Nineteen

G-CODE

Definition and Implementation

So far we have described a basic compilation algorithm from super-combinators into G-code. The next step, code generation, is to compile the G-code program into target machine code.

The basic idea is that to each G-code instruction there corresponds a simple sequence of target machine instructions, so that we can generate target code for a G-code program simply by generating these sequences for each instruction:

G-code	*Target machine code*
PUSH 3	<Target code for PUSH 3>
UPDATE 4	<Target code for UPDATE 4>

Typically the output of the code generator would be a program in the assembly code of the target machine, which would then be assembled, linked with any run-time libraries, and run.

In order to perform code generation in this way we need to know:

(i) exactly what each G-code instruction is supposed to do;
(ii) how the various bits of the abstract G-machine are mapped on to the target machine.

We will address these two issues in order.

19.1 What the G-code Instructions Do

The G-machine is a finite-state machine, with the following components:

(i) S, the stack.

(ii) G, the graph.
(iii) C, the G-code sequence remaining to be executed.
(iv) D, the dump. This consists of a stack of pairs (S, C), where S is a stack and C is a code sequence.

Thus the entire *state* of the G-machine is a 4-tuple <S, G, C, D>. We will describe the *operation* of the G-machine by means of *state transitions*. First, however, we need some notation for each component of the state.

19.1.1 Notation

A stack whose top item is n is written n:S, where S is a stack. An empty stack is written [ ].

A code sequence whose first instruction is I is written I:C, where C is a code sequence. An empty code sequence is written [ ].

A dump whose top pair is (S,C) is written (S,C):D, where D is a dump. An empty dump is written [ ].

The possible types of nodes in the graph are written like this:

INT i	an integer.
CONS n_1 n_2	a CONS node.
AP n_1 n_2	an application node.
FUN k C	a function (supercombinator or built-in) of k arguments, with code sequence C.
HOLE	a node which is to be filled in later. This is used for constructing cyclic graphs.

The notation G[n=AP n_1 n_2] stands for a graph in which node n is an application of n_1 to n_2 (n is just a name for this node). The notation G[n=G n'] stands for a graph in which node n has the same contents as node n' (we will need this only to describe the UPDATE instruction).

The *graph* is a logical concept, implemented by the *heap*. A *node* in the logical graph need not necessarily occupy a *cell* in the physical heap. In the case of CONS, AP, FUN and HOLE a logical node will indeed occupy a physical cell, but an INT node (i.e. an integer) will occupy a cell in a boxed implementation but will not in an unboxed implementation (see Section 10.6).

19.1.2 State Transitions for the G-machine

To illustrate the way in which we can use state transitions to describe the effect of instructions, consider the instruction PUSHINT i. We can write the following transition:

<S, G, PUSHINT i:C, D> ⇒ <n:S, G[n=INT i], C, D>

This says that when PUSHINT i is the first instruction, the G-machine makes a transition (denoted by ⇒) to a new state in which

(i) a new node n is pushed onto the stack,
(ii) the graph is updated with the information that node n is INT i,
(iii) the code to be executed is everything after the PUSHINT i,
(iv) and the dump is unchanged.

Notice that the name n, which is introduced on the right-hand side, is intended to be a new and unique node name.

More complicated instructions can be described using pattern-matching. EVAL is an example of this:

```
<n:S, G[n=AP n1 n2], EVAL:C, D>
⇒  <n:[], G[n=AP n1 n2], UNWIND:[], (S,C):D>

<n:S, G[n=FUN 0 C'], EVAL:C, D>
⇒ <n:[], G[n=FUN 0 C'], C':[], (S,C):D>

<n:S, G[n=INT i], EVAL:C, D>
⇒ <n:S, G[n=INT i], C, D>
```

and similarly for CONS and non-CAF FUN nodes.

