Chapter 3

The G-machine

In this chapter we introduce our first compiler-based implementation, the G-machine, which was developed at the Chalmers Institute of Technology, Göteborg, Sweden, by Augustsson and Johnsson. The material in this chapter is based on their series of papers [Augustsson 1984, Johnsson 1984] culminating in their Ph.D. theses [Augustsson 1987, Johnsson 1987].

3.1 Introduction to the G-machine

The fundamental operation of the template instantiation machine was to construct an instance of a supercombinator body, implemented by the `instantiate` function. This is a rather slow operation, because `instantiate` must recursively traverse the template *each time an instantiation is performed*. When we think of the machine instructions that are executed by `instantiate`, we see that they will be of two kinds: those concerned with traversing the template, and those concerned with actually constructing the instance.

The 'Big Idea' of the G-machine, and other compiled implementations, is this:

> *Before running the program, translate each supercombinator body to a sequence of instructions which, when executed, will construct an instance of the supercombinator body.*

Executing this code should be faster than calling an instantiation function, because all the instructions are concerned with constructing the instance. There are no instructions required to traverse the template, because all that has been done during the translation process. Running a program is thereby split into two stages. In the first stage a compiler is used to produce some intermediate form of the program; this is referred to as *compile-time*. In the second stage the intermediate form is executed; this is called *run-time*.

Since all we ever do to a supercombinator is to instantiate it, we can discard the original supercombinators once the translation is done, keeping only the compiled code.

In principle, then, we use a *G-machine compiler* to turn a program in our source language into a sequence of *machine language instructions*. Because we may wish to implement our language on many different pieces of hardware (68000 based, or VAX, etc.) it is useful to have an *abstract* *machine*. A good abstract machine has two properties: firstly, it can be easily translated into any concrete machine code (for example 68000 assembler); secondly, it is easy to generate the abstract machine code from the source.

Notice that we are faced with a trade-off here. We can ideally satisfy the first property (easy concrete code generation) by making the abstract machine the *same* as the real machine. But this makes the second property much harder to fulfil. An abstract machine is therefore a stepping-stone between the source language and a particular machine code.

3.1.1 An example

Here is a small example of the G-machine compiler in action. Consider the function

```
f g x = K (g x)
```

This would be compiled to the sequence of G-code instructions:

```
Push 1
Push 1
Mkap
Pushglobal K
Mkap
Slide 3
Unwind
```

In Figure 3.1, we show how this code will execute. On the left-hand side of each diagram is the stack, which grows downwards. The remainder of each diagram is the heap. The application nodes are represented by an @ character, expressions are labelled with lower-case letters, and supercombinators are labelled with upper-case letters.

In Figure 3.1, diagram (a), we see the state of the machine before executing the sequence of instructions for `f`. The spine has been unwound, just as it was in the template machine. The top two items on the stack are pointers to the application nodes, whose right-hand parts are the expressions to be bound for `g` and `x`.

The `Push` instruction uses addressing relative to the top of the stack. Ignoring the pointer to the supercombinator node `f`, the first stack item is numbered `0`, the next is numbered `1` and so on. The next diagram (b) shows the changed stack, after executing a `Push 1` instruction. This pushes a pointer to the expression `x` onto the stack, `x` being two stack items down the stack. After another `Push 1` we have a pointer to `g` on top of the stack; again this is two stack items down the stack, because the previous instruction pushed a new pointer onto the stack. The new diagram is (c).

Diagram (d) shows what happens when a `Mkap` instruction is executed. It takes two pointers from the stack and makes an application node from them; leaving a pointer to the result on the stack. In diagram (e) we execute a `Pushglobal K` instruction, with the effect of pushing a pointer to the `K` supercombinator. Another `Mkap` instruction completes the instantiation of the body of `f`, as shown in diagram (f).

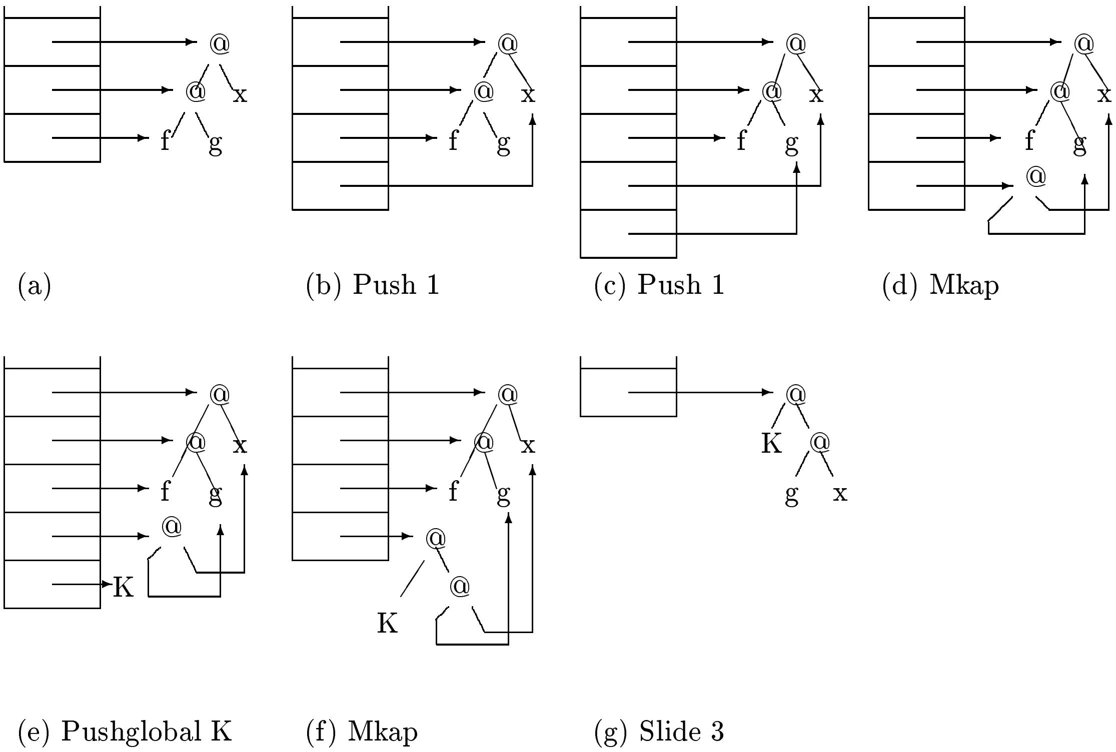


Figure 3.1: Execution of code for the `f` supercombinator

We can now replace the original expression, `f g x`, with the newly instantiated body: `K (g x)`. In the first version of the G-machine – which is not lazy – we simply slide the body down three places on the stack, discarding the three pointers that were there. This is achieved by using a `Slide 3` instruction, as shown in diagram (g). The final `Unwind` instruction will cause the machine to continue to evaluate.

This concludes a brief overview of the execution of the G-machine.

3.1.2 Further optimisations

A modest performance gain can be achieved by eliminating the interpretive overhead of traversing the template, as we have discussed. However, it turns out that compilation also opens the door to a whole host of short-cuts and optimisations which are simply not available to the template instantiation machine. For example, consider the following definition:

```
f x = x + x
```

The template machine would evaluate `x` twice; on the second occasion it would of course find that it was already evaluated. A compiled implementation can spot at compile-time that `x` will already be evaluated, and omit the evaluation step.

3.2 Code sequences for building templates

We recall that the template instantiator operates in the following way:

- The machine has *terminated* when the single item on top of the stack is a pointer to an integer.
- If this is not the case then we *unwind* any application nodes we come across until we reach a supercombinator node. We then *instantiate* a copy of the supercombinator body, making substitutions for its arguments.

At the heart of the Mark 1 template machine are the two functions `scStep` and `instantiate`, which are defined on pages 59 and 60. If we take a look at the definitions of `scStep` and `instantiate`, we can give the following description to the operation of instantiating a supercombinator:

1. Construct a *local environment* of variable names to addresses in the heap.
2. Using this local environment, make an instance of the supercombinator body in the heap. Variables are not copied; instead the corresponding address is used.
3. Remove the pointers to the application nodes and the supercombinator node from the stack.
4. Push the address of the newly created instance of the supercombinator onto the stack.

In the template instantiator, making an instance of a supercombinator involves traversing the tree structure of the expression which is the body of the supercombinator. Because expressions are defined recursively, the tree-traversal function `instantiate` is defined recursively. For example, look at the definition of `instantiate` – on page 60 – for the case of `EAp e1 e2`. First we call `instantiate` for `e1` and then for `e2`, holding on to the addresses of the graph for each sub-expression. Finally we combine the two addresses by building an application node in the graph.

We would like to compile a *linear sequence of instructions* to perform the operation of instantiating an expression.

3.2.1 Postfix evaluation of arithmetic

The desire to construct a linear sequence of instructions to instantiate an expression is reminiscent of the postfix evaluation of arithmetic expressions. We explore this analogy further before returning to the G-machine.

The language of arithmetic expressions consists of: numbers, addition and multiplication. We can represent this language as the type `aExpr`.

```
> data AExpr         = Num Int
>            | Plus AExpr AExpr
>            | Mult AExpr AExpr
```

It is intended that the language should have an 'obvious' meaning; we can give this using the function aInterpret.

```
> aInterpret :: AExpr -> Int
> aInterpret (Num n)      = n
> aInterpret (Plus e1 e2) = aInterpret e1 + aInterpret e2
> aInterpret (Mult e1 e2) = aInterpret e1 * aInterpret e2
```

Alternatively, we can compile the expression into a postfix sequence of operators (or instructions). To evaluate the expression we use the compiled operators and a stack of values. For example, the arithmetic expression $2 + 3 \times 4$ would be represented as the sequence

$$[\texttt{INum}\ 2,\ \texttt{INum}\ 3,\ \texttt{INum}\ 4,\ \texttt{IMult},\ \texttt{IPlus}]$$

We can give the instructions for our postfix machine as the type aInstruction.

```
> data AInstruction  = INum Int
>                    | IPlus
>                    | IMult
```

The state of the evaluator is a pair, which is a sequence of operators and a stack of numbers. The meaning of a code sequence is then given in the following transition rules.

$$(3.1)\quad \boxed{\begin{array}{lrr} & [\,] & [n] \\ \Longrightarrow & & n \end{array}}$$

$$(3.2)\quad \boxed{\begin{array}{lrr} & \texttt{INum}\ n : i & ns \\ \Longrightarrow & i & n : ns \end{array}}$$

$$(3.3)\quad \boxed{\begin{array}{lrr} & \texttt{IPlus} : i & n_0 : n_1 : ns \\ \Longrightarrow & i & (n_1 + n_0) : ns \end{array}}$$

$$(3.4)\quad \boxed{\begin{array}{lrr} & \texttt{IMult} : i & n_0 : n_1 : ns \\ \Longrightarrow & i & (n_1 \times n_0) : ns \end{array}}$$

Translating these transition rules into Miranda gives:

```
> aEval :: ([AInstruction], [Int]) -> Int
> aEval ([],        [n])      = n
> aEval (INum n:is, s)        = aEval (is, n:    s)
> aEval (IPlus: is, n0:n1:s) = aEval (is, n1+n0:s)
> aEval (IMult: is, n0:n1:s) = aEval (is, n1*n0:s)
```

To generate the sequence of postfix code for an expression we must define a compiler. This takes an expression and delivers a sequence of instructions, which when executed will compute the value of the expression.

```
> aCompile :: AExpr -> [AInstruction]
> aCompile (Num n)      = [INum n]
> aCompile (Plus e1 e2) = aCompile e1 ++ aCompile e2 ++ [IPlus]
> aCompile (Mult e1 e2) = aCompile e1 ++ aCompile e2 ++ [IMult]
```

The key idea from this is given by the type of the `aCompile` function. It returns a list of instructions.

> *The postfix representation of expressions is a way of* flattening *or* linearising *an expression tree, so that the expression can be represented by a flat sequence of operators.*

Exercise 3.1. Using structural induction, or otherwise, prove that the postfix evaluation of arithmetic expressions results in the same answer as the tree evaluation of expressions. That is: prove that for all expressions `e` of type `aExpr`,

$$\texttt{aInterpret e} = \texttt{aEval (aCompile e, [])}$$

This is an example of a *congruence proof.*

Exercise 3.2. Extend the functions `aInterpret`, `aCompile` and `aEval` to handle `let` expressions. Prove that for all expressions in `e` of type `aExpr`, these new functions satisfy the relation:

$$\texttt{aInterpret e} = \texttt{aEval (aCompile e, [])}$$

Can you extend the language to even more complicated expressions, e.g. `letrec` expressions? Can you prove that you have correctly implemented these extensions?

3.2.2 Using postfix code to construct graphs

We can use the same technique to create an instance of a supercombinator body. In this case the 'values' on the stack will be addresses of parts of the expression being instantiated.

The operations of the template construction instructions will be different from those we saw in the arithmetic example above, in that the instructions generally have the side-effect of allocating nodes in the heap. As an example, consider introducing an `Mkap` instruction. This instruction makes an application node, in the heap, from the top two addresses on the stack. It leaves a pointer to this new node on the stack upon completion.

There is no reason to invent a new evaluation stack of addresses, as our template instantiation machine already has such a stack. However, there is an important point to remember if we do make use of this stack:

> *The map of the* stack locations *corresponding to variable names will change as we pop and push objects from the stack. We must therefore keep track of this when we are compiling expressions.*

Our access to items in the stack is relative to the top of the stack. So, if an item is added, the offset to reach that item is increased by one; similarly, when an item is popped, the offset is decreased by one.

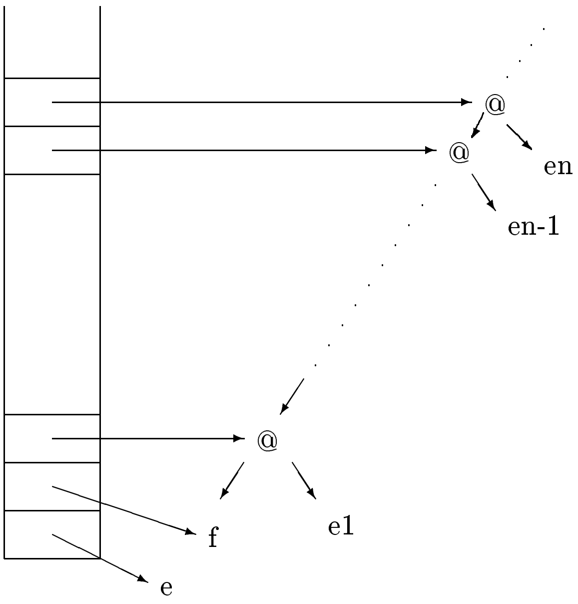


Figure 3.2: The stack layout for the Mark 1 machine

3.2.3 What happens after an instantiation has been made?

Once the instantiation of the supercombinator body has been made we must tidy up the stack, and arrange the continuation of the evaluation process. On completing the evaluation of the postfix sequence for a supercombinator with n arguments, the stack will have the following form:

- On top there will be the address in heap of the newly instantiated body, `e`.
- Next there are the $n + 1$ pointers. From these we can access the arguments used in the instantiation process.
- The last of the $n+1$ pointers points to the root of the expression we have just instantiated.

This is shown in Figure 3.2.

We must replace the redex with the newly instantiated body, and pop off n items from the stack, using the `Slide` instruction. To find the next supercombinator we must now start unwinding again, using the `Unwind` instruction. By adding operations to do the tidying and unwinding to the postfix operator sequence, we have transformed the template instantiator into our Mark 1 G-machine.

The code for the function `f x1 ... xn = e` is:

```
<code to construct an instance of e>
Slide n+1
Unwind
```

3.3 Mark 1: A minimal G-machine

We now present the code for a complete G-machine and its compiler. It does not perform updates (which are introduced in Section 3.4) or arithmetic (which is introduced in Section 3.6).

