it is on top of the stack).
(iii) Then we construct an instance of E_b, using the new values of ρ and d, leaving a pointer to the instance on top of the stack.
(iv) Now a pointer to the instance of E_b is on top of the stack, and underneath it is a pointer to the instance of E_x. We no longer want the latter, so we squeeze it out by sliding down the top element of the stack on top of it.

Figure 18.8 shows the execution of a let after these four stages.

In symbols:

$$C[\![\ \text{let}\ x = E_x\ \text{in}\ E_b\ ]\!]\ \rho\ d$$
$$= C[\![\ E_x\ ]\!]\ \rho\ d;\ C[\![\ E_b\ ]\!]\ \rho[x=d+1]\ (d+1);\ SLIDE\ 1$$

Remembering that ρ is a function taking a variable as its argument, the notation '$\rho[x=d+1]$' means 'a function which behaves just like ρ except when it is applied to x, in which case it delivers the result (d+1)'. In other words,

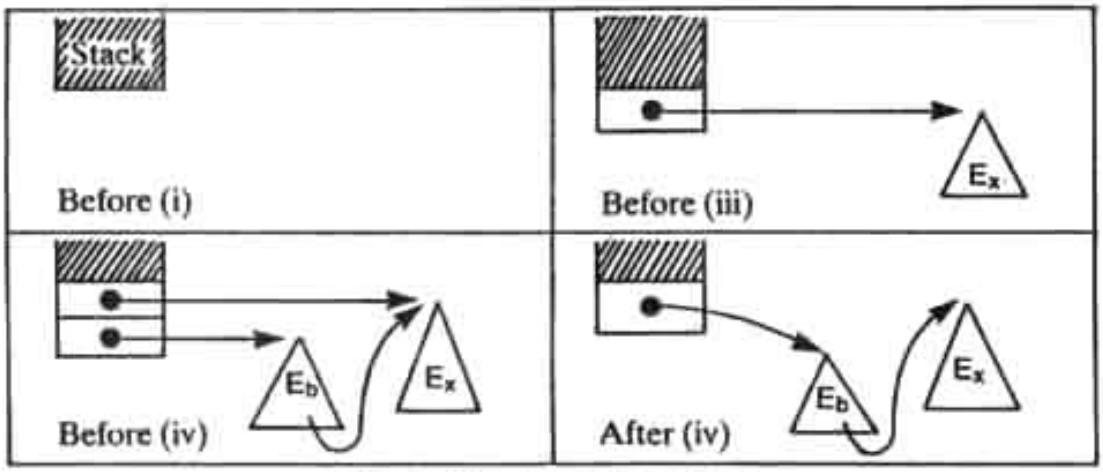


Figure 18.8 Execution of a let

ρ[x=d+1] is just ρ augmented with information about where to find x. Symbolically,

$$\rho[x=n]\ x = n$$
$$\rho[x=n]\ y = \rho\ y \qquad \text{if } x \neq y$$

The 'SLIDE 1' instruction squeezes out one element from the stack.

The job was fairly easy to do because we could access the graph constructed by the let definition in just the same way as we access the parameters of a supercombinator. This is another strong reason for performing the stack rearrangement described in Section 18.5.1.

18.5.3.5 E is a letrec-expression

Finally, we consider the rule for

$$\mathbf{C}[\![\ \text{letrec D in } E_b\ ]\!]\ \rho\ d$$

where D is a set of definitions and E_b is an expression. Recall that a letrec in a supercombinator body is just a description of a *cyclic graph*. The way to construct such a graph is:

(i) First allocate some empty cells, one for each definition, putting pointers to them on the stack. These empty cells are called *holes*.
(ii) Now augment the context ρ and d to say that the values of the variables bound in the letrec can be found in the stack locations just allocated.
(iii) Then for each definition body:

  (a) construct an instance of it, leaving a pointer to the instance on top of the stack, and
  (b) then update its corresponding hole with the instance (using the UPDATE instruction; this also removes the pointer on top of the stack).

During the instantiation process, occurrences of names bound in the letrec will be replaced by pointers to the corresponding hole, because we have augmented the context in stage (ii).

(iv) Now instantiate E_b, leaving a pointer to it on the stack.
(v) Lastly, squeeze out the pointers to the definition bodies. This is why the SLIDE instruction has an argument, telling it how many elements to squeeze out.