The appropriate state transition for EVAL is selected depending on what kind of node is found on top of the stack (the node n):

(i) The first equation describes what EVAL does if the node on top of the stack is an application. The current stack and code are pushed onto the dump, a new stack is formed with the top of the old stack as its only element, and UNWIND is executed.
(ii) The second equation describes what EVAL does if the node on top of the stack is a compiled supercombinator of arity zero (that is, a CAF; see Section 18.6). In this case the machine saves its state on the dump, forms a new stack with the CAF as its only element, and executes the code associated with the CAF (which will subsequently update the FUN node with its reduced value).
(iii) The third equation describes what EVAL does if the node on top of the stack is an integer: it does nothing! The same applies if the node on top of the stack is a CONS or non-CAF function node.

An omitted transition indicates a run-time machine error (e.g. n is a HOLE).

Notice that in the first rule for EVAL we have (strictly speaking) to repeat the 'G[n=AP n1 n2]' on the right-hand side of the rule, since G alone would imply that node n was no longer in the graph. This is clumsy and hard to read, since the reader has to check that node n is the same on both sides of the rule. Accordingly we abbreviate the rule to

```
<n:S, G[n=AP n1 n2], EVAL:C, D>
⇒  <n:[], G, UNWIND:[], (S,C):D>
```

and imply that nodes not explicitly mentioned in the G field on the right-hand side are unchanged from the left-hand side.

Using this notation we can now give a complete description of the G-code instructions (Figures 19.1 and 19.2). The transitions for UNWIND are a little complicated, so we will explain them briefly. There are four cases:

(i) The item on top of the stack is an integer or a CONS node. In this case it must be the only element of the stack, and the expression being evaluated is in WHNF. UNWIND therefore completes evaluation by restoring the saved stack and code from the dump, and putting the result of the evaluation on the top of the restored stack.

(ii) The item on top of the stack is (a pointer to) an application node. In this case we just push the head of the application on the stack and repeat the UNWIND instruction.

(iii) The item on top of the stack is a function, and there are enough arguments on the stack. In this case we rearrange the stack as described in Section 18.5.1, and begin executing the code for the function. The v_i are the vertebrae on the spine, while the n_i are the arguments to the function.

(iv) The item on top of the stack is a function, but there are too few arguments for it to execute (this is described by the {a<k} condition). In this case the expression being evaluated is in WHNF, so UNWIND completes evaluation by restoring the saved stack and code from the dump, and putting the result of the evaluation on the top of the restored stack.

19.1.3 The Printing Mechanism

The G-code instructions developed so far are intended to reduce an expression to WHNF. As we saw in Section 11.2, though, we also need a printing mechanism which repeatedly invokes the evaluator to reduce expressions to WHNF and prints them. It would be nice if we could describe the printing mechanism within the same framework, and we now do so.

We introduce one new instruction, PRINT, which prints the top element on the stack. In order to describe its action we need to add one new component in the G-machine state: O, the output produced by the machine. The empty output is denoted by [ ], and O;x denotes the output O followed by the output x. Now we can define PRINT:

<O, n:S, G[n=INT i], PRINT:C, D> ⇒ <O;i, S, G, C, D>

<O, n:S, G[n=CONS n_1 n_2], PRINT:C, D>
⇒ <O, n_1:n_2:S, G, EVAL:PRINT:EVAL:PRINT:C, D>

All the other instructions leave O unchanged.

EVAL	<v:S, G[v=AP v' n], EVAL:C, D> ⇒ <v:[ ], G, UNWIND:[ ], (S,C):D>
	<n:S, G[n=FUN 0 C'], EVAL:C, D> ⇒ <n:[ ], G, C':[ ], (S,C):D>
	<n:S, G[n=INT i], EVAL:C, D> ⇒ <n:S, G, C, D> and similarly for CONS and non-CAF FUN nodes.
UNWIND	<n:[ ], G[n=INT i], UNWIND:[ ], (S,C):D> ⇒ <n:S, G, C, D> and similarly for CONS nodes.
	<v:S, G[v=AP v' n], UNWIND:[ ], D> ⇒ <v':v:S, G, UNWIND:[ ], D>
	<v_0:v_1:...:v_k:S, G[v_0=FUN k C, v_i=AP v_{i-1} n_i $(1 \le i \le k)$], UNWIND:[ ], D> ⇒ <n_1:n_2:...:n_k:v_k:S, G, C, D>
{a<k)	<v_0:v_1:...:v_a:[ ], G[v_0=FUN k C'], UNWIND:[ ], (S,C):D> ⇒ <v_a:S, G, C, D>
RETURN	<v_0:v_1:...:v_k:[ ], G, RETURN:[ ], (S,C) :D > ⇒ <v_k:S, G, C, D>
JUMP	<S, G, JUMP L:...:LABEL L:C, D> ⇒ <S, G, C, D>
JFALSE	<n:S, G[n=BOOL true], JFALSE L:C, D> ⇒ <S, G, C, D> <n:S, G[n=BOOL false], JFALSE L:...:LABEL L:C, D> ⇒ <S, G, C, D>