3.3.1 Overall structure

At the top level the G-machine is very similar to the template instantiator; as usual the whole system is knitted together with a `run` function.

```
> -- The function run is already defined in gofers standard.prelude
> runProg :: [Char] -> [Char]
> runProg = showResults . eval . compile . parse
```

The parser data structures and functions are included because we will need access to them.

```
> -- :a language.lhs -- parser data types
```

3.3.2 Data type definitions

Fundamental to graph reduction implementation techniques is the graph. We use the `heap` data type, amongst others, from the utilities provided in Appendix A.

```
> -- :a util.lhs -- heap data type and other library functions
```

The Mark 1 G-machine uses the five-tuple, `gmState`, as its state. A `gmState` holds all the information that we need during the execution of the compiled program.

```
> type GmState
>  = (GmCode,       -- Current instruction stream
>   GmStack,        -- Current stack
>   GmHeap,         -- Heap of nodes
>   GmGlobals,      -- Global addresses in heap
>   GmStats)        -- Statistics
```

In describing the G-machine, we will make use of state *access functions* to access the components of a state. The advantage of this approach is that when we modify the state to accommodate new components, we may reuse most of the original code we have written. We will use the prefix `get` to denote an access function that gets a component from a state, and the prefix `put` to replace a component in a state.

We consider the type definitions of each of the five components of the state, and their access functions, in turn.

- The instruction stream is of type `gmCode` and is simply a list of `instructions`.

```
> type GmCode = [Instruction]
```

To get convenient access to the code, when the state is later augmented with extra components, we define two functions: `getCode` and `putCode`.

```
> getCode :: GmState -> GmCode
> getCode (i, stack, heap, globals, stats) = i


> putCode :: GmCode -> GmState -> GmState
> putCode i' (i, stack, heap, globals, stats)
>    = (i', stack, heap, globals, stats)
```

There are only six instructions initially. We will describe these in more detail in subsection 3.3.3.

```
> data Instruction
>     = Unwind
>     | Pushglobal Name
>     | Pushint Int
>     | Push Int
>     | Mkap
>     | Slide Int
> instance Eq Instruction
>     where
>     Unwind          == Unwind              = True
>     Pushglobal a    == Pushglobal b        = a == b
>     Pushint a       == Pushint b           = a == b
>     Push a          == Push b              = a == b
>     Mkap            == Mkap                = True
>     Slide a         == Slide b             = a == b
>     _               == _                   = False
```

- The G-machine stack `gmStack` is a list of addresses in the heap.

```
> type GmStack = [Addr]
```

To get convenient access to the stack, when the state is later augmented with extra components, we define two functions `getStack` and `putStack`

```
> getStack :: GmState -> GmStack
> getStack (i, stack, heap, globals, stats) = stack


> putStack :: GmStack -> GmState -> GmState
> putStack stack' (i, stack, heap, globals, stats)
>    = (i, stack', heap, globals, stats)
```

- Just as we did in the case of the template instantiator, we use the heap data structure from `utils` to implement heaps.

```
> type GmHeap = Heap Node
```

Again, to access this component of the state we define access functions.

```
> getHeap :: GmState -> GmHeap
> getHeap (i, stack, heap, globals, stats) = heap
```

```
> putHeap :: GmHeap -> GmState -> GmState
> putHeap heap' (i, stack, heap, globals, stats)
>    = (i, stack, heap', globals, stats)
```

In the minimal G-machine there are only three types of nodes: numbers, `NNum`; applications, `NAp`; and globals, `NGlobal`.

```
> data Node
>    = NNum Int              -- Numbers
>    | NAp Addr Addr         -- Applications
>    | NGlobal Int GmCode    -- Globals
```

Number nodes contain the relevant number; application nodes apply the function at the first address to the expression at the second address. The `NGlobal` node contains the number of arguments that the global expects and the code sequence to be executed when the global has enough arguments. This replaces the `NSupercomb` nodes of the template instantiator, which held a template instead of the arity and code.

- Because we will later be making a lazy implementation it is important that there is only one node for each global. The address of a global can be determined by looking up its value in the association list `gmGlobals`. This corresponds to the `tiGlobals` component of the template machine.

```
> type GmGlobals = ASSOC Name Addr
```

The access function we use is `getGlobals`; in the Mark 1 machine, this component is constant so we do not need a corresponding put function.

```
> getGlobals :: GmState -> GmGlobals
> getGlobals (i, stack, heap, globals, stats) = globals
```

- The statistics component of the state is implemented as an abstract data type.

```
> statInitial  :: GmStats
> statIncSteps :: GmStats -> GmStats
> statGetSteps :: GmStats -> Int
```

The implementation of `gmStats` is now given.

```
> type GmStats = Int
> statInitial    = 0
> statIncSteps s = s+1
> statGetSteps s = s
```

To access this component we define `getStats` and `putStats`:

```
> getStats :: GmState -> GmStats
> getStats (i, stack, heap, globals, stats) = stats


> putStats :: GmStats -> GmState -> GmState
> putStats stats' (i, stack, heap, globals, stats)
>    = (i, stack, heap, globals, stats')
```

3.3.3 The evaluator

The G-machine evaluator, `eval`, is defined to produce a list of states. The first one is the one constructed by the compiler. If there is a last state, then the result of the evaluation will be on the top of the stack component of the last state.

```
> eval :: GmState -> [GmState]
> eval state = state: restStates
>              where
>              restStates | gmFinal state     = []
>                         | otherwise         = eval nextState
>              nextState  = doAdmin (step state)
```

The function `doAdmin` uses `statIncSteps` to modify the statistics component of the state.

```
> doAdmin :: GmState -> GmState
> doAdmin s = putStats (statIncSteps (getStats s)) s
```

The important parts of the evaluator are the functions `gmFinal` and `step` which we will now look at.

Testing for a final state

The G-machine interpreter has finished when the code sequence that it is executing is empty. We express this condition in the `gmFinal` function.

```
> gmFinal :: GmState -> Bool
> gmFinal s = case (getCode s) of
>                   []        -> True
>                   otherwise -> False
```

Taking a step

The `step` function is defined so that it makes a state transition based on the instruction it is executing.

```
> step :: GmState -> GmState
> step state = dispatch i (putCode is state)
>              where (i:is) = getCode state
```

We `dispatch` on the current instruction `i` and replace the current code sequence with the code sequence `is`; this corresponds to advancing the program counter in a real machine.

```
> dispatch :: Instruction -> GmState -> GmState
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
> dispatch (Push n)       = push n
> dispatch (Slide n)      = slide n
> dispatch Unwind         = unwind
```

As we can see, the `dispatch` function simply selects a state transition to execute.

Let us begin by looking at the transition rules for the postfix instructions. There will be one for each syntactic object in `instruction`. We begin with the `Pushglobal` instruction, which uses the `globals` component of the state to find the unique `NGlobal` node in the `heap` that holds the global f. If it cannot find one, it prints a suitable error message.

$$(3.5) \quad \boxed{\begin{array}{lrrll} & \texttt{Pushglobal}\ f : i & s & h & m[f : a] \\ \Longrightarrow & i & a : s & h & m \end{array}}$$

We implement this rule using the `pushglobal` function.

```
> pushglobal :: Name -> GmState -> GmState
> pushglobal f state
>       = putStack (a: getStack state) state
>       where a = aLookup (getGlobals state) f (error ("Undeclared global " ++ f))
```

The remaining transitions are for constructing the body of a supercombinator. The transition for `Pushint` places an integer node into the heap.

$$(3.6) \quad \boxed{\begin{array}{lrrll} & \texttt{Pushint}\ n : i & s & h & m \\ \Longrightarrow & i & a : s & h[a : \texttt{NNum}\ n] & m \end{array}}$$

The corresponding function is `pushint`. The number is placed in the new heap `heap'` with address `a`. We then place the heap and stack back into the state.

```
> pushint :: Int -> GmState -> GmState
> pushint n state
>       = putHeap heap' (putStack (a: getStack state) state)
>       where (heap', a) = hAlloc (getHeap state) (NNum n)
```

The `Mkap` instruction uses the two addresses on the top of the stack to construct an application node in the heap. It has the following transition rule.

$$(3.7) \qquad \boxed{\begin{array}{lrrlr} & \texttt{Mkap} : i & a_1 : a_2 : s & h & m \\ \Longrightarrow & i & a : s & h[a : \texttt{NAp}\ a_1\ a_2] & m \end{array}}$$

This transition becomes `mkap`. Again `heap'` and `a` are respectively the new heap and the address of the new node.

```
> mkap :: GmState -> GmState
> mkap state
>       = putHeap heap' (putStack (a:as') state)
>       where (heap', a)  = hAlloc (getHeap state) (NAp a1 a2)
>             (a1:a2:as') = getStack state
```

The `Push` instruction is used to take a copy of an argument which was passed to a function. To do this it has to 'look through' the application node which is pointed to from the stack. We must also remember to skip over the supercombinator node which is on the stack.

$$(3.8) \qquad \boxed{\begin{array}{lrrlr} & \texttt{Push}\ n : i & a_0 : \ldots : a_{n+1} : s & h[a_{n+1} : \texttt{NAp}\ a_n\ a'_n] & m \\ \Longrightarrow & i & a'_n : a_0 : \ldots : a_{n+1} : s & h & m \end{array}}$$

```
> push :: Int -> GmState -> GmState
> push n state
>    = putStack (a:as) state
>    where  as = getStack state
>           a  = getArg (hLookup (getHeap state) (as !! (n+1)))
```

This uses the auxiliary function `getArg` to select the required expression from an application node.

```
> getArg :: Node -> Addr
> getArg (NAp a1 a2) = a2
```

> *Because of the stack structure we have changed the addressing mode of the* `Push` *instruction from that used in [Peyton Jones 1987].*

Next, the tidying up of the stack, which occurs after a supercombinator has been instantiated and before continuing unwinding, is performed by the `Slide` instruction.

$$(3.9)\quad \begin{array}{|lrrrr|} \hline & \texttt{Slide}\ n:i & a_0:\ldots:a_n:s & h & m \\ \Longrightarrow & i & a_0:s & h & m \\ \hline \end{array}$$

```
> slide :: Int -> GmState -> GmState
> slide n state
>       = putStack (a: drop n as) state
>       where (a:as) = getStack state
```

Unwind is the most complex instruction because it replaces the outer loop of our template instantiator. The Unwind instruction is always the last instruction of a sequence, as we shall see in the next section. The newState constructed depends on the item on top of the stack; this depends on the transition rule that is fired, which also depends on the item on top of the stack.

```
> unwind :: GmState -> GmState
> unwind state
>      = newState (hLookup heap a)
>      where
>              (a:as) = getStack state
>              heap   = getHeap state
```

We first consider the case where there is a number on top of the stack. In this case, we are finished; the G-machine has terminated, and we place [] in the code component to signify this fact.

$$(3.10)\quad \begin{array}{|lrlll|} \hline & [\texttt{Unwind}] & a:s & h[a:\texttt{NNum}\ n] & m \\ \Longrightarrow & [] & a:s & h & m \\ \hline \end{array}$$

```
>              newState (NNum n)      = state
```

If there is an application node on top of the stack then we must continue to unwind from the next node.

$$(3.11)\quad \begin{array}{|llrll|} \hline & [\texttt{Unwind}] & a:s & h[a:\texttt{NAp}\ a_1\ a_2] & m \\ \Longrightarrow & [\texttt{Unwind}] & a_1:a:s & h & m \\ \hline \end{array}$$

```
>              newState (NAp a1 a2)   = putCode [Unwind] (putStack (a1:a:as) state)
```

The most complicated rule occurs when there is a global node on top of the stack. There are two cases to consider, depending on whether there are enough arguments to reduce the supercombinator application.

Firstly, if there are not enough arguments to reduce the supercombinator application then the program was ill-typed. We will ignore this case for the Mark 1 G-machine. Alternatively, when there are enough arguments, it is possible to reduce the supercombinator, by 'jumping to' the code for the supercombinator. In the transition rule this is expressed by moving the supercombinator code into the code component of the machine.

$\mathcal{SC}[\![d]\!]$ is the G-machine code for the supercombinator definition d.

$$\mathcal{SC}[\![f\ x_1\ \ldots\ x_n\ =\ e]\!]\ =\ \mathcal{R}[\![e]\!]\ [x_1 \mapsto 0, \ldots, x_n \mapsto n-1]\ n$$

$\mathcal{R}[\![e]\!]\ \rho\ d$ generates code which instantiates the expression e in environment ρ, for a supercombinator of arity d, and then proceeds to unwind the resulting stack.

$$\mathcal{R}[\![e]\!]\ \rho\ d\ =\ \mathcal{C}[\![e]\!]\ \rho \mathbin{+\!\!+} [\texttt{Slide}\ d+1, \texttt{Unwind}]$$

$\mathcal{C}[\![e]\!]\ \rho$ generates code which constructs the graph of e in environment ρ, leaving a pointer to it on top of the stack.

$$\begin{array}{rcll}
\mathcal{C}[\![f]\!]\ \rho & = & [\texttt{Pushglobal}\ f] & \text{where } f \text{ is a supercombinator} \\
\mathcal{C}[\![x]\!]\ \rho & = & [\texttt{Push}\ (\rho\ x)] & \text{where } x \text{ is a local variable} \\
\mathcal{C}[\![i]\!]\ \rho & = & [\texttt{Pushint}\ i] & \\
\mathcal{C}[\![e_0\ e_1]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{C}[\![e_0]\!]\ \rho^{+1} \mathbin{+\!\!+} [\texttt{Mkap}] & \text{where } \rho^{+n}\ x = (\rho\ x) + n
\end{array}$$

Figure 3.3: The $\mathcal{SC}$, $\mathcal{R}$ and $\mathcal{C}$ compilation schemes

$$(3.12)\qquad \begin{array}{|lrlll|}
\hline
 & [\texttt{Unwind}] & a_0 : \ldots : a_n : s & h[a_0 : \texttt{NGlobal}\ n\ c] & m \\
\Longrightarrow & c & a_0 : \ldots : a_n : s & h & m \\
\hline
\end{array}$$

```
>              newState (NGlobal n c)
>                    | length as < n        = error "Unwinding with too few arguments"
>                    | otherwise    = putCode c state
```

We have now seen how the instructions are defined, but we have not seen how to generate the postfix sequences of operators, or instruction sequences as we shall refer to them from now on. This is the subject of the next subsection.

3.3.4 Compiling a program

We describe the compiler using a set of *compilation schemes*. Each supercombinator definition is compiled using the compilation scheme $\mathcal{SC}$. The compiled code generated for each supercombinator is defined in Figure 3.3. Corresponding to the compilation schemes $\mathcal{SC}$, $\mathcal{R}$ and $\mathcal{C}$ are *compiler functions* `compileSc`, `compileR` and `compileC`. We consider each of these in turn.