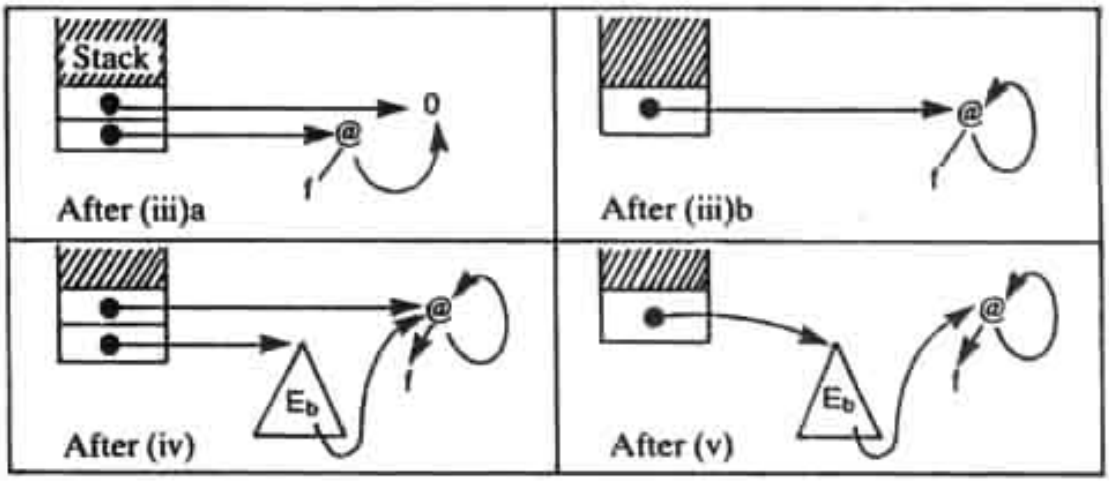


Figure 18.9 Execution of letrec x = f x in E_b

Figure 18.9 shows various stages in the execution of

$$\mathbf{C}[\![\ \text{letrec } x = f\ x \text{ in } E_b\ ]\!]\ \rho\ d$$

In symbols, we write:

$$\begin{aligned}&\mathbf{C}[\![\ \text{letrec } D \text{ in } E_b\ ]\!]\ \rho\ d\\ &= \mathbf{CLetrec}[\![\ D\ ]\!]\ \rho'\ d';\ \mathbf{C}[\![\ E_b\ ]\!]\ \rho'\ d';\ \text{SLIDE } (d'-d)\\ &\text{where}\\ &(\rho',\ d') = \mathbf{Xr}[\![\ D\ ]\!]\ \rho\ d\end{aligned}$$

This uses two new auxiliary functions **CLetrec** and **Xr**, which are defined as follows.

$$\begin{aligned}\mathbf{CLetrec}\ &\left[\!\!\left[\begin{array}{l}x_1 = E_1\\ x_2 = E_2\\ \ldots\\ x_n = E_n\end{array}\right]\!\!\right]\ \rho\ d = \text{ALLOC } n;\\ &\qquad \mathbf{C}[\![\ E_1\ ]\!]\ \rho\ d;\ \text{UPDATE } n;\\ &\qquad \mathbf{C}[\![\ E_2\ ]\!]\ \rho\ d;\ \text{UPDATE } n-1;\\ &\qquad \ldots\\ &\qquad \mathbf{C}[\![\ E_n\ ]\!]\ \rho\ d;\ \text{UPDATE } 1;\end{aligned}$$

CLetrec performs the first two steps of the process. The 'ALLOC n' instruction allocates n holes in the heap and pushes pointers to them onto the stack. Then the instances of the definition bodies are constructed and the UPDATE instruction overwrites a hole with the root of the corresponding instance.

$$\mathbf{Xr}\ \left[\!\!\left[\begin{array}{l}x_1 = E_1\\ x_2 = E_2\\ \ldots\\ x_n = E_n\end{array}\right]\!\!\right]\ \rho\ d = (\ \rho\left[\begin{array}{l}x_1=d+1\\ x_2=d+2\\ \ldots\\ x_n=d+n\end{array}\right],\ d+n\ )$$

Xr just computes the augmented ρ and the new value of d, returning them as a pair (ρ', d'). The [ ] bracket updates ρ to include all the new information.

The final 'SLIDE (d'−d)' slides down the top element of the stack, squeezing out the pointers to the E_i.