Figure 19.1 G-machine state transitions (control)

PUSH	<n_0:n_1:...:n_k:S, G, PUSH k:C, D> ⇒ <n_k:n_0:n_1:...:n_k:S, G, C, D>
PUSHINT	<S, G, PUSHINT i:C, D> ⇒ <n:S, G[n=INT i], C, D>
PUSHGLOBAL	similarly
POP	<n_1:n_2:...:n_k:S, G, POP k:C, D> ⇒ <S, G, C, D>
SLIDE	<n_0:n_1:...:n_k:S, G, SLIDE k:C, D> ⇒ <n_0:S, G, C, D>
UPDATE	<n_0:n_1:...:n_k:S, G, UPDATE k:C, D> ⇒ <n_1:...:n_k:S, G[n_k=G n_0], C, D>
ALLOC	<S, G, ALLOC k:C, D> ⇒ <n_1:n_2:...:n_k:S, G[n_1=HOLE, ..., n_k=HOLE], C, D>
HEAD	<n:S, G[n=CONS n_1 n_2], HEAD:C, D> ⇒ <n_1:S, G, C, D>
NEG	<n:S, G[n=INT i], NEG:C, D> ⇒ <n':S, G[n'=INT (−i)], C, D>
ADD	<n_1:n_2:S, G[n_1=INT i_1, n_2=INT i_2], ADD:C, D> ⇒ <n:S, G[n=INT (i_1+i_2)], C, D>
MKAP	<n_1:n_2:S, G, MKAP:C, D> ⇒ <n:S, G[n=AP n_1 n_2], C, D>
CONS	similarly

Figure 19.2 G-machine state transitions (stack and data)

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.