The `compile` function turns a program into an initial state for the G-machine. The initial code sequence finds the global `main` and then evaluates it. The heap is initialised so that it contains a node for each global declared. `globals` contains the map from global names to the `NGlobal` nodes provided for them.

```
> compile :: CoreProgram -> GmState
```

```
> compile program
>    = (initialCode, [], heap, globals, statInitial)
>    where (heap, globals) = buildInitialHeap program
```

To construct the initial heap and to provide the map of the global nodes for each global defined we use `buildInitialHeap`. This is just as it was in the template machine.

```
> buildInitialHeap :: CoreProgram -> (GmHeap, GmGlobals)
> buildInitialHeap program
>    = mapAccuml allocateSc hInitial compiled
>    --where compiled = map compileSc (preludeDefs ++ program) ++
>    --                 compiledPrimitives
>    where compiled = map compileSc program
```

The `buildInitialHeap` function uses `mapAccuml` to allocate nodes for each compiled global; the compilation occurring (where necessary) in `compiled`, which has type `[gmCompiledSC]`.

```
> type GmCompiledSC = (Name, Int, GmCode)
```

The function `allocateSc` allocates a new global for its compiled supercombinator argument, returning the new heap and the address where the global is stored.

```
> allocateSc :: GmHeap -> GmCompiledSC -> (GmHeap, (Name, Addr))
> allocateSc heap (name, nargs, instns)
>       = (heap', (name, addr))
>       where (heap', addr) = hAlloc heap (NGlobal nargs instns)
```

In the initial state, we want the machine to evaluate the value of the program. We recall that this is just the value of the global `main`.

```
> initialCode :: GmCode
> initialCode = [Pushglobal "main", Unwind]
```

Each supercombinator is compiled using `compileSc`, which implements the $\mathcal{SC}$ scheme of Figure 3.3. It returns a triple containing the supercombinator name, the number of arguments the supercombinator needs before it can be reduced, and the code sequence associated with the supercombinator.

```
> compileSc :: (Name, [Name], CoreExpr) -> GmCompiledSC
> compileSc (name, env, body)
>       = (name, length env, compileR body (zip2 env [0..]))
```

This in turn uses `compileR`, which corresponds to the $\mathcal{R}$ scheme of Figure 3.3.

```
> compileR :: GmCompiler
> compileR e env = compileC e env ++ [Slide (length env + 1), Unwind]
```

```
> compileR e env = compileC e env ++ [Slide (length env + 1), Unwind]
```

Each of the compiler schemes has the same type: gmCompiler.

```
> type GmCompiler = CoreExpr -> GmEnvironment -> GmCode
```

We use the fact that we can represent the map ρ from the compilation scheme as an association list. Not only can we look up the offsets for a variable from this list, but we may also calculate how many arguments there are on the stack. This is used in compileR to find out how many stack elements to squeeze out with a Slide instruction. The list has type gmEnvironment, which is defined as:

```
> type GmEnvironment = ASSOC Name Int
```

This constructs the instantiation of the supercombinator body using compileC, which corresponds to the $\mathcal{C}$ scheme of Figure 3.3.

```
> compileC :: GmCompiler
> compileC (EVar v)    env
>  | elem v (aDomain env)          = [Push n]
>  | otherwise                     = [Pushglobal v]
>  where n = aLookup env v (error "Can't happen")
> compileC (ENum n)    env = [Pushint n]
> compileC (EAp e1 e2) env = compileC e2 env ++
>                            compileC e1 (argOffset 1 env) ++
>                            [Mkap]
```

We can change the stack offsets using the function argOffset. If env implements ρ, then (argOffset n env) implements ρ^{+n}.

```
> argOffset :: Int -> GmEnvironment -> GmEnvironment
> argOffset n env = [(v, n+m) | (v,m) <- env]
```

An example compilation

Let us look at the compilation of the K combinator. When compiling this function, we will begin by evaluating the following expression.

```
  compileSc ("K", ["x", "y"], EVar "x")
```

The first element of the tuple is the name (K in this case); the second is the argument list (in this case we have two variables: x and y); and the third component of the tuple is the body of the supercombinator (which for this example is just the variable x).

When we rewrite this expression, we get:

```
  ("K", 2, compileR (EVar "x") [("x", 0), ("y", 1)])
```

The resulting triple consists of the name (K), the number of arguments we need to reduce the supercombinator (two in this case), and the code sequence to perform the instantiation. When we rewrite this expression we will generate the code sequence for this supercombinator. Notice that the environment is represented by the expression `[("x", 0), ("y", 1)]`; this tells us that when we instantiate the body, a pointer to `x` will be on top of the argument stack and a pointer to `y` will be immediately below `x` on the stack.

```
("K", 2, compileC (EVar "x") [("x", 0), ("y", 1)] ++ [Slide 3, Unwind])
```

The `compileR` function is defined to compile the body using `compileC`, and to add a `Slide` and an `Unwind` instruction at the end.

To compile the body we look up `x` and find that it is on top of the stack. We generate code to make a copy of the top of the stack, using `Push 0`.

```
("K", 2, [Push 0, Slide 3, Unwind])
```

Exercise 3.3. Write out the equivalent sequence of transformations for the `S` combinator from the prelude definitions. Recall that `S` is defined as:

```
S f g x = f x (g x)
```

Check the final result by running the compiler and machine with any of the simple programs given in Appendix B. (`S` is in the standard prelude.)

Primitives

In this minimal G-machine there are no primitives, so there is nothing to implement!

```
> compiledPrimitives :: [GmCompiledSC]
> compiledPrimitives = []
```

3.3.5 Printing the results

Because a number of the state components are abstract data types (and are therefore not directly printable) we must define a pretty-printer for the states that the machine produces. It is also a fact that the output is voluminous and not very informative if it is all displayed at once. The printing is controlled by `showResults`. It produces three pieces of output: the super-combinator code sequences, the state transitions and the final statistics.

```
> showResults :: [GmState] -> [Char]
> showResults states
>       = iDisplay (iConcat [
>       iStr "Supercombinator definitions", iNewline,
>       iInterleave iNewline (map (showSC s) (getGlobals s)),
>       iNewline, iNewline, iStr "State transitions", iNewline, iNewline,
>       iLayn (map showState states),
```

```
>       iNewline, iNewline,
>       showStats (last states)])
>       where (s:ss) = states
```

Taking each of these in turn, we begin with `showSC`. This finds the code for the supercombinator in the unique global heap node associated with the global, and prints the code sequence using `showInstructions`.

```
> showSC :: GmState -> (Name, Addr) -> Iseq
> showSC s (name, addr)
>       = iConcat [ iStr "Code for ", iStr name, iNewline,
>             showInstructions code, iNewline, iNewline]
>       where (NGlobal arity code) = (hLookup (getHeap s) addr)
```

Then `showInstructions` is used to output a code sequence.

```
> showInstructions :: GmCode -> Iseq
> showInstructions is
>       = iConcat [iStr "  Code:{",
>            iIndent (iInterleave iNewline (map showInstruction is)),
>            iStr "}", iNewline]
```

The output for each individual instruction is given by `showInstruction`.

```
> showInstruction :: Instruction -> Iseq
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = (iStr "Pushglobal ") `iAppend` (iStr f)
> showInstruction (Push n)       = (iStr "Push ")       `iAppend` (iNum n)
> showInstruction (Pushint n)    = (iStr "Pushint ")    `iAppend` (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
> showInstruction (Slide n)      = (iStr "Slide ")      `iAppend` (iNum n)
```

The next major piece of output is the state transitions; these are individually dealt with using `showState`.

```
> showState :: GmState -> Iseq
> showState s
>    = iConcat [showStack s,         iNewline,
>    showInstructions (getCode s),   iNewline]
```

To correspond with our diagrams, we would like to have the top of stack at the bottom of the printed stack. To this end we reverse the stack.

```
> showStack :: GmState -> Iseq
> showStack s
>       = iConcat [iStr " Stack:[",
>            iIndent (iInterleave iNewline
>                        (map (showStackItem s) (reverse (getStack s)))),
>            iStr "]"]
```

Each stack item is displayed using showStackItem. It prints the address stored in the stack and the object in the heap to which it points.

```
> showStackItem :: GmState -> Addr -> Iseq
> showStackItem s a
>       = iConcat [iStr (showaddr a), iStr ": ",
>            showNode s a (hLookup (getHeap s) a)]
```

The function showNode needs to invert the association list of global names and heap addresses to display the global nodes it comes across.

```
> showNode :: GmState -> Addr -> Node -> Iseq
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>    where v = head [n | (n,b) <- getGlobals s, a==b]
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ", iStr (showaddr a1),
>                                        iStr " ",   iStr (showaddr a2)]
```

Finally, we print the accumulated statistics, using showStats.

```
> showStats :: GmState -> Iseq
> showStats s
>       = iConcat [ iStr "Steps taken = ", iNum (statGetSteps (getStats s))]
```

This concludes the description of the basic G-machine. We now move on to consider ways to make it more sophisticated.

3.3.6 Improvements to the Mark 1 G-machine

Exercise 3.4. Run the program `main = S K K 3`. How many steps does it take? Why is it different from that obtained for the template machine? Do you think that comparing steps taken is a fair comparison of the machines?

Exercise 3.5. Try running some other programs from Appendix B. Remember, there is no arithmetic in this simple machine.

Exercise 3.6. It is possible to use the same trick we used for Pushglobal to implement Pushint: for each distinct number we create a unique node in the heap. For example, when we first execute `Pushint 2`, we update gmGlobals so that it associates "2" with the address in heap of the node `NNum 2`.

In the transition rules, if there is already a global called n, then we can reuse this global node.

$$(3.13) \quad \boxed{\begin{array}{llllll} & \texttt{Pushint}\ n : i & s & h & m[n:a] \\ \Longrightarrow & i & a:s & h & m \end{array}}$$

Alternatively, when this is not the case, we will create a new node and add it to the global map.

$$(3.14) \quad \boxed{\begin{array}{llllll} & \texttt{Pushint}\ n : i & s & h & m \\ \Longrightarrow & i & a:s & h[a:\texttt{NNum}\ n] & m[n:a] \end{array}}$$

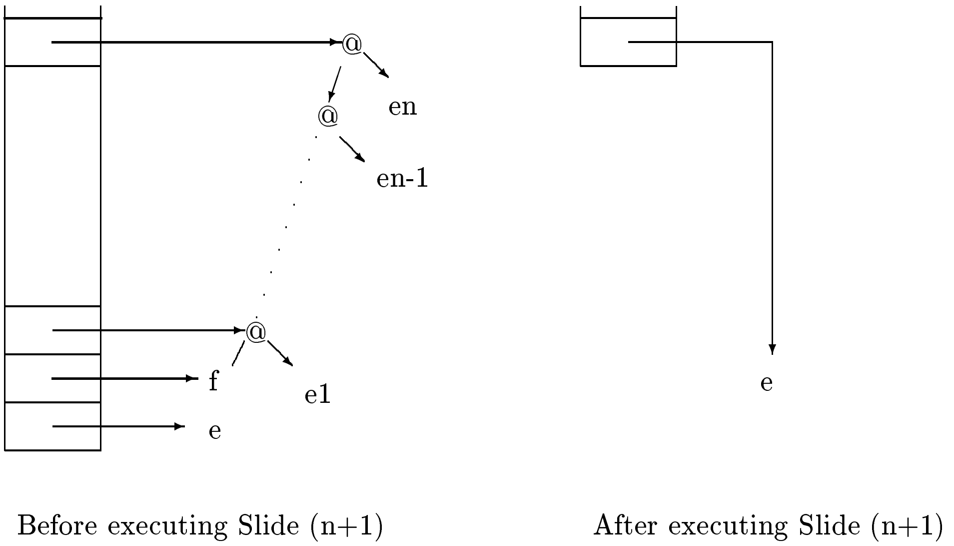


Figure 3.4: Mark 1 G-machine (executing `Slide n+1`)

> The advantage of this scheme is that we can reuse the same number node in the heap each time a `Pushint` is executed.
>
> Implement this new transition `pushint` for the `Pushint` instruction. You should define an access function for the global component, calling it `putGlobals`.

3.4 Mark 2: Making it lazy

We will now make a number of small changes to the Mark 1 G-machine in order to make it lazy. The Mark 1 machine is not lazy at the moment because it does not overwrite the root node of the original expression before unwinding. This updating is described in Section 2.1.5. In the Mark 2 machine, the idea is that after instantiating the body of the supercombinator, we will *overwrite* the root of the original redex with an *indirection node* pointing to the newly constructed instance. The effect is that the machine 'remembers' the value that was instantiated last time the redex was reduced, and hence does not need to recalculate it.

We implement this change as follows. In the Mark 1 machine the code for each supercombinator concluded with [`Slide` $(n + 1)$, `Unwind`]. To capture updating we replace this with [`Update` n, `Pop` n, `Unwind`]. This is illustrated in the following diagrams, in which we use `#` to represent indirection nodes.

Figure 3.4 shows how the Mark 1 machine executes a `Slide` $n + 1$ instruction. In Figure 3.5 we see the Mark 2 machine executing the sequence [`Update` n, `Pop` n]; this being the sequence we propose to use as a lazy replacement for [`Slide` $n + 1$]. The `Update` instruction is responsible for overwriting the root node with the newly created instance of the body of the supercombinator. The `Pop` instruction is used to remove the arguments from the stack, as they are now no longer needed.

Let us first consider the necessary modifications to the data structures.

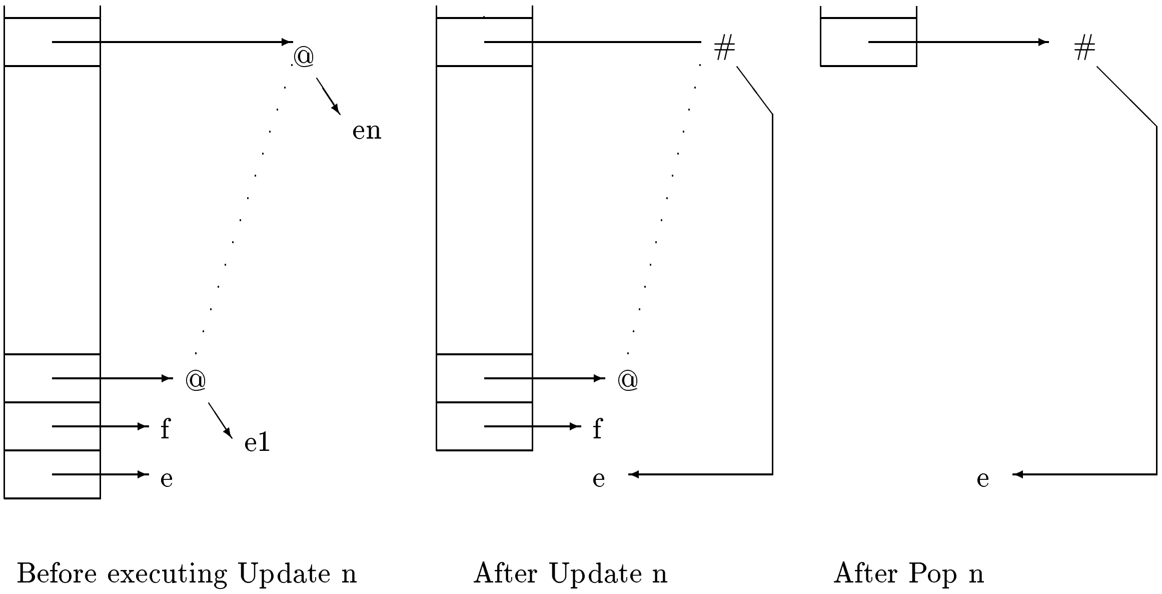


Figure 3.5: Mark 2 G-machine (executing `[Update n, Pop n]`)

3.4.1 Data structures

In place of the single instruction `Slide` $n+1$ that we generated last time we now generate the sequence of instructions [`Update` n, `Pop` n]. Therefore we are going to have to include these instructions in the new instruction set.

```
> data Instruction = Unwind
>                  | Pushglobal Name
>                  | Pushint Int
>                  | Push Int
>                  | Mkap
>                  | Update Int
>                  | Pop Int
> instance Eq Instruction
>    where
>    Unwind          == Unwind                = True
>    Pushglobal a    == Pushglobal b          = a == b
>    Pushint a       == Pushint b             = a == b
>    Push a          == Push b                = a == b
>    Mkap            == Mkap                  = True
>    Update a        == Update b              = a == b
>    _               == _                     = False
```

Exercise 3.7. Modify the function `showInstruction`, so that it displays the new instructions.

To implement the indirection nodes we must have a new node type in the heap: `NInd` which we use for indirections.

```
> data Node
>   = NNum Int                -- Numbers
>   | NAp Addr Addr           -- Applications
>   | NGlobal Int GmCode      -- Globals
>   | NInd Addr               -- Indirections
> instance Eq Node
>   where
>   NNum a       == NNum b         = a == b    -- needed to check conditions
>   NAp a b      == NAp c d        = False     -- not needed
>   NGlobal a b  == NGlobal c d    = False     -- not needed
>   NInd a       == NInd b         = False     -- not needed
```

Again we must redefine the display function `showNode`, so that it reflects the extension of the data type.

Exercise 3.8. Make the necessary change to `showNode`.

We have not yet given a semantics to the two new instructions. This is done below.