Warning: there will be a problem if a definition body consists of a single variable name bound in the same letrec; for example

```
letrec x = y
       y = CONS 1 y
in E
```

This gives a problem because UPDATE will try to update one hole with another. However, the definition of x will be removed at an earlier stage in the compiler, by the optimization of Section 14.7.3, which replaces occurrences of x by y in E.

C⟦ E ⟧ ρ d

Constructs the graph for an instance of E in a context given by ρ and d. It leaves a pointer to the graph on top of the stack.

C⟦ i ⟧ ρ d	= PUSHINT i
C⟦ f ⟧ ρ d	= PUSHGLOBAL f
C⟦ x ⟧ ρ d	= PUSH (d − ρ x)
C⟦ E_1 E_2 ⟧ ρ d	= **C⟦ E_2 ⟧ ρ d**; **C⟦ E_1 ⟧ ρ (d+1)**; MKAP
C⟦ let x=E_x in E ⟧ ρ d	= **C⟦ E_x ⟧ ρ d**; **C⟦ E ⟧ ρ[x=d+1] (d+1)**; SLIDE 1
C⟦ letrec D in E ⟧ ρ d	= **CLetrec⟦ D ⟧ ρ' d'**; **C⟦ E ⟧ ρ' d'**; SLIDE (d'−d) where (ρ', d') = **Xr⟦ D ⟧ ρ d**

Figure 18.10 The C compilation scheme

CLetrec⟦ D ⟧ ρ d

Takes a mutually recursive set of definitions D, constructs an instance of each body, and leaves the pointers to the instances on top of the stack.

$$\textbf{CLetrec}\left[\!\!\left[\begin{array}{l} x_1 = E_1 \\ x_2 = E_2 \\ \cdots \\ x_n = E_n \end{array}\right]\!\!\right] \rho\ d =$$

ALLOC n;
C⟦ E_1 ⟧ ρ d; UPDATE n;
C⟦ E_2 ⟧ ρ d; UPDATE n−1;
...
C⟦ E_n ⟧ ρ d; UPDATE 1;

Xr⟦ D ⟧ ρ d

Returns a pair (ρ', d') which gives the context augmented by the definitions D.

$$\textbf{Xr}\left[\!\!\left[\begin{array}{l} x_1 = E_1 \\ x_2 = E_2 \\ \cdots \\ x_n = E_n \end{array}\right]\!\!\right] \rho\ d = \left(\rho \left[\begin{array}{l} x_1=d+1 \\ x_2=d+2 \\ \cdots \\ x_n=d+n \end{array}\right],\ d+n \right)$$

Figure 18.11 Auxiliary compilation schemes **CLetrec** and **Xr**

18.5.3.6 Summary

We are done! The C compilation scheme has been described in considerable detail because the same ideas will be used again and again in what follows. It is worth some study to ensure that you understand what is going on. Figures 18.10 and 18.11 summarize the C scheme.

18.6 Supercombinators with Zero Arguments

The lambda-lifting algorithm given in earlier chapters may produce some supercombinators with no arguments. The most obvious example of this is the $Prog supercombinator.

Such supercombinators are simply *constant expressions* (sometimes called *constant applicative forms* or CAFs), since they have no parameters at all. The presence of CAFs raises two issues, compilation and garbage collection, which we now discuss.

18.6.1 Compiling CAFs

How should we compile CAFs? There are two alternatives:

(i) Do not compile them at all. Instead keep them as pieces of graph. Since they are not functions they will never be copied, so they can be shared without further ado. This is a perfectly acceptable solution, but it does mean that the compiled program is a mixture of target machine code and graph.

(ii) Treat them as supercombinators with zero arguments and compile them to G-code which will, when executed, construct an instance of their graph. Since we want to share this graph (and not make repeated copies of it) the instance should overwrite the compiled code in some way.

This is easily achieved. We allocate a single graph node, tagged as a function, which holds a pointer to the compiled code. This node is shared by anyone who uses the supercombinator. When the compiled code executes, the current context will contain a pointer to that node as its only element (since there are no arguments), so the node will be updated with the result, and this update will be seen by anyone else sharing the node. The F scheme is therefore quite adequate to compile the code for the body.

The advantage of this is that the compiled program consists almost entirely of target machine code, plus some individual graph nodes, one per supercombinator. In the Chalmers G-machine these nodes are allocated space physically adjacent to the target machine code of the supercombinator, outside the main heap. Such CAF nodes should not be in read-only memory, however, since they must be updated after their code is executed.