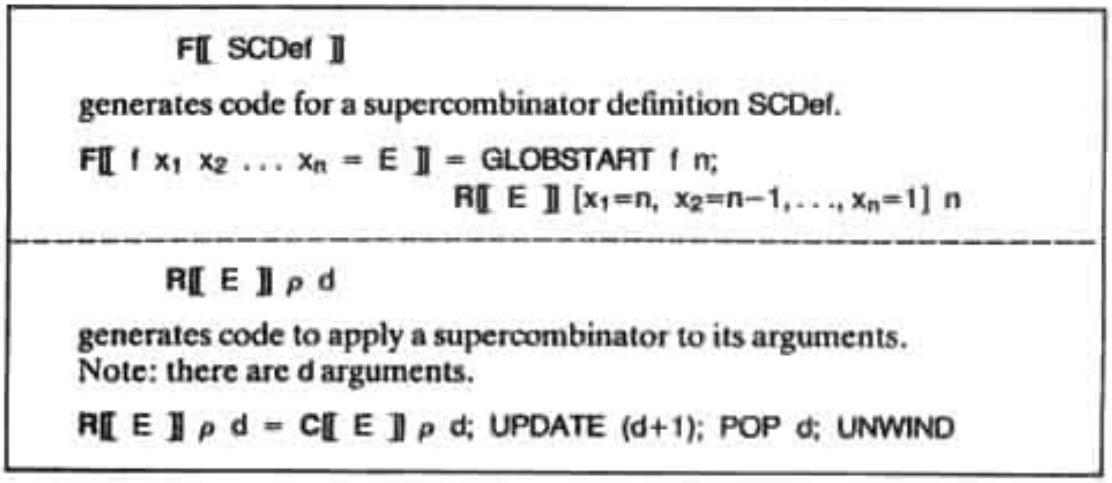


Figure 18.7 The R compilation scheme

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,

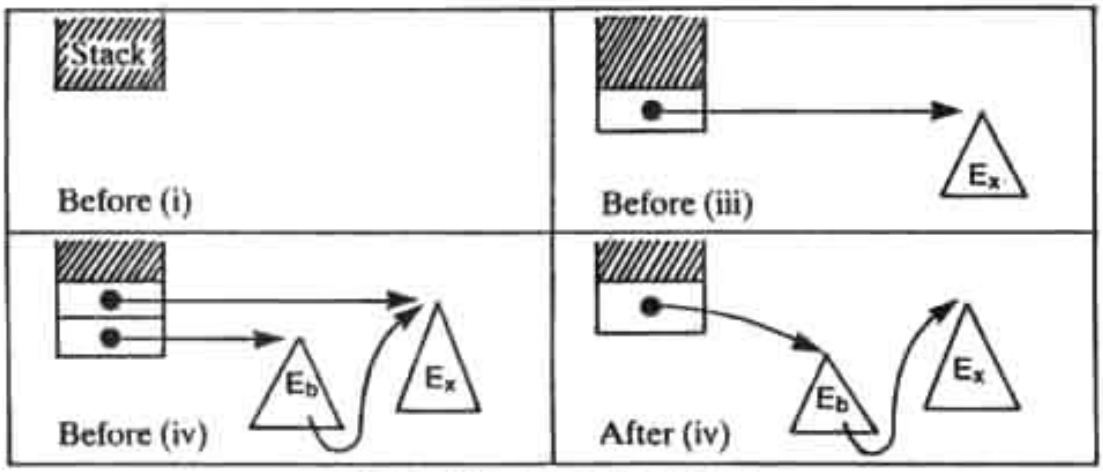


Figure 18.8 Execution of a let

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.

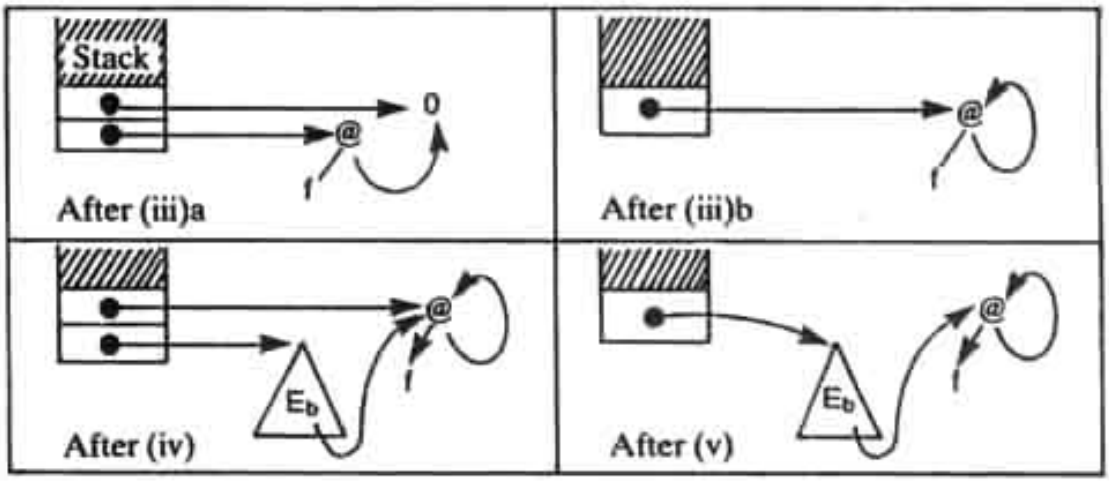


Figure 18.9 Execution of letrec x = f x in E_b

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

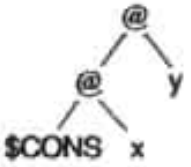
The names of built-in functions will appear in our implementation in three distinct ways. For example, CONS can appear in the following ways:

(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.