3.4.2 The evaluator

The effect of an `Update` n instruction is to overwrite the $n+1^{\text{th}}$ stack item with an indirection to the item on top of the stack. Notice that this addressing mode is different from that used in [Peyton Jones 1987]. For the intended application of this instruction the $a_1 \dots a_n$ are the n application nodes forming the spine, and a_0 is the function node.

$$(3.15)\quad \boxed{\begin{array}{lrrll} & \texttt{Update}\ n : i & a : a_0 : \dots : a_n : s & h & m \\ \Longrightarrow & i & a_0 : \dots : a_n : s & h[a_n : \texttt{NInd}\ a] & m \end{array}}$$

The `Pop` n instruction simply removes n stack items. Again, in the Mark 2 G-machine $a_1 \dots a_n$ are the application nodes forming the spine of the redex.

$$(3.16)\quad \boxed{\begin{array}{lrrll} & \texttt{Pop}\ n : i & a_1 : \dots : a_n : s & h & m \\ \Longrightarrow & i & s & h & m \end{array}}$$

We must also define a transition for `Unwind` when the top of stack item is an indirection. The effect is to replace the current stack item with the item that the indirection points to.

$$(3.17)\quad \boxed{\begin{array}{llrll} & [\texttt{Unwind}] & a_0 : s & h[a_0 : \texttt{NInd}\ a] & m \\ \Longrightarrow & [\texttt{Unwind}] & a : s & h & m \end{array}}$$

Exercise 3.9. Modify the `dispatch` function of the Mark 1 machine to incorporate the new instructions; implement the new transition rules.

$\mathcal{R}[\![e]\!]\ \rho\ d$ generates code which instantiates the expression e in environment ρ, for a supercombinator of arity d, and then proceeds to unwind the resulting stack.

$$\mathcal{R}[\![e]\!]\ \rho\ d \quad = \quad \mathcal{C}[\![e]\!]\ \rho \mathbin{+\!\!+} [\texttt{Update}\ d, \texttt{Pop}\ d, \texttt{Unwind}]$$

Figure 3.6: The $\mathcal{R}$ compilation scheme for Mark 2 G-machine

3.4.3 The compiler

The only change to the compiler lies in the code generated by the $\mathcal{R}$ scheme. The new definition is given in Figure 3.6.

Exercise 3.10. Modify `compileR` to implement the new $\mathcal{R}$ scheme.

Exercise 3.11. Run the lazy evaluator on the program:

```
twice f x = f (f x)
id x = x
main = twice twice id 3
```

How many steps does it take? Why is it different from that obtained for the Mark 1 machine? Is it fair to compare the number of steps taken for the machines?

3.5 Mark 3: `let(rec)` expressions

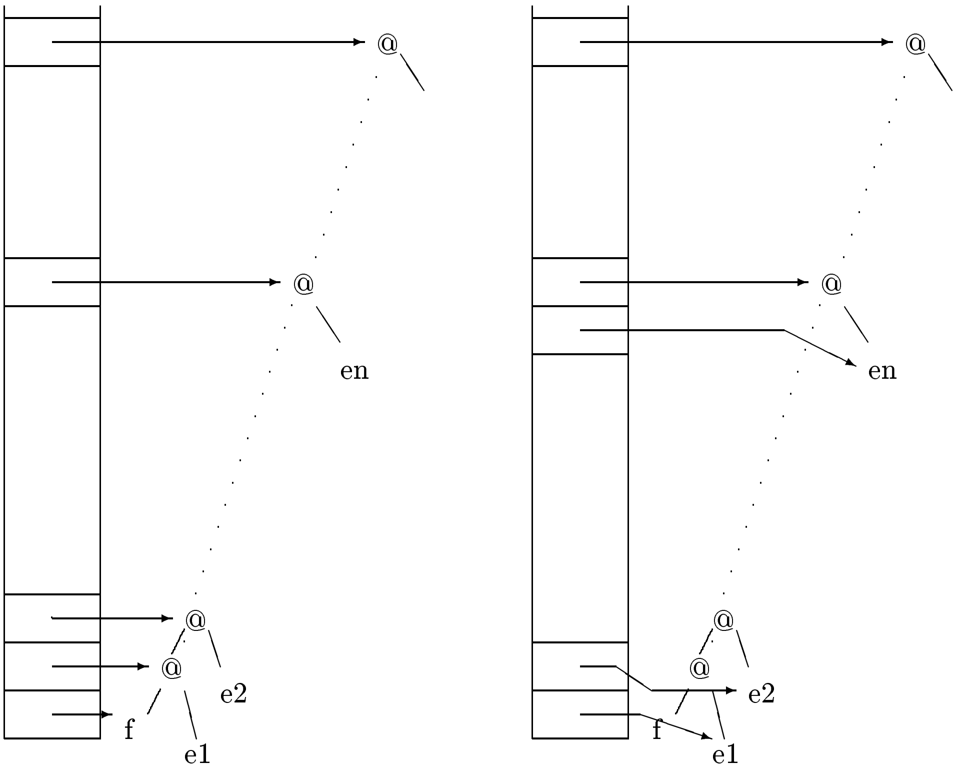
We now extend the language so that the compiler will accept supercombinators whose body includes `let(rec)`-*bound variables*. These are represented in the data type `coreExpr` by the constructor `ELet`. It takes three arguments: a boolean flag which says whether the definitions are to be treated recursively, the definitions themselves and an expression in which the definitions are to be used.

Before we attempt to extend the machine by adding local definitions, we will have another look at the stack. In particular we will try to define a more efficient *access method* for variables. Besides the efficiency argument, we also wish to make access to locally bound variables the same as that used to bind function parameters.

Argument access from the stack

Suppose that the unwinding process has reached a supercombinator node `f`, and that the supercombinator takes n arguments. In the Mark 1 machine the stack will be in the state shown in the left-hand diagram of Figure 3.7.

Having reached the supercombinator `f`, in the Mark 3 G-machine, the stack is slightly modified. The equivalent Mark 3 stack is shown in the right-hand diagram of Figure 3.7; the top n elements now point directly to the expressions `e1` ... `en`. The important point here is that we have faster



Marks 1 and 2 Mark 3 onwards

Figure 3.7: Stack layout on entry to function `f`

access to the variables (provided that the variable is accessed at least once). This is because we only look at the application node once, to get its right-hand argument.

This improves the efficiency of access to the expressions that will be substituted for the formal parameters in the supercombinator. In terms of the Mark 1 machine:

- we no longer need the function `getArg` in the `Push` instruction,
- but we do need to rearrange the stack when we `Unwind` a supercombinator with sufficient arguments.

Notice that we have retained a pointer to the root of the redex so that we can perform an `Update`.

The effects on instructions

When we choose to use the new stack layout, we necessarily have to modify certain of the machine instructions to cope. The instructions affected are `Push` and `Unwind`. The `Push` instruction will have to change because we do not need to 'look through' the application node to get at the argument.

$$(3.18)\qquad \boxed{\begin{array}{lrrcc} & \texttt{Push}\ n : i & a_0 : \ldots : a_n : s & h & m \\ \Longrightarrow & i & a_n : a_0 : \ldots : a_n : s & h & m \end{array}}$$

The other modification required for the new stack layout is that `Unwind` must rearrange the stack. This *rearrangement* is required whenever a supercombinator with sufficient arguments is found on the top of the stack. The new transition rule for `Unwind` is:

$$(3.19)\qquad \boxed{\begin{array}{lrrcc} & [\texttt{Unwind}] & a_0 : \ldots : a_n : s & h\left[\begin{array}{lcl} a_0 & : & \texttt{NGlobal}\ n\ c \\ a_1 & : & \texttt{NAp}\ a_0\ a'_1 \\ & \cdots & \\ a_n & : & \texttt{NAp}\ a_{n-1}\ a'_n \end{array}\right] & m \\ \Longrightarrow & c & a'_1 : \ldots : a'_n : a_n : s & h & m \end{array}}$$

Notice that this definition of `Unwind` will work properly for the case where n is zero.

Exercise 3.12. Rewrite the `dispatch` function and the new transitions for the new instruction set. You should make use of the function `rearrange` to rearrange the stack.

```
> rearrange :: Int -> GmHeap -> GmStack -> GmStack
> rearrange n heap as
>       = take n as' ++ drop n as
>       where as' = map (getArg . hLookup heap) (tl as)
```

Exercise 3.13. Test the compiler and new abstract machine on some sample programs from Appendix B, to ensure that the implementation still works.

3.5.1 Locally bound variables

Now we return to the implementation of `let(rec)` expressions, considering the non-recursive case first. The variables $x_1 \ldots x_n$, in the expression `let` x_1=e_1; ... ; $x_n = e_n$ `in` e, can be treated in the same way as the arguments to a supercombinator, once the expressions $e_1 \ldots e_n$ have been created. That is, we access the variables $x_1 \ldots x_n$ via offsets into the stack, using the environment to record their locations.

Suppose that the code to build the local definitions is called `Code`, then the sequence of actions shown in Figure 3.8 will be necessary. Initially, the stack will contain pointers to the arguments to the supercombinator. After the code to build the local definitions has executed we will have n new pointers on the stack. We can now proceed to build the body of the `let` expression, in a new environment that maps x_i to the pointer to e_i. Finally, we need to throw away the pointers to the expressions $e_1 \ldots e_n$ from the stack.

Because we have added n new variables to the stack ($x_1 \ldots x_n$) we must note this fact in the variable map we use to compile e. The code to construct the local bindings – which we have called `Code` – will simply build the graph of each expression $e_1 \ldots e_n$ in turn, leaving the address of the piece of graph on the stack.

After building the body expression e – which may use any of the variables $x_1 \ldots x_n$ – we must remove the pointers to $e_1 \ldots e_n$ from the stack. This is accomplished by using a `Slide` instruction. The complete scheme for compiling a non-recursive local definition is given in Figure 3.10 (p.114).

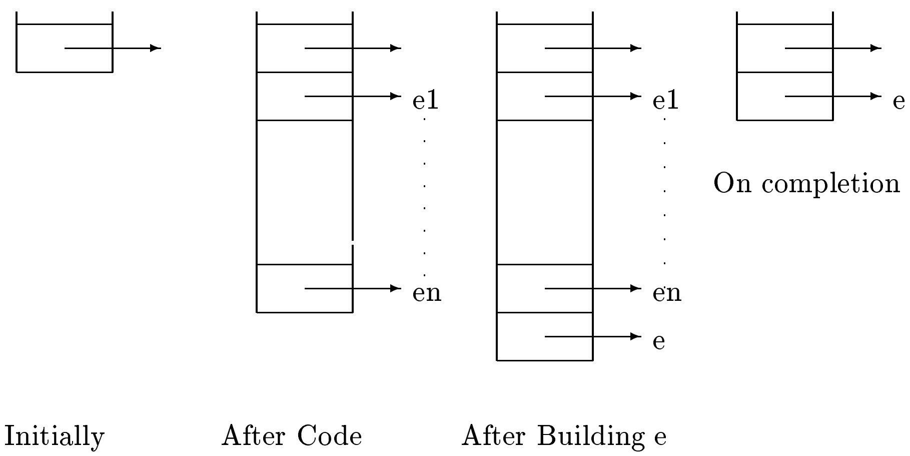


Figure 3.8: Stack usage in non-recursive local definitions

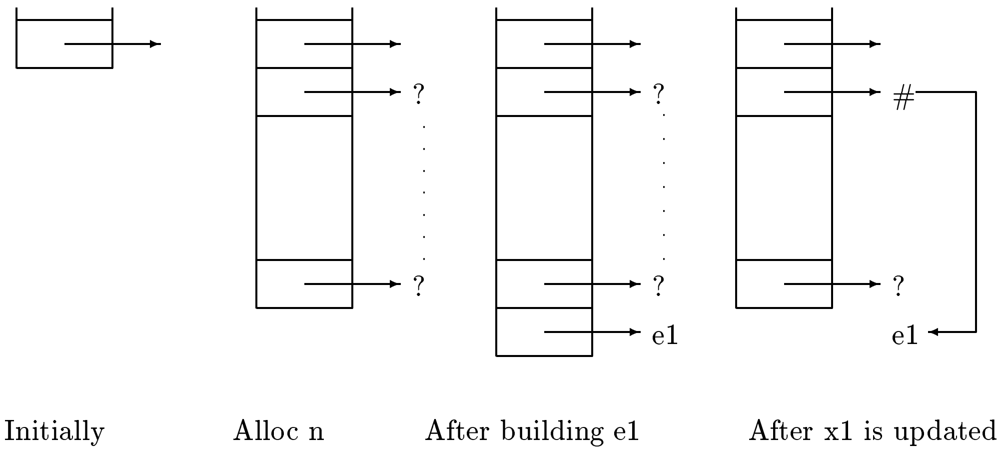


Figure 3.9: Constructing a recursively defined expression: `e1`

The situation with recursive local definitions is more complicated: each of the expressions $e_1 \ldots e_n$ must be compiled so that the variables $x_1 \ldots x_n$ are in scope. To do this we create empty nodes in the graph, leaving pointers to them on the stack. Each expression $e_1 \ldots e_n$ is then compiled using the same variable map that we used for the compilation of the body of the non-recursive case. At the end of each expression's compiled code we place an `Update` instruction that will overwrite the empty node with the correct piece of graph. To do this we need one new instruction – `Alloc n` – which will create `n` empty graph nodes for us. In Figure 3.9 the empty graph nodes are represented by a `?` symbol.

The process shown in Figure 3.9 needs to be repeated until each of the expressions $e_1 \ldots e_n$ has been processed. Compiling code for the body e is then the same as the previous case for non-recursive local definitions. We now add the new data types for the Mark 3 machine.

3.5.2 Data structures

The instruction data type includes all of the instructions of the Mark 2 machine, with the new `Alloc` instruction and the `Slide` instruction from the Mark 1 machine.

Exercise 3.14. Modify the data type `instruction` so that it includes `Alloc` and `Slide`. You will also need to modify the function `showInstruction`, to accommodate these new instructions.

3.5.3 The evaluator

For the Mark 3 G-machine we will need to add the `Alloc` instruction which creates `n` locations in the heap. We use these locations to mark the places we will store the locally bound expressions. These nodes are initially created as indirection nodes that point to an illegal heap address: `hNull`. Because these nodes created by `Alloc` are going to be overwritten, it does not really matter what value we assign them.

$$
\text{(3.20)} \quad \boxed{\begin{array}{lrrcc}
 & \texttt{Alloc}\ n : i & s & h & m \\
\Longrightarrow & i & a_1 : \ldots : a_n : s & h\left[\begin{array}{lcl} a_1 & : & \texttt{NInd hNull} \\ & \cdots & \\ a_n & : & \texttt{NInd hNull} \end{array}\right] & m
\end{array}}
$$

To implement `alloc`, the transition function for the `Alloc` instruction, we use an auxiliary function `allocNodes`. Given the number of nodes required and the current heap, it returns a pair consisting of the modified heap and the list of addresses of the indirection nodes.

```
> allocNodes :: Int -> GmHeap -> (GmHeap, [Addr])
> allocNodes 0     heap = (heap,  [])
> allocNodes (n+1) heap = (heap2, a:as)
>                         where (heap1, as) = allocNodes n heap
>                               (heap2, a)  = hAlloc heap1 (NInd hNull)
```

Exercise 3.15. Extend the `dispatch` function, with cases for the new instructions. You should use `allocNodes` to implement `alloc`, the transition function for the `Alloc` instruction.