18.6.2 Garbage Collection of CAFs

Supercombinators which have one or more arguments need not be garbage-collected at all, since they cannot grow in size. CAFs, on the other hand, can grow in size without bound. For example, consider the program:

```
$from n = CONS n ($from (+ n 1))
$Ints = $from 1
$F x y = ...$Ints...
$Prog = ...$F...
-----------------------------------
$Prog
```

$Ints is the infinite list of integers, and we would like to recover the space this list occupies when it is no longer needed. Unfortunately, we will be unable to reclaim this space if we decide that all supercombinators should not be subject to garbage collection.

$Ints can be recovered when there are no references to it, directly or indirectly, from $Prog. However, $Prog may refer to $Ints indirectly, by using $F which uses $Ints, so we cannot recover $Ints just because $Prog does not refer to it directly.

The only clean way around this is to associate with each supercombinator (of any number of arguments, including zero) a list of CAFs to which it refers directly or indirectly. Then, for mark-scan garbage collection, to mark a supercombinator of one or more arguments we simply mark all the CAFs in its associated CAF list. To mark an unreduced CAF we mark its CAF list, while a reduced CAF is indistinguishable from any other heap structure and is marked as usual.

Another way to understand this is to see that in a template-instantiating implementation, the template for $F would refer to that for $Ints. Hence, $Ints would be reached by the mark phase of garbage collection during the normal marking traversal of $F. In a compiled implementation, however, the reference to $Ints is buried in the code for $F, and the CAF list for $F makes this dependency sufficiently explicit for the garbage collector to understand it.

This technique, or something similar, is essential to prevent ever-expanding CAFs from filling up the machine.

18.7 Getting it all Together

We can now put all the pieces together to describe how to compile a complete program. Consider the program:

```
$F x = NEG x
$Prog = $F 3
-----------------------------------
$Prog
```

(Note: such a program will never be generated by the lambda-lifter due to η-optimization, but it serves here as the smallest feasible example program.) This will compile to the following G-code:

```
BEGIN;                    Beginning of program
PUSHGLOBAL $Prog;         Load $Prog
EVAL; PRINT;              Evaluate and print it
END;

GLOBSTART $F, 1;          Beginning of $F (one argument)
PUSH 0;                   Push x
PUSHGLOBAL $NEG;          Push $NEG
MKAP;                     Construct ($NEG x)
UPDATE 2;                 Update the root of the redex
POP 1;                    Pop the parameter
UNWIND;                   Continue evaluation

GLOBSTART $Prog, 0;       Beginning of $Prog (no arguments)
PUSHINT 3;                Push 3
PUSHGLOBAL $F;            Push $F
MKAP;                     Construct ($F 3)
UPDATE 1;                 Update the $Prog
UNWIND;                   Continue evaluation
```

We have now described a complete compilation scheme for compiling a program into G-code. It is far from optimal, as we will soon see, but even in its present form it should work faster than a template-instantiation implementation.

The only mysterious feature of the above code is the function $NEG. It is one of the built-in functions in the run-time system, and we now describe the G-code for these functions.

18.8 The Built-in Functions

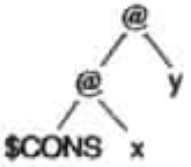
The names of built-in functions will appear in our implementation in three distinct ways. For example, CONS can appear in the following ways:

(i) As a (built-in) function in the supercombinator program. For example

```
$S x y = CONS y x
```

(ii) As a G-code instruction, which takes the top two elements on the stack, forms a CONS cell from them, and puts a pointer to the result on top of the stack (see Section 18.2).

(iii) As a built-in run-time function. For example, at run-time the machine may have to evaluate a graph like this



The spine will be unwound and the function $CONS will be found at the tip. Just as in the $from example (Section 18.2) the code for $CONS will be entered to perform the reduction. This means that there should be a G-code sequence for the $CONS function, and for all other built-in functions.

It is for this reason that we prefix this form of CONS with a $. At run-time it appears just like any user-defined supercombinator; that is as a (boxed) G-code sequence. In the next few chapters, therefore, we will not make any distinction between built-in functions and supercombinators. Sometimes we will call them *globals*; this is the origin of the PUSHGLOBAL instruction.