3.5.4 The compiler

The only change to the compiler is that there are now two more cases for which the $\mathcal{C}$ scheme can compile code. The modification to `compileC` is simple. It can now cope with a wider range of `coreExprs`. We need two new functions: `compileLetrec` and `compileLet`.

```
> compileC :: GmCompiler
> compileC (EVar v)      args
>  | elem v (aDomain args) = [Push n]
>  | otherwise             = [Pushglobal v]
>         where n = aLookup args v (error "")
```

$\mathcal{C}[\![e]\!]\ \rho$ generates code which constructs the graph of e in environment ρ, leaving a pointer to it on top of the stack.

$$
\begin{array}{lcll}
\mathcal{C}[\![f]\!]\ \rho & = & [\texttt{Pushglobal}\ f] & \text{where } f \text{ is a supercombinator} \\
\mathcal{C}[\![x]\!]\ \rho & = & [\texttt{Push}\ (\rho\ x)] & \text{where } x \text{ is a local variable} \\
\mathcal{C}[\![i]\!]\ \rho & = & [\texttt{Pushint}\ i] & \\
\mathcal{C}[\![e_0\ e_1]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho +\!\!+ \mathcal{C}[\![e_0]\!]\ \rho^{+1} +\!\!+ [\texttt{Mkap}] & \text{where } \rho^{+n}\ x \text{ is } (\rho\ x) + n \\
\multicolumn{4}{l}{\mathcal{C}[\![\texttt{let}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ & \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ & \\
 & & \mathcal{C}[\![e]\!]\ \rho' +\!\!+ [\texttt{Slide}\ n] & \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\multicolumn{4}{l}{\mathcal{C}[\![\texttt{letrec}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
 & = & [\texttt{Alloc}\ n] +\!\!+ & \\
 & & \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{Update}\ n-1] +\!\!+ \ldots +\!\!+ & \\
 & & \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{Update}\ 0] +\!\!+ & \\
 & & \mathcal{C}[\![e]\!]\ \rho' +\!\!+ [\texttt{Slide}\ n] & \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0]
\end{array}
$$

Figure 3.10: The modified $\mathcal{C}$ compilation scheme for `let` and `letrec`

```
> compileC (ENum n)    env = [Pushint n]
> compileC (EAp e1 e2) env = compileC e2 env ++
>                            compileC e1 (argOffset 1 env) ++
>                            [Mkap]
> compileC (ELet recursive defs e) args
>          | recursive     = compileLetrec compileC defs e args
>          | otherwise     = compileLet    compileC defs e args
```

The definition of `compileLet` follows the specification given in Figure 3.10. It takes as arguments: the compilation scheme `comp` for the body `e`, the definitions `defs` and the current environment `env`. We have provided the compiler parameter so that in later versions of the machine we do not have to rewrite this function.

```
> compileLet :: GmCompiler -> [(Name, CoreExpr)] -> GmCompiler
> compileLet comp defs expr env
>   = compileLet' defs env ++ comp expr env' ++ [Slide (length defs)]
>     where env' = compileArgs defs env
```

The compilation of the new definitions is accomplished by the function `compileLet'`.

```
> compileLet' :: [(Name, CoreExpr)] -> GmEnvironment -> GmCode
> compileLet' []                  env = []
> compileLet' ((name, expr):defs) env
>     = compileC expr env ++ compileLet' defs (argOffset 1 env)
```

`compileLet` also uses `compileArgs` to modify the offsets into the stack for the compilation of the body, `e`.

```
> compileArgs :: [(Name, CoreExpr)] -> GmEnvironment -> GmEnvironment
> compileArgs defs env
>     = zip (map first defs) [n-1, n-2 .. 0] ++ argOffset n env
>             where n = length defs
```

An example

In this example we will show how the code for the fixpoint combinator Y is compiled. The definition we will use is:

```
Y f = letrec x = f x in x
```

This is the so-called 'knot-tying' fixpoint combinator; we will see why it has this name when we run the resulting code. When the above definition is compiled, the `compileSc` function will need to produce code for the supercombinator.

```
compileSc ("Y", ["f"], ELet True [("x", EAp (EVar "f") (EVar "x"))] (EVar "x"))
```

This in turn calls the `compileR` function with an environment for the variable `f`; having first created a name for the supercombinator (`Y`) and its number of arguments (`1`).

```
("Y", 1, compileR e [("f", 0)])
where e = ELet True [("x", EAp (EVar "f") (EVar "x"))] (EVar "x")
```

For convenience we will refer to the body of the expression as `e`. The function `compileR` calls `compileC`, placing the tidying-up code at the end.

```
("Y", 1, compileC e [("f", 0)] ++ [Update 1, Pop 1, Unwind])
```

Referring to the compilation scheme in Figure 3.10, we see that to compile a `letrec` we first create a new environment. In the figure this is called ρ'; in this example we will call it `p`. It is an extension of the initial environment in which we also give a stack location to the local variable `x`.

```
("Y", 1, [Alloc 1] ++
         compileC (EAp (EVar "f") (EVar "x")) p ++ [Update 0] ++
         compileC (EVar "x") p ++ [Slide 1] ++
         [Update 1, Pop 1, Unwind])
         where p = [("x", 0), ("f", 1)]
```

The code generation is laid out in the same fashion as the compilation scheme. When the expressions involving `compileC` are simplified we get:

```
("Y", 1, [Alloc 1] ++
         [Push 0, Push 2, Mkap] ++ [Update 0] ++
         [Push 0] ++ [Slide 1] ++
         [Update 1, Pop 1, Unwind])
```

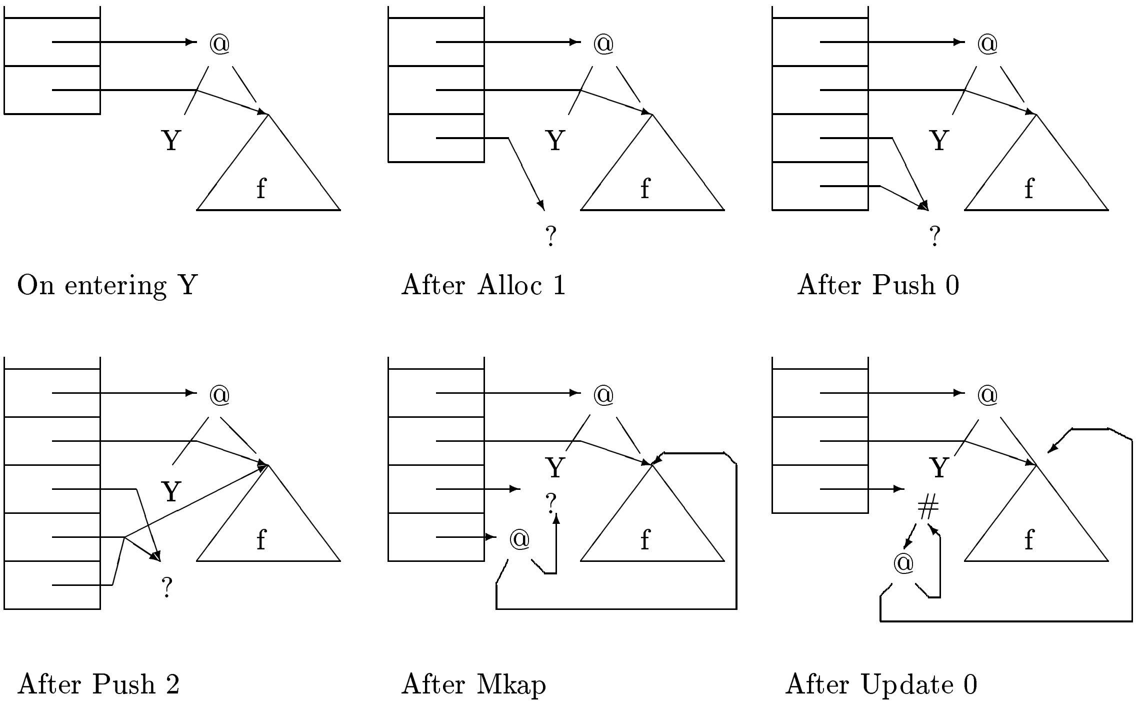


Figure 3.11: Execution of code for `Y`

Which gives the following code sequence:

```
("Y", 1, [Alloc 1, Push 0, Push 2, Mkap, Update 0, Push 0,
          Slide 1, Update 1, Pop 1, Unwind])
```

We can see the way in which this code executes in Figure 3.11. This definition of the `Y` supercombinator is called 'knot-tying' because we are tying a knot in the graph when we do the `Update 0` as the fifth instruction. We have not shown the remainder of the instructions, as this is left as Exercise 3.18.

Exercise 3.16. The compilation of `letrec`s is defined in Figure 3.10. Implement the function `compileLetrec` to perform this operation.

Exercise 3.17. What test programs would you use to show that the new compiler and instruction set work properly?

Exercise 3.18. By running the code generated for the supercombinator `Y`, or otherwise, draw the remainder of the state transitions in the style of Figure 3.11.

Exercise 3.19. Give a shorter, alternative, code sequence for the supercombinator `Y`. It should still construct a 'knot-tying' version.

Exercise 3.20. In the absence of a `letrec` construct in the language, how would you define the fixpoint combinator `Y`? How is this definition different from the one we used in the example?

3.6 Mark 4: Adding primitives

In this section we add primitive operations to the G-machine; this makes it useful. By primitive operations we mean operations like addition, multiplication and so on. We will use addition as a running example throughout this section.

The addition instruction will be called `Add`; which adds two numbers from the heap, placing the result into a new node in the heap. The addresses of the two arguments are on top of the stack, and this is where the address of the result is subsequently placed. It has the following transition rule.

$$\text{(3.21)} \quad \boxed{\begin{array}{lrrll} & \texttt{Add} : i & a_0 : a_1 : s & h[a_0 : \texttt{NNum}\ n_0,\ a_1 : \texttt{NNum}\ n_1] & m \\ \Longrightarrow & i & a : s & h[a : \texttt{NNum}\ n_0 + n_1] & m \end{array}}$$

We could continue to expand the G-machine with other instructions to implement the remainder of the operations required, but before we do, let us pause to consider whether we have missed something here. The problem is that the rule only applies if the two objects on top of the stack are numbers. Since we are working on a machine that supports lazy evaluation there is no good reason to suppose that this will always be the case. In the template machine `Add` checked that its arguments were evaluated. In the G-machine we want to keep the instructions simple, so we will only use `Add` in situations where we guarantee that the arguments are already evaluated.

What we do instead is to augment the instruction set further with an `Eval` instruction. This satisfies the following constraint:

> *Suppose that we are in a state:*
>
> $$\boxed{\texttt{Eval} : i \quad a : s \quad h \quad m}$$
>
> *Whenever execution resumes with instruction sequence i, the state will be:*
>
> $$\boxed{i \quad a : s \quad h' \quad m}$$
>
> *and the item on top of the stack will be in WHNF.*
>
> *It is also possible for* `Eval` *to fail to terminate; this will be the case when the node pointed to from the top of the stack has no WHNF.*

If the node whose address is on top of the stack is already in WHNF, then the `Eval` instruction does nothing. If there is reduction to be performed, then the action of `Eval` is to perform an evaluation to WHNF. If this call terminates then execution resumes with nothing except the heap component changed. This is similar to the structure of *subroutine call and return* traditionally used in programming language implementation. We recall that the classic way to implement this feature is to use a stack. The stack will save sufficient of the machine's current context that it can resume when the subroutine call completes.

In the Mark 4 G-machine this stack is called the *dump*, and is a stack of pairs, whose first component is a code sequence, and whose second component is a stack. This is similar to the dump in the template machine (see Section 2.6), except we now have to restore the original code sequence as well as the original stack. Hence both components are kept on the dump.

3.6.1 Data structures

We extend the G-machine state by adding a dump component. As previously discussed, this is used to implement recursive calls to the evaluator.

```
> type GmState = ( GmCode,              -- current Instruction
>                  GmStack,      -- current Stack
>                  GmDump,       -- current Dump
>                  GmHeap,       -- Heap of Nodes
>                  GmGlobals,    -- Global adresses in Heap
>                  GmStats)      -- Statistics
```

The dump itself is a stack of `dumpItem`. Each of these is a pair consisting of the instruction stream and stack to use when we resume the original computation.

```
> type GmDump = [GmDumpItem]
> type GmDumpItem = (GmCode, GmStack)
```

When we add this new component we must change all of the previously specified access functions. We must also add access functions for the dump.

```
> getDump :: GmState -> GmDump
> getDump (i, stack, dump, heap, globals, stats) = dump


> putDump :: GmDump -> GmState -> GmState
> putDump dump' (i, stack, dump, heap, globals, stats)
>    = (i, stack, dump', heap, globals, stats)
```

Notice that it is only in the access functions that we have done pattern matching on G-machine states. Changes to other functions as a result of adding new components to the state are no longer needed.

Exercise 3.21. Make the relevant changes to the other access functions.

In addition to the new definition of state, we also need some new instructions. We reuse all of the instructions from the Mark 3 machine.

```
> data Instruction
>           = Slide Int
>           | Alloc Int
>                   | Update Int
>           | Pop Int
>           | Unwind
>           | Pushglobal Name
>           | Pushint Int
>           | Push Int
>           | Mkap
```

In addition we include the Eval instruction,

```
>              | Eval
```

the following arithmetic instructions:

```
>              | Add | Sub | Mul | Div | Neg
```

and the following comparison instructions:

```
>              | Eq | Ne | Lt | Le | Gt | Ge
```

We also include a primitive form of conditional in the Cond instruction.

```
>            | Cond GmCode GmCode
```

Exercise 3.22. Add cases to showInstruction to print all of the new instructions.

3.6.2 Printing the state

We take this opportunity to revise the definition of showState, so that it displays the dump component.

```
> showState :: GmState -> Iseq
> showState s
>    = iConcat       [showStack s,                 iNewline,
>                     showDump s,                  iNewline,
>                     showInstructions (getCode s), iNewline]
```

We therefore need to define showDump.

```
> showDump :: GmState -> Iseq
> showDump s
>     = iConcat       [iStr "  Dump:[",
>                      iIndent (iInterleave iNewline
>                             (map showDumpItem (reverse (getDump s)))),
>                      iStr "]"]
```

This in turn needs the function showDumpItem.

```
> showDumpItem :: GmDumpItem -> Iseq
> showDumpItem (code, stack)
>     = iConcat       [iStr "<",
>                      shortShowInstructions 3 code, iStr ", ",
>                      shortShowStack stack,         iStr ">"]
```

We use the function `shortShowInstructions` to print only the first three instructions of the instruction stream in the dump items. This is usually sufficient to indicate where the computation will resume.

```
> shortShowInstructions :: Int -> GmCode -> Iseq
> shortShowInstructions number code
>     = iConcat [iStr "{", iInterleave (iStr "; ") dotcodes, iStr "}"]
>     where   codes   = map showInstruction (take number code)
>             dotcodes      | length code > number  = codes ++ [iStr "..."]
>                           | otherwise             = codes
```

Similarly, we do not need the full details of the stack component of the dump item either, so we use `shortShowStack`.

```
> shortShowStack :: GmStack -> Iseq
> shortShowStack stack
>     = iConcat [iStr "[",
>                iInterleave (iStr ", ") (map (iStr . showaddr) stack),
>                iStr "]"]
```

3.6.3 The new instruction transitions

Evaluator instructions

There are actually very few instructions that manipulate the dump. First, there is `Eval` itself, which creates a new dump item whenever the node on top of the stack is not in WHNF. Secondly, there is a modification to the `Unwind` instruction that pops a dump item when an evaluation is completed.

We first describe the new `Unwind` instruction. When the expression held in the stack is in WHNF, `Unwind` can restore the old context from the dump, placing the last address in the stack on the restored old stack. We see this clearly in the transition for the case of numbers.[1]

$$(3.22)\quad \boxed{\begin{array}{lrrrlr} & [\texttt{Unwind}] & a:s & \langle i',\, s'\rangle : d & h[a:\texttt{NNum}\ n] & m \\ \Longrightarrow & i' & a:s' & d & h & m \end{array}}$$

The expression with address a is in WHNF because it is an integer, so we restore the old instruction sequence i' and the stack is now the old stack s' with the address a on top. All other transitions for `Unwind` remain the same as they were in the Mark 3 machine (except that they have the dump component in their state).

We are now in a position to specify the rule for `Eval`. It saves the remainder of the stack s and the rest of the instructions i as a dump item on the dump. The new code sequence is just unwinding and the new stack contains the singleton a.

$$(3.23)\quad \boxed{\begin{array}{lrrrrr} & \texttt{Eval}:i & a:s & d & h & m \\ \Longrightarrow & [\texttt{Unwind}] & [a] & \langle i,\, s\rangle : d & h & m \end{array}}$$

[1]The rule only applies if the dump is non-empty; if the dump is empty then the machine has terminated.

Arithmetic instructions

The *dyadic arithmetic operators* all have the following generic transition rule. Let us suppose that the arithmetic operator we wish to implement is $\odot$; the transition rule for the instruction $\odot$ is then:

$$(3.24) \quad \boxed{\begin{array}{lrrcll} & \odot : i & a_0 : a_1 : s & d & h[a_0 : \texttt{NNum}\ n_0,\ a_1 : \texttt{NNum}\ n_1] & m \\ \Longrightarrow & i & a : s & d & h[a : \texttt{NNum}\ (n_0 \odot n_1)] & m \end{array}}$$

What has happened is that the two numbers on top of the stack have had the dyadic operator $\odot$ applied to them. The result, which is entered into the heap, has its address placed on the stack. The `Neg` instruction negates the number on top of the stack, so it has transition rule:

$$(3.25) \quad \boxed{\begin{array}{lrrcll} & \texttt{Neg} : i & a : s & d & h[a : \texttt{NNum}\ n] & m \\ \Longrightarrow & i & a' : s & d & h[a' : \texttt{NNum}\ (-n)] & m \end{array}}$$

Notice how similar all of the dyadic operations are. First we extract the two numbers from the heap, then we perform the operation, and finally we place the answer back in the heap. This suggests that we should write some higher-order functions that are parameterised over the extraction from heap (which we call 'unboxing' the value), and insertion back into the heap (which we call 'boxing' the value), along with the specific operation we wish to perform.

Let us write the boxing operations first. `boxInteger` takes a number and an initial state, and returns a new state in which the number has been placed into the heap, and a pointer to this new node left on top of the stack.

```
> boxInteger :: Int -> GmState -> GmState
> boxInteger n state
>       = putStack (a: getStack state) (putHeap h' state)
>       where (h', a) = hAlloc (getHeap state) (NNum n)
```

Now to extract an integer at address `a` from a state, we will use `unboxInteger`.

```
> unboxInteger :: Addr -> GmState -> Int
> unboxInteger a state
>       = ub (hLookup (getHeap state) a)
>       where   ub (NNum i) = i
>               ub n        = error "Unboxing a non-integer"
```

A generic *monadic operator* can now be specified in terms of its boxing function, `box`, its unboxing function `unbox`, and the operator `op` on the unboxed values.

```
> primitive1 :: (b -> GmState -> GmState)  -- boxing function
>          -> (Addr -> GmState -> a)       -- unbixing function
>          -> (a -> b)                     -- operator
>          -> (GmState -> GmState)         -- state transition
```

```
> primitive1 box unbox op state
>    = box (op (unbox a state)) (putStack as state)
>    where (a:as) = getStack state
```

The generic *dyadic operators* can now be implemented in a similar way using `primitive2`.

```
> primitive2 :: (b -> GmState -> GmState)  -- boxing function
>          -> (Addr -> GmState -> a)       -- unbixing function
>          -> (a -> a -> b)                -- operator
>          -> (GmState -> GmState)         -- state transition
```

```
> primitive2 box unbox op state
>    = box (op (unbox a0 state) (unbox a1 state)) (putStack as state)
>    where (a0:a1:as) = getStack state
```

To be even more explicit, `arithmetic1` implements all *monadic arithmetic*, and `arithmetic2` implements all *dyadic arithmetic*.

```
> arithmetic1 ::   (Int -> Int)            -- arithmetic operator
>                  -> (GmState -> GmState) -- state transition
```

```
> arithmetic1 = primitive1 boxInteger unboxInteger
```

```
> arithmetic2 ::   (Int -> Int -> Int)     -- arithmetic operation
>                  -> (GmState -> GmState) -- state transition
```

```
> arithmetic2 = primitive2 boxInteger unboxInteger
```

As the alert reader would expect, we will be taking advantage of the generality of these functions later in the chapter.

Exercise 3.23. Implement all of the new instruction transitions for the machine. Modify the `dispatch` function to deal with the new instructions. You should use the higher-order functions `primitive1` and `primitive2` to implement the operators.

Exercise 3.24. Why are indirection nodes never left on top of the stack on completing an `Eval` instruction?

Comparison instructions

The *comparison operators* all have the following generic transition rule. Let us suppose that the comparison operator we wish to implement is $\odot$; the transition rule for the instruction $\odot$ is then:

$$
(3.26) \qquad \begin{array}{|lrrcll|}
\hline
 & \odot : i & a_0 : a_1 : s & d & h[a_0 : \texttt{NNum}\ n_0,\ a_1 : \texttt{NNum}\ n_1] & m \\
\Longrightarrow & i & a : s & d & h[a : \texttt{NNum}\ (n_0 \odot n_1)] & m \\
\hline
\end{array}
$$

What has happened is that the two numbers on top of the stack have had the dyadic operator $\odot$ applied to them. The result, which is entered into the heap, has its address placed on the stack. This is almost the same as arithmetic.

The difference is that an operation, `==` say, returns a boolean and not an integer. To fix this we turn booleans into integers using the following rule:

- we represent `True` by the integer `1`;
- we represent `False` by the integer `0`.

To make the use of `primitive2` possible, we define `boxBoolean`

```
> boxBoolean :: Bool -> GmState -> GmState
> boxBoolean b state
>    = putStack (a: getStack state) (putHeap h' state)
>    where   (h',a) = hAlloc (getHeap state) (NNum b')
>            b' | b                = 1
>               | otherwise        = 0
```

Using this definition we can write a generic comparison function, which we call `comparison`. This function takes a *boxing* function for the booleans, the unboxing function for integers (`unboxInteger`), and a comparison operator; it returns a state transition.

```
> comparison :: (Int -> Int -> Bool) -> GmState -> GmState
> comparison = primitive2 boxBoolean unboxInteger
```

Finally, we implement the `Cond` instruction, which we will use to compile the `if` function. It has two transition rules:

$$
(3.27) \qquad \begin{array}{|lrrcll|}
\hline
 & \texttt{Cond}\ i_1\ i_2 : i & a : s & d & h[a : \texttt{NNum}\ 1] & m \\
\Longrightarrow & i_1 +\!\!+\ i & s & d & h & m \\
\hline
\end{array}
$$

In the first case – where there is the number `1` on top of the stack – we take the first branch. This means that we execute the instructions i_1 before continuing to execute the instructions i.

$$
(3.28) \qquad \begin{array}{|lrrcll|}
\hline
 & \texttt{Cond}\ i_1\ i_2 : i & a : s & d & h[a : \texttt{NNum}\ 0] & m \\
\Longrightarrow & i_2 +\!\!+\ i & s & d & h & m \\
\hline
\end{array}
$$

Alternatively, if the number on top of the stack is 0, we execute the instruction sequence i_2 first, and then the sequence i.

Exercise 3.25. Implement the transitions for the comparison instructions and the `Cond` instruction.

3.6.4 The compiler

The compiler will eventually need to be changed to take advantage of these new instructions to compile arithmetic expressions. For the moment we make only the minimum set of changes that will allow us to use the arithmetic instructions we have so laboriously added. First, the `compile` function must create a new initial state in which the initial dump is empty and in which the initial code sequence differs from the one we have used so far.

```
> compile :: CoreProgram -> GmState
> compile program
>    = (initialCode, [], [], heap, globals, statInitial)
>    where (heap, globals) = buildInitialHeap program
```

```
> initialCode :: GmCode
> initialCode = [Pushglobal "main", Eval]
```

Exercise 3.26. Why has the initial instruction sequence been changed? What happens if we retain the old one?

The simplest way to extend the compiler is simply to add G-machine code for each of the new built-in functions to the `compiledPrimitives`. The initial four instructions of the sequence ensure that the arguments have been evaluated to integers.

```
> compiledPrimitives :: [GmCompiledSC]
> compiledPrimitives
>    =       [("+", 2, [Push 1, Eval, Push 1, Eval, Add, Update 2, Pop 2, Unwind]),
>              ("-", 2, [Push 1, Eval, Push 1, Eval, Sub, Update 2, Pop 2, Unwind]),
>              ("*", 2, [Push 1, Eval, Push 1, Eval, Mul, Update 2, Pop 2, Unwind]),
>              ("/", 2, [Push 1, Eval, Push 1, Eval, Div, Update 2, Pop 2, Unwind]),
```

We also need to add the negation function. As this only takes one argument, we only evaluate one argument.

```
>              ("negate", 1, [Push 0, Eval, Neg, Update 1, Pop 1, Unwind]),
```

The comparison operations are implemented as follows.

```
>              ("==", 2, [Push 1, Eval, Push 1, Eval, Eq, Update 2, Pop 2, Unwind]),
>              ("~=", 2, [Push 1, Eval, Push 1, Eval, Ne, Update 2, Pop 2, Unwind]),
>              ("<",  2, [Push 1, Eval, Push 1, Eval, Lt, Update 2, Pop 2, Unwind]),
```

```
>              ("<=", 2, [Push 1, Eval, Push 1, Eval, Le, Update 2, Pop 2, Unwind]),
>              (">",  2, [Push 1, Eval, Push 1, Eval, Gt, Update 2, Pop 2, Unwind]),
>              (">=", 2, [Push 1, Eval, Push 1, Eval, Ge, Update 2, Pop 2, Unwind]),
```

The `if` function is compiled so that it uses `Cond` for the branching.

```
>              ("if", 3, [Push 0, Eval, Cond [Push 1] [Push 2],
>                                  Update 3, Pop 3, Unwind])]
```

Exercise 3.27. What test programs from Appendix B would you use in order to check that the new instructions and compiler work?

3.7 Mark 5: Towards better handling of arithmetic

The way the G-machine is implemented at the moment, each arithmetic operator is called via one of the compiled primitives. We can improve on this arrangement by observing that often we can call the arithmetic operator directly. For example, consider the following simple program:

```
main = 3+4*5
```

This generates the following code when we use the current compiler:

```
[Pushint 5, Pushint 4, Pushglobal "*", Mkap, Mkap,
 Pushint 3, Pushglobal "+", Mkap, Mkap, Eval]
```

When executed this code will take 33 steps and use 11 heap nodes. Our first thought must surely be that we can use the instructions `Add` and `Mul` in place of calls to the functions '`+`' and '`*`'. This leads to the following improved code:

```
[Pushint 5, Pushint 4, Mul, Pushint 3, Add]
```

This will take only five steps to execute and uses five heap nodes.

3.7.1 A problem

A possible problem arises when we consider our next example program.

```
main = K 1 (1/0)
```

This generates the following code:

```
[Pushint 0, Pushint 1, Pushglobal "/", Mkap, Mkap,
 Pushint 1, Pushglobal "K", Mkap, Mkap, Eval]
```

If we follow the pattern of the previous example we might try generating the code:

```
[Pushint 0, Pushint 1, Div,
 Pushint 1, Pushglobal "K", Mkap, Mkap, Eval]
```

The problem is that the division operator is applied *before* we reduce K, with the result that a *division-by-zero error* is generated. A correct compiler must generate code that will not give such errors.

What has happened is that our code is too strict. The code is evaluating expressions that it need not – which results in errors arising where they should not. A similar problem will also arise when *non-terminating expressions* are inadvertently evaluated.

3.7.2 The solution

The solution to the problem is to keep track of the context in which an expression appears. We will distinguish two contexts[2]:

Strict The value of the expression will be required in WHNF.

Lazy The value of the expression may or may not be required in WHNF.

Corresponding to each context, we have a compilation scheme which will compile an expression to a sequence of G-machine instructions. In the strict context this compilation scheme is the $\mathcal{E}$ scheme; in the lazy context we will use the $\mathcal{C}$ scheme we have seen already.

We would like to find as many strict contexts as possible, since these contexts allow us to generate better code. We make the following observation: whenever a supercombinator is instantiated it is because we wish to evaluate its value to WHNF. From this we conclude that the body of a supercombinator can always be evaluated in a strict context. There are also expressions where we know that some sub-expressions will be evaluated to WHNF if the expression is evaluated to WHNF.

The class of strict context expressions can be described recursively.

- The expression in the body of a supercombinator definition is in a strict context.
- If $e_0 \odot e_1$ occurs in a strict context, where $\odot$ is an arithmetic or comparison operator, then the expressions e_0 and e_1 are also in a strict context.
- If `negate` e occurs in a strict context, then the expression e also does.
- If the expression `if` e_0 e_1 e_2 occurs in a strict context, then so do the expressions e_0, e_1, and e_2.
- If `let(rec)` Δ `in` e occurs in a strict context then the expression e is also in a strict context.

An example should make this clear; consider the body of the supercombinator `f`:

[2] It is possible to distinguish more contexts. Projection analysis [Wadler 1987] and evaluation transformers [Burn 1991] are two ways to do this.

$\mathcal{R}[\![e]\!]\ \rho\ d$ generates code which instantiates the expression e in environment ρ, for a supercombinator of arity d, and then proceeds to unwind the resulting stack.

$$\mathcal{R}[\![e]\!]\ \rho\ d \quad = \quad \mathcal{E}[\![e]\!]\ \rho \mathbin{+\!\!+} [\texttt{Update}\ d, \texttt{Pop}\ d, \texttt{Unwind}]$$

$\mathcal{E}[\![e]\!]\ \rho$ compiles code that evaluates an expression e to WHNF in environment ρ, leaving a pointer to the expression on top of the stack.

$$\begin{array}{rcl}
\mathcal{E}[\![i]\!]\ \rho & = & [\texttt{Pushint}\ i] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{let}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
& = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
& & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} \mathbin{+\!\!+} \\
& & \mathcal{E}[\![e]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Slide}\ n] \\
& & \qquad\qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{letrec}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
& = & [\texttt{Alloc}\ n] \mathbin{+\!\!+} \\
& & \mathcal{C}[\![e_1]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Update}\ n-1] \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
& & \mathcal{C}[\![e_n]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Update}\ 0] \mathbin{+\!\!+} \\
& & \mathcal{E}[\![e]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Slide}\ n] \\
& & \qquad\qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\mathcal{E}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho & = & \mathcal{E}[\![e_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{E}[\![e_0]\!]\ \rho^{+1} \mathbin{+\!\!+} [\texttt{Add}] \\
& & \text{And similarly for other arithmetic and comparison expressions} \\
\mathcal{E}[\![\texttt{negate}\ e]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ \rho \mathbin{+\!\!+} [\texttt{Neg}] \\
\mathcal{E}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\ \rho & = & \mathcal{E}[\![e_0]\!]\ \rho \mathbin{+\!\!+} [\texttt{Cond}\ (\mathcal{E}[\![e_1]\!]\ \rho)\ (\mathcal{E}[\![e_2]\!]\ \rho)] \\
\mathcal{E}[\![e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho \mathbin{+\!\!+} [\texttt{Eval}] \qquad \text{the default case}
\end{array}$$

Figure 3.12: The $\mathcal{R}$ and $\mathcal{E}$ compilation schemes for the Mark 5 machine

```
f x y = (x+y) + g (x*y)
```

Both `(x+y)` and `g (x*y)` will be evaluated in a strict context – because the body of the supercombinator is. In the first case this causes `x` and `y` to be evaluated in a strict context – because `+` propagates the strict context. In the second expression, the presence of a user-defined supercombinator means that the sub-expression `x*y` will be compiled assuming that the expression may not be evaluated.

This suggests that we can implement the strict-context compiler, $\mathcal{E}$, in a recursive manner. Because the body of each supercombinator is evaluated in a strict context we will need to call the $\mathcal{E}$ scheme function from the $\mathcal{R}$ scheme. This satisfies the first of the points above. To propagate the context information into sub-expressions we will recursively invoke the $\mathcal{E}$ scheme for arithmetic expressions.

The new compiler schemes are defined in Figure 3.12.

To make it easier to extend the set of built-in operators that can be compiled using the new scheme, we define `builtInDyadic`.

```
> builtInDyadic :: ASSOC Name Instruction
> builtInDyadic
>     =        [("+", Add), ("-", Sub), ("*", Mul), ("div", Div),
>               ("==", Eq), ("~=", Ne), (">=", Ge),
>               (">",  Gt), ("<=", Le), ("<",  Lt)]
```

Exercise 3.28. Modify the existing compiler functions `compileR` and `compileE` so that they implement the $\mathcal{R}$ scheme and $\mathcal{E}$ scheme of Figure 3.12. You should use `builtInDyadic`.