No confusion between the first two cases should arise, because the meaning should be clear from its context. One slight annoyance is that now we have

```
C[[ CONS ]] ρ d = PUSHGLOBAL $CONS
```

which makes it look as if C 'sticks the $ on a global', but this is contradicted by the case of a supercombinator:

```
C[[ $X ]] ρ d = PUSHGLOBAL $X
```

We content ourselves with the general rule as given in the C scheme, namely

```
C[[ f ]] ρ d = PUSHGLOBAL f
```

and remember that a $ is added to built-in functions. (This is, of course, a purely notational point.)

The third case above raises the question of what the G-code sequences for CONS and the other built-in functions are, and we will develop them in this section. The built-in functions we will consider are those given in the left-hand column of Figure 18.4; those in the right-hand column are analogous. In doing this we will also develop some new G-code instructions.

18.8.1 $NEG, $+, and the EVAL Instruction

NEGate is an example of a function which has to evaluate its argument. As we have seen before (Sections 11.4 and 12.2) this always seems to require a new mechanism for recursive argument evaluation, and the G-machine is no exception. The new mechanism we introduce is the G-code instruction EVAL, which evaluates the top item on the stack, leaving the evaluated object on the

stack. With the aid of this instruction we can give the following sequence for $NEG:

```
EVAL;       Evaluate the argument
NEG;        Negate it
UPDATE 1;   Update the root of the redex
UNWIND;     Continue
```

The code for $+ is similar, complicated only by having to get the appropriate parameter on top of the stack before calling EVAL:

```
PUSH 1;     Get second argument
EVAL;       Evaluate it
PUSH 1;     Get first argument
EVAL;       Evaluate it
ADD;        Add them
UPDATE 3;   Update root of redex
POP 2;      Pop parameters
UNWIND;     Evaluation is complete
```

The EVAL instruction does the following:

(i) Examines the object on top of the stack. If it is a CONS cell, an integer (boolean, character), a supercombinator or a built-in function, EVAL does nothing.
(ii) If it is an application cell, EVAL creates a new stack, pushes the top item of the old stack, saves the current program counter (which now points to the instruction after the EVAL), and then executes the UNWIND instruction.

After each reduction an UNWIND instruction is executed. If this UNWIND discovers that the expression is in WHNF, it restores the old stack and jumps to the saved return address.

As we saw in Section 11.6, we can build the new stack directly on top of the old stack. Indeed they can overlap by one item, since the top element of the old stack is the same as the bottom element of the new stack. We need to save two items on another stack, called the *dump*:

(i) the old stack depth, or (equivalently) the old stack pointer;
(ii) the old program counter.

The UNWIND instruction at the end of the code for $NEG or $+ will always discover that evaluation is complete, because we know that the result of a negation or addition is an integer. It is wasteful, therefore, for UNWIND to test the result for being in WHNF. We can encode this information by using a new instruction, RETURN, instead of UNWIND. RETURN assumes that the expression being evaluated is now in WHNF, but otherwise behaves just like UNWIND; that is, it restores the old stack and jumps to the saved program counter.

The new code for $NEG would therefore be:

```
EVAL;          Evaluate the argument
NEG;           Negate it
UPDATE 1;      Update the root of the redex
RETURN;        Evaluation is complete
```

18.8.2 $CONS

When the code for $CONS is entered, the two objects to be CONSed are on top of the stack, and below them is a pointer to the root of the redex. We can therefore produce the following code sequence for $CONS:

```
CONS;          Form the CONS cell
UPDATE 1;      Update the root of the redex
RETURN;        Result guaranteed to be in WHNF
```

CONS is a G-code instruction which CONSes together the top two items on the stack, pops them and pushes a pointer to the CONS cell. The CONS cell is then copied over the root of the redex by UPDATE. The CONS cell cannot be applied to anything (or the type-checker would have complained), so the expression being evaluated must now be in WHNF; we can thus use RETURN instead of UNWIND.

The treatment of $PACK-SUM-d-r is similar, except that we need a new G-code instruction PACKSUM d,r which constructs a structured data object with structure tag d and r fields, whose values are found on the stack. CONS is then equivalent to PACKSUM 2,2. $PACK-PRODUCT-r can be treated similarly, using a new G-code instruction PACKPRODUCT r. If sum types and product types are represented in the same way, then a single G-code instruction would suffice.

18.8.3 $HEAD

$HEAD is a function which evaluates its argument (to WHNF); it expects the result to be a CONS cell, from which it can extract the head (that is, the first field). Then, for the reasons we discussed in Section 12.4, it must evaluate the head of the cell before overwriting the root of the redex with it. Failing to do this final evaluation would result in the duplication of work.