Occasionally the machine will fail when the new compiler is used. What we need is to introduce a new rule for the `Unwind` instruction.

$$\begin{array}{|lccccl|} \hline & [\mathtt{Unwind}] & [a_0, \ldots, a_k] & \langle i, s\rangle : d & h[a_0 : \mathtt{NGlobal}\ n\ c] & m \\ \Longrightarrow & i & a_k : s & d & h & m \text{ when } k < n \\ \hline \end{array} \tag{3.29}$$

This allows us to use `Eval` to evaluate any object to WHNF, and not just numbers.

Exercise 3.29. Implement the new transition for `Unwind`. Write a program that fails without the new `Unwind` transition.

Exercise 3.30. Compare the execution of the example program used at the start of this section, with its execution on the Mark 4 machine.

```
main = 3+4*5
```

Try some other programs from Appendix B.

The way we implemented the strict context in the compiler is addis simple example of the *inherited attributes* from compiler theory. If we regard the strict context as an attribute of an expression, then a sub-expression *inherits* its strict context from its parent expression. The general theory is discussed in [Aho *et al.* 1986].

It is unfortunately not possible – in general – to determine at compile-time whether an expression should be compiled with a strict context. We therefore have to accept a compromise. In this book we have only treated a limited set of expressions – the arithmetic expressions – in a special way. Much active research is concerned with extending this analysis to cover more general expressions [Burn 1991].

3.8 Mark 6: Adding data structures

In this section we extend the G-machine to deal with arbitrary data structures. As discussed in Chapter 1, two new Core-language constructs are required for programs involving data structures: *constructors*, `EConstr`; and *case expressions*, `ECase`. Our goal, in producing the Mark 6 machine, is to compile code for these expressions.

3.8.1 Overview

In Section 1.1.4 we saw that a constructor with tag `t` and arity `a` was represented as `Pack{t,a}` in the core language. For example, the usual list data type has two constructors: `Pack{1,0}` and `Pack{2,2}`. These correspond to Miranda's [] and (:) respectively.

A `case` expression, which has no direct counterpart in Miranda, is used to inspect the values held in a constructor. For example, we can write the length function for a list as:

```
length xs = case xs of
                 <1>      -> 0;
                 <2> y ys -> 1 + length ys
```

It is instructive to look at the way we execute the `case` expression

1. To evaluate the `case` expression, we first evaluate `xs` to WHNF.
2. Once this evaluation has occurred, we are able to tell which of the *alternatives* to take. The *tag* of the evaluated expression – which must be a structured data object – determines which alternative we take. In the above example, for `length`:
   - If the tag of the constructor for `xs` is `1` then the list is empty and we take the first branch. We therefore return `0`.
   - If the tag is `2`, then the list is non-empty. This time there are components of the constructor (`y` and `ys`). The length of the list is one more than the length of `ys`.

We will assume that whenever we attempt to dismantle a constructor it has been applied to the correct number of arguments. A constructor in this state is said to be *saturated.* As an example, in Section 1.1.3, `Cons` is defined to take two arguments, so it is saturated when it is applied to two expressions.

We also note that a Core-language program can now return a result that is a structured data object. The Mark 5 G-machine must be able to print the structured data object in a lazy fashion. Let us first consider what additions will need to be made to the data structures of the Mark 5 machine.

3.8.2 Data structures

It would be nice to allow the machine to return values which are not just numbers. We would like to be able to return values that consist of constructors. This will require us to evaluate the components of the structure recursively, and then return these values. To do this we need to add yet another component to the state: `gmOutput`. This will hold the result of the program.

```
> type GmState =
>    (GmOutput,               -- Current Output
>     GmCode,                 -- Current Instruction Stream
>     GmStack,                -- Current Stack
>     GmDump,                 -- The Dump
```

```
>     GmHeap,                  -- Heap of Nodes
>     GmGlobals,               -- Global addresses in Heap
>     GmStats)                 -- Statistics
```

This component is defined to be a character string.

```
> type GmOutput = [Char]
```

We can write the access functions in the obvious way.

```
> getOutput :: GmState -> GmOutput
> getOutput (o, i, stack, dump, heap, globals, stats) = o


> putOutput :: GmOutput -> GmState -> GmState
> putOutput o' (o, i, stack, dump, heap, globals, stats)
>      = (o', i, stack, dump, heap, globals, stats)
```

Exercise 3.31. Make the appropriate changes to the remainder of the access functions.

To support constructor nodes in the heap, we augment the type `node` with `NConstr`; this takes a positive number which will represent a *tag*, and a list of *components* which we represent as the list of the addresses of the nodes in heap.

```
> data Node
>           = NNum Int              -- Numbers
>           | NAp Addr Addr         -- Applications
>           | NGlobal Int GmCode    -- Globals
>           | NInd Addr
>           | NConstr Int [Addr]
> instance Eq Node
>   where
>   NNum a       == NNum b          = a == b    -- needed to check conditions
>   NAp a b      == NAp c d         = False     -- not needed
>   NGlobal a b  == NGlobal c d     = False     -- not needed
>   NInd a       == NInd b          = False     -- not needed
>   NConstr a b  == NConstr c d     = False     -- not needed
```

3.8.3 Printing the result

Because we have a new state component which we wish to display, we must redefine the function `showState`.

```
> showState :: GmState -> Iseq
> showState s
```

```
>      = iConcat [showOutput s,                 iNewline,
>              showStack s,                   iNewline,
>              showDump s,                    iNewline,
>              showInstructions (getCode s), iNewline]
```

The showOutput function is easy, because the output component is already a string.

```
> showOutput :: GmState -> Iseq
> showOutput s = iConcat [iStr "Output:\"", iStr (getOutput s), iStr "\""]
```

The only other change (apart from changing showInstruction for the new instruction set) occurs in showNode, because we have extended the data type to include constructor nodes.

```
> showNode :: GmState -> Addr -> Node -> Iseq
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>   where v = head [n | (n,b) <- getGlobals s, a==b]
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ",  iStr (showaddr a1),
>                                       iStr " ",    iStr (showaddr a2)]
> showNode s a (NInd a1)     = iConcat [iStr "Ind ", iStr (showaddr a1)]
> showNode s a (NConstr t as)
>  = iConcat [iStr "Cons ", iNum t, iStr " [",
>             iInterleave (iStr ", ") (map (iStr.showaddr) as), iStr "]"]
```

3.8.4 The instruction set

The new instruction set is now defined. It simply adds four new instructions to the Mark 4 machine.

```
> data Instruction
>    = Slide Int
>    | Alloc Int
>    | Update Int
>    | Pop Int
>    | Unwind
>    | Pushglobal Name
>    | Pushint Int
>    | Push Int
>    | Mkap
>    | Eval
>    | Add | Sub | Mul | Div
>    | Neg
>    | Eq | Ne | Lt | Le | Gt | Ge
>    | Cond GmCode GmCode
```

The four new instructions that are added to the machine are as follows:

```
>     | Pack Int Int
>     | Casejump [(Int, GmCode)]
>     | Split Int
>     | Print
```

Exercise 3.32. Extend `showInstruction` to match the new instruction set.

The `Pack` instruction is simple; it assumes that there are sufficient arguments on the stack to construct a *saturated constructor*. When there are, it proceeds to make a saturated constructor; if there are not enough arguments, then the instruction is undefined.

$$
(3.30) \quad \boxed{\begin{array}{llllll} & o & \texttt{Pack}\ t\ n : i & a_1 : \ldots : a_n : s & d & h & m \\ \Longrightarrow & o & i & a : s & d & h[a : \texttt{NConstr}\ t[a_1, \ldots, a_n]] & m \end{array}}
$$

The transition rule for `Casejump` expects (a) that the node on top of the stack is in WHNF, and (b) that the node is a structured data object. Using the tag from this object we select one of the alternative instruction sequences, and the current instruction stream is then prefixed by the code for the particular alternative selected.

$$
(3.31) \quad \boxed{\begin{array}{llllll} & o & \texttt{Casejump}\ [\ldots, t\texttt{->}i', \ldots] : i & a : s & d & h[a : \texttt{NConstr}\ t\ ss] & m \\ \Longrightarrow & o & i' +\!\!+ i & a : s & d & h & m \end{array}}
$$

This is a simple way to specify a multiway jump and join. That is, by prefixing the current code i by the code for the alternative i', we achieve the effect of first running the code for the alternative and then resuming with whatever the remainder of the code for the main expression requires[3].

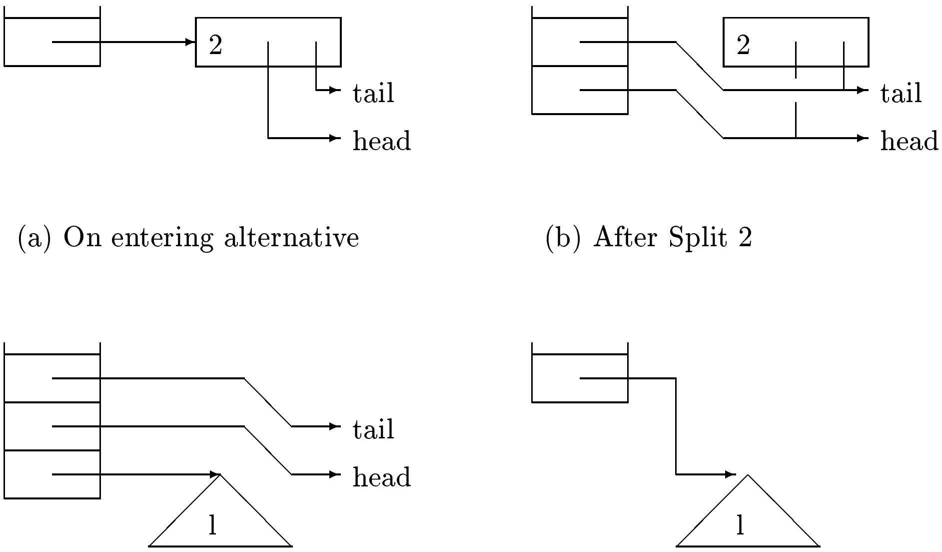
The code for each alternative begins with a `Split` n instruction and terminates with a `Slide` n instruction. The value of n is determined by the number of components in the constructor. The `Split` instruction is used to gain access to the components of a constructor.

Consider the code sequence generated for the `length` function:

```
[Push 0, Eval,
 Casejump [1 -> [Pushint 0]
           2 -> [Split 2, Push 1, Pushglobal "length", Mkap,
                 Eval, Pushint 1, Add, Slide 2]],
 Update 1,
 Pop 1,
 Unwind]
```

The execution of this pattern is shown in Figure 3.13, where we see that the `Slide` and `Split` instructions are being used temporarily to extend the current set of local bindings. Assuming

[3]It should be noted that code sequences using `Casejump` are not flat. We can, however, construct the flat code sequences we desire by labelling each alternative and jumping to the labelled code addresses. We have not done this as it unnecessarily complicates the code generation.



(a) On entering alternative

(b) After Split 2

(c) After code for body

(d) After Slide 2

Figure 3.13: Running compiled code for alternatives

that the `length` function was applied to a non-nil node, when we execute the `Casejump` instruction, we take the alternative labelled 2. This is the initial diagram (a). The `Split 2` instruction 'unpacks' the constructor node onto the stack. This is shown in diagram (b). After completing the body of the alternative, i.e. the code sequence

```
[Push 1, Pushglobal "length", Mkap, Eval, Pushint 1, Add]
```

the length of the list argument to this call of `length` will be on top of the stack labelled `l` in diagram (c). To complete the execution we remove the pointers to `head` and `tail`; this is shown in diagram (d). The transition for `Split` is straightforward.

$$(3.32) \quad \boxed{\begin{array}{lllllll} & o & \texttt{Split}\ n : i & a : s & d & h[a : \texttt{NConstr}\ t\ [a_1, \ldots, a_n]] & m \\ \Longrightarrow & o & i & a_1 : \ldots : a_n : s & d & h & m \end{array}}$$

Next, we describe the transitions for `Print`. There are two transitions for `Print`; one each for constructors and numbers.

$$(3.33) \quad \boxed{\begin{array}{lllllll} & o & \texttt{Print} : i & a : s & d & h[a : \texttt{NNum}\ n] & m \\ \Longrightarrow & o +\!\!+ [n] & i & s & d & h & m \end{array}}$$

The rule for constructors is more complex, as it must arrange to print each component of the constructor. For simplicity we will only print out the components.

$$(3.34) \quad \boxed{\begin{array}{lllllll} & o & \texttt{Print} : i & a : s & d & h[a : \texttt{NConstr}\ t\ [a_1, \ldots, a_n]] & m \\ \Longrightarrow & o & i' +\!\!+ i & a_1 : \ldots : a_n : s & d & h & m \end{array}}$$

$\mathcal{E}[\![e]\!]\ \rho$ compiles code that evaluates an expression e to WHNF in environment ρ, leaving a pointer to the expression on top of the stack.

$$\begin{array}{rcl}\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Casejump}\ \mathcal{D}[\![alts]\!]\ \rho] \\ \mathcal{E}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}\ e_1\ \ldots\ e_a]\!]\ \rho & = & \mathcal{C}[\![e_a]\!]\ \rho^{+0} +\!\!+ \ldots \mathcal{C}[\![e_1]\!]\ \rho^{+(a-1)} +\!\!+ [\texttt{Pack}\ t\ a]\end{array}$$

$\mathcal{C}[\![e]\!]\ \rho$ generates code which constructs the graph of e in environment ρ, leaving a pointer to it on top of the stack.

$$\mathcal{C}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}\ e_1\ \ldots\ e_a]\!]\ \rho \quad = \quad \mathcal{C}[\![e_a]\!]\ \rho^{+0} +\!\!+ \ldots \mathcal{C}[\![e_1]\!]\ \rho^{+(a-1)} +\!\!+ [\texttt{Pack}\ t\ a]$$

$\mathcal{D}[\![alts]\!]\ \rho$ compiles the code for the alternatives in a `case` expression.

$$\mathcal{D}[\![alt_1\ \ldots\ alt_n]\!]\ \rho \quad = \quad [\mathcal{A}[\![alt_1]\!]\ \rho\ , \ldots,\ \mathcal{A}[\![alt_n]\!]\ \rho]$$

$\mathcal{A}[\![alt]\!]\ \rho$ compiles the code for an alternative in a `case` expression.

$$\begin{array}{rcl}\mathcal{A}[\![\texttt{<}t\texttt{>}\ x_1 \ldots x_n\ \texttt{->}\ body]\!]\ \rho & = & t\ \texttt{->}\ [\texttt{Split}\ n] +\!\!+ \mathcal{E}[\![body]\!]\ \rho' +\!\!+ [\texttt{Slide}\ n] \\ & & \qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto 0 \ldots x_n \mapsto n-1]\end{array}$$

Figure 3.14: Compilation schemes for `case` expressions

The code i' is simply:

$$[\underbrace{\texttt{Eval, Print}, \ldots, \texttt{Eval, Print}}_{n}].$$

Lastly, we must add a new rule for `Unwind`, that tells it to return when unwinding an `NConstr`, just like the rule for `NNum`.

$$(3.35)\quad \begin{array}{|lrlllr|}\hline & [\texttt{Unwind}] & a : s & \langle i',\ s'\rangle : d & h[a : \texttt{NConstr}\ n\ as] & m \\ \Longrightarrow & i' & a : s' & d & h & m \\ \hline\end{array}$$

Exercise 3.33. Implement the new transitions and modify the `dispatch` function.