The code for $HEAD is:

```
EVAL;          Evaluate to WHNF
HEAD;          Take its head
EVAL;          Evaluate the head
UPDATE 1;      Update root of redex
UNWIND;        Continue
```

Notice that we cannot use RETURN at the end, even though the result of the

HEAD must be in WHNF (since it has been EVALuated). Consider, for example, the expression

```
($HEAD E) 3
```

where E is some expression. Here, $HEAD evaluates E, takes its head, evaluates it, updates the ($HEAD E) redex and then *applies the result* to 3. Evaluation of the whole expression is not complete merely because the result of the ($HEAD E) reduction is in WHNF.

$TAIL and $SEL-SUM-r-i are precisely analogous to $HEAD, except that we need a new G-code instruction SELSUM r,i which selects the ith component of a structured data object of sum type and of size r. Similarly, $SEL-r-i (the selector functions for product types) requires the introduction of a new G-code instruction SELPRODUCT r,i. If sum and product types use the same representation, then only one new G-code instruction is required.

18.8.4 $IF, and the JUMP Instruction

In order to generate code for $IF we need to introduce two jump instructions (JUMP and JFALSE), and a label pseudo-instruction (LABEL).

The code for $IF is:

```
PUSH 0;          Get first argument
EVAL;            Evaluate it
JFALSE L1;       Jump to L1 if false
PUSH 1;          Get second argument
JUMP L2;

LABEL L1;        Pseudo-instruction; a label
PUSH 2;          Get third argument

LABEL L2;
EVAL;            Evaluate before overwriting
UPDATE 4;        Overwrite root
POP 3;           Pop arguments
UNWIND;          Continue
```

(L1 and L2 are unique labels.)

The reason for the last EVAL instruction was mentioned in the previous section, as was the reason for using UNWIND rather than RETURN.

In order to implement $CASE-n we need an n-way jump instruction,

```
CASEJUMP L1,L2,...,Ln
```

which examines the structure tag of the object on top of the stack, and jumps to one of n labels depending on its value. Apart from this, its treatment is identical to $IF, so we will not mention it any further.

18.9 Summary

This chapter has presented the payoff for the hard work earlier in the book. We have developed:

(i) a compilation algorithm which takes a supercombinator program and compiles it into G-code;
(ii) G-code sequences for a representative range of built-in functions.

The next chapter completes the picture by giving a precise description of G-code and a discussion on how to implement it.

References

Augustsson, L. 1984. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pp. 218–27, August.

Burstall, R.M., MacQueen, D.B., and Sanella, D.T. 1980. HOPE: an experimental applicative language. In *Proceedings of the ACM Lisp Conference*, pp. 136–43, August.

Clark, R. (editor) 1981. *UCSD P-system and UCSD Pascal Users' Manual*, 2nd edition. Softech Microsystems, San Diego.

Elworthy, D. 1985. Implementing a Ponder cross compiler for the SKIM processor. Dip. Comp. Sci. Dissertation, Computer Lab., Cambridge. July.

Fairbairn, J. 1982. Ponder and its type system. *Technical Report 31*. Computer Lab., Cambridge. November.

Fairbairn, J. 1985. Design and implementation of a simple typed language based on the lambda calculus. *Technical Report 75*. Computer Lab., Cambridge. May.

Fairbairn, J., and Wray, S.C. 1986. Code generation techniques for functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming, Boston*, pp. 94–104, August.

Field, A. 1985. *The Compilation of FP/M Programs into Conventional Machine Code*. Dept Comp. Sci., Imperial College. June.

Griss, M.L., and Hearn, A.C. 1981. A portable Lisp compiler. *Software – Practice and Experience*. Vol. 11, pp. 541–605.

Hudak, P., and Kranz, D. 1984. A combinator based compiler for a functional language. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pp. 122–32, January.

Johnsson, T. 1984. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction, Montreal*, pp. 58–69, June.

Lester, D. 1985. The correctness of a G-machine compiler. MSc dissertation, Programming Research Group, Oxford. December.

Rees, J.A., and Adams, N.I. 1982. T – a dialect of LISP. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 114–22, August.

Richards, M. 1971. The portability of the BCPL compiler. *Software – Practice and Experience*. Vol. 1, no. 2, pp. 135–46.

Steele, G.L., and Sussman, G.J. 1978. *The Revised Report on Scheme*. AI Memo 452, MIT. January.