3.8.5 The compiler

In Figure 3.14 the new cases for the $\mathcal{E}$ and $\mathcal{C}$ compilation schemes are presented. They require auxiliary compilation schemes, $\mathcal{D}$ and $\mathcal{A}$, to deal with the alternatives that may be selected in a `case` expression. The function `compileAlts` (corresponding to the $\mathcal{D}$ scheme) compiles a list of alternatives, using the current environment, and produces a list of tagged code sequences. It also uses the `comp` argument (which corresponds to $\mathcal{A}$) to compile the body of each of the alternatives. For the moment this argument will always be `compileE'`.

```
> compileAlts ::    (Int -> GmCompiler)     -- compiler for alternative bodies
>                   -> [CoreAlt]            -- the list of alternatives
>                   -> GmEnvironment        -- the current environment
>                   -> [(Int, GmCode)]      -- list of alternative code sequences
> compileAlts comp alts env
>  = [(tag, comp (length names) body (zip names [0..] ++ argOffset (length names) env))
>          | (tag, names, body) <- alts]
```

The `compileE'` scheme is a small modification to the `compileE` scheme. It simply places a `Split` and `Slide` around the code generated by the ordinary `compileE` scheme.

```
> compileE' :: Int -> GmCompiler
> compileE' offset expr env
>     = [Split offset] ++ compileE expr env ++ [Slide offset]
```

Exercise 3.34. Make the relevant changes to `compile`, and modify `initialCode` to have a final `Print` instruction.

Exercise 3.35. Add the new cases to the compiler functions `compileE` and `compileC`.

Exercise 3.36. What changes are required to print out the output in 'structured form'. By this we mean placing the constructors and parentheses into the output component `gmOutput`, as well as integers.

3.8.6 Using the new boolean representation in comparisons

In this section we show how the Mark 6 machine we have constructed, can be modified to use our new representation of booleans. We first observe that we can implement the booleans as structured data objects; with `True` and `False` being represented as constructors of zero arity and tags 2 and 1 respectively.

How do we implement conditionals? This can be done by adding a new definition to the program for `if`. It returns either its second or third argument, depending on the first argument.

```
        if c t f = case c of
                        <1> -> f;
                        <2> -> t
```

The first change we require lies in the comparison operations. These have the following generic transition rule.

$$(3.36) \qquad \begin{array}{|lccccll|} & o & \odot : i & a_0 : a_1 : s & d & h[a_0 : \texttt{NNum}\ n_0,\ a_1 : \texttt{NNum}\ n_1] & m \\ \Longrightarrow & o & i & a : s & d & h[a : \texttt{Constr}\ (n_0 \odot n_1)\ []] & m \end{array}$$

For example, in the `Eq` instruction, we replace $\odot$ with a function that returns 2 (the tag for `True`) if the two numbers n_0 and n_1 are the same, and 1 (the tag for `False`) otherwise.

We can implement the transitions quickly, by reusing some of the code we developed for the Mark 4 machine. In Section 3.6.3 we saw how to represent some generic arithmetic and comparison operators. In fact, because of the way in which we structured the definition of the `comparison` function, we can almost immediately use our new representation of booleans.

The boxing function `boxBoolean` takes a comparison operation and a state in which there are two integers on top of the stack. It returns the new state in which there is the boolean result of comparing the two integers on top of the stack.

```
> boxBoolean :: Bool -> GmState -> GmState
> boxBoolean b state
>  = putStack (a: getStack state) (putHeap h' state)
>    where (h',a) = hAlloc (getHeap state) (NConstr b' [])
>          b' | b = 2              -- 2 is tag of True
>             | otherwise = 1      -- 1 is tag of False
```

Exercise 3.37. Run some example programs from Appendix B; for example try the factorial program:

```
fac n = if (n==0) 1 (n * fac (n-1))
```

3.8.7 Extending the language accepted

As astute readers might have noticed, there are some legal expressions involving `ECase` and `EConstr` for which our compiler will fail. The legal expressions that we cannot compile fall into two classes:

1. Occurrences of `ECase` in non-strict contexts; i.e. in expressions compiled by the $\mathcal{C}$ scheme.
2. Occurrences of `EConstr` in expressions where it is applied to too few arguments.

Both problems can be solved by using program transformation techniques. The solution for `ECase` is to make the offending expressions into supercombinators which are then applied to their free variables. For example, the program:

```
f x = Pack{2,2} (case x of <1> -> 1; <2> -> 2) Pack{1,0}
```

can be transformed into the equivalent program:

```
f x = Pack{2,2} (g x) Pack{1,0}
g x = case x of <1> -> 1; <2> -> 2
```

The trick for `EConstr` is to create a supercombinator for each constructor; this will be generated with enough free variables to saturate the constructor. Here is an example.

```
prefix p xs = map (Pack{2,2} p) xs
```

This is transformed to:

```
prefix p xs = map (f p) xs
f p x = Pack{2,2} p x
```

Another way to solve this problem is to modify the `Pushglobal` instruction, so that it will work for functions with names of the form: '`Pack{t,a}`'. We can then simply look for constructor functions, such as `f` in the example above, in the `globals` component of the state. If the function is not already present, we can create a new global node to associate with the function because the node has a particularly simple structure.

```
NGlobal a [Pack t a, Update 0, Unwind]
```

The new transitions are, firstly, if the function exists already:

$$(3.37)\quad \boxed{\begin{array}{llllllll} & o & \texttt{Pushglobal Pack\{}t\texttt{,}n\texttt{\}} : i & s & d & h & m[\texttt{Pack\{}t\texttt{,}n\texttt{\}} : a] \\ \Longrightarrow & o & i & a : s & d & h & m \end{array}}$$

and secondly when it does not already exist:

$$(3.38)\quad \boxed{\begin{array}{llllllll} & o & \texttt{Pushglobal Pack\{}t\texttt{,}n\texttt{\}} : i & s & d & h & m \\ \Longrightarrow & o & i & a : s & d & h[a : \texttt{gNode}_{t,n}] & m[\texttt{Pack\{}t\texttt{,}n\texttt{\}} : a] \end{array}}$$

where $\texttt{gNode}_{t,n}$ is

$$\texttt{NGlobal}\ n\ [\texttt{Pack}\ t\ n,\ \texttt{Update}\ 0,\ \texttt{Unwind}]$$

Our compiler can then generate code for expressions with unsaturated constructor nodes directly. It does this by generating the following code for unsaturated constructors.

$$\mathcal{C}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}]\!]\ \rho = [\texttt{Pushglobal "Pack\{}t\texttt{,}a\texttt{\}"}]$$

Exercise 3.38. Implement the extensions to the `pushglobal` function for the `Pushglobal` instruction and modify the compiler.

3.9 Mark 7: Further improvements

Let us consider again the example program we saw when we developed the Mark 5 compiler.

```
main = 3+4*5
```

This generates the following code when we use the Mark 6 compiler:

```
[Pushint 5, Pushint 4, Mul, Pushint 3, Add]
```

When executed this code will use five heap nodes. Is it possible to reduce this still further?

The answer is yes. We can reduce the number of heap accesses for arithmetic further by using a stack of numbers to represent intermediate values in the computation. In the Mark 7 machine these values are held in a new state component called the *V-stack*. The problem is that placing numbers into the heap or extracting them is an expensive operation on a real machine. It is much more efficient to use the machine's register set or stack. In the Mark 7 G-machine we will use a stack; this means that we do not have to worry about running out of registers.

The new code for the program

```
main = 3+4*5
```

is very similar to that which we previously generated:

```
[Pushbasic 5, Pushbasic 4, Mul, Pushbasic 3, Add, Mkint]
```

The first instruction `Pushbasic 5` places `5` on top of the V-stack. Next we push `4` onto the V-stack, following this by a multiplication. This instruction now expects its arguments to be in the V-stack. It will place the answer into the V-stack as well. The next two instructions add `3` to the value on top of the V-stack. The final instruction, `Mkint`, takes the value on top of the V-stack and places it into a number node in the heap, leaving a pointer to this new node on top of the S-stack.

3.9.1 Executing the factorial function using the V-stack

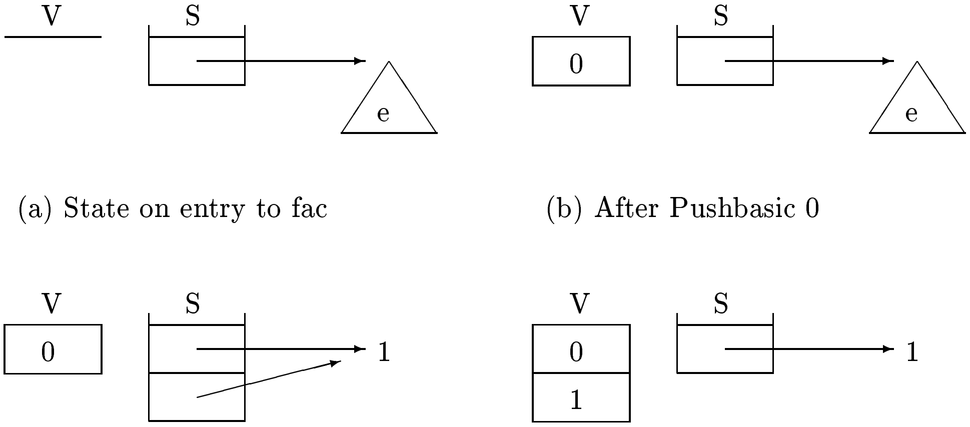
We begin an investigation into the Mark 7 machine, by way of an example. We will be looking at the execution of the factorial function, defined as follows:

```
fac n = if (n==0) 1 (n * fac (n-1))
```

Using the Mark 7 compiler, we will generate the following code sequence for the body of the supercombinator.

```
[Pushbasic 0, Push 0, Eval, Get, Eq,
 Cond [Pushint 1, Update 1, Pop 1, Unwind]
      [Pushint 1, Push 1, Pushglobal "-", Mkap, Mkap, Pushglobal "fac", Mkap,
       Eval, Get, Push 0, Eval, Get, Mul, Mkint, Update 1, Pop 1, Unwind]
```

When this code commences execution, the V-stack is empty, and there is one item on the ordinary stack. We will call the latter the S-stack from now on to distinguish the two sorts of stack. In Figure 3.15 – starting with diagram (a) – we see the initial state in which a pointer to the argument to `fac` is on top of the S-stack. In diagram (b) we see what happens when a `Pushbasic` instruction is executed: an integer is pushed onto the V-stack. Diagram (c) shows that the argument to `fac` has now been evaluated, whilst in diagram (d) we see the effect of a `Get` instruction. It has extracted the value from the node in the heap, and placed it into the V-stack.



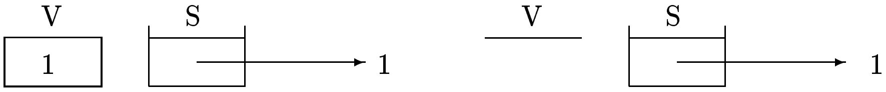
(a) State on entry to fac

(b) After Pushbasic 0

(c) After Push 0, Eval

(d) After Get

Figure 3.15: Mark 7 machine running `fac`



(e) After Eq

(f) After Cond

Figure 3.16: Mark 7 machine running `fac`

In diagram (e) (Figure 3.16) we see the state after an `Eq` instruction has executed. It has compared the two items in the V stack, and discovered that they are not equal. The Mark 7 G-machine represents the boolean value `False` by `1` in the V-stack. In diagram (f), the `Cond` instruction has inspected this value, and used it to select which branch to execute.

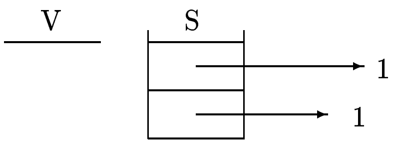
In diagram (g) (Figure 3.17) the state after the construction and evaluation of `fac (1-1)` is shown. The next instruction is `Get` which fetches the newly evaluated value into the V stack. Diagram (i) shows that we evaluate and fetch the value from the node `1` into the V-stack. In diagram (j) a `Mul` instruction has multiplied the two values in the V-stack, placing the result back there. In diagram (k) a `Mkint` instruction has moved this result from the V-stack to the heap, recording the address of the newly created node on the S-stack.

In a machine where there is a performance penalty for creating and accessing objects in the heap – compared with keeping the objects in a stack – we expect the use of the V-stack to be an improvement. Having seen how the V-stack works, we now make small modifications to the G-machine to implement the Mark 7 machine.

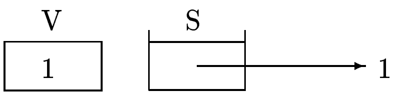
3.9.2 Data structures

The use of the V-stack requires that each G-machine state has a new state component `gmVStack` added to its state.

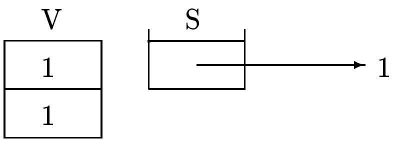
```
> type GmState = (GmOutput,      -- Current output
```



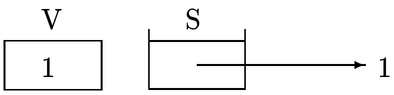
(g) After Pushint 1, ... Mkap, Eval



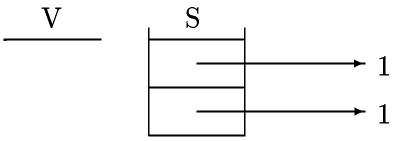
(h) After Get



(i) After Push 0, Eval, Get



(j) After Mul



(k) After Mkint

Figure 3.17: Mark 7 machine running `fac`

```
>               GmCode,             -- Current instruction stream
>               GmStack,            -- Current stack
>               GmDump,             -- Current dump
>               GmVStack,           -- Current V-stack
>               GmHeap,             -- Heap of nodes
>               GmGlobals,          -- Global addresses in heap
>               GmStats)            -- Statistics
```

As we have already stated this new component behaves as a stack of numbers.

```
> type GmVStack = [Int]
```

We add access functions for this component.

```
> getVStack :: GmState -> GmVStack
> getVStack (o, i, stack, dump, vstack, heap, globals, stats) = vstack


> putVStack :: GmVStack -> GmState -> GmState
> putVStack vstack' (o, i, stack, dump, vstack, heap, globals, stats)
>      = (o, i, stack, dump, vstack', heap, globals, stats)
```

Exercise 3.39. Make the relevant changes to the other access functions.

Displaying the states

The function showState is changed so that it prints out the V-stack component.

```
> showState :: GmState -> Iseq
> showState s
>      = iConcat [showOutput s,                 iNewline,
>            showStack s,                  iNewline,
>            showDump s,                   iNewline,
>            showVStack s,                 iNewline,
>            showInstructions (getCode s), iNewline]
```

To do this we define the function showVStack.

```
> showVStack :: GmState -> Iseq
> showVStack s
>      = iConcat [iStr "Vstack:[",
>            iInterleave (iStr ", ") (map iNum (getVStack s)),
>            iStr "]"]
```

3.9.3 Instruction set

An obvious first requirement is that each of the arithmetic transitions will now have to be modified to get their values from, and return their results to, the V-stack instead of the ordinary stack. Let us first deal with the case of dyadic primitives. The generic transition for the operation $\odot$ is given below. It takes two arguments from the V-stack and places the result of the operation $\odot$ back onto the V-stack.

$$(3.39)\quad \boxed{\begin{array}{lllllllll} & o & \odot : i & s & d & n_0 : n_1 : v & h & m \\ \Longrightarrow & o & i & s & d & n_0 \odot n_1 : v & h & m \end{array}}$$

The Neg instruction now has the following transition: it simply replaces the number on top of the V-stack with its negation.

$$(3.40)\quad \boxed{\begin{array}{lllllllll} & o & \texttt{Neg} : i & s & d & n : v & h & m \\ \Longrightarrow & o & i & s & d & (-n) : v & h & m \end{array}}$$

We also need instructions to move values between the heap and the V-stack. We begin with Pushbasic, which pushes an integer n onto the V-stack.

$$(3.41)\quad \boxed{\begin{array}{lllllllll} & o & \texttt{Pushbasic}\ n : i & s & d & v & h & m \\ \Longrightarrow & o & i & s & d & n : v & h & m \end{array}}$$

To move a value from the V-stack to the heap, we use two instructions: Mkbool and Mkint. These treat the integer on top of the V-stack as booleans and integers respectively. We begin with Mkbool.

(3.42)
$$\begin{array}{|lllllllll|}\hline & o & \texttt{Mkbool} : i & s & d & t : v & h & m \\ \Longrightarrow & o & i & a : s & d & v & h[a : \texttt{NConstr}\ t\ []] & m \\ \hline\end{array}$$

The transition for Mkint is similar, except that it creates a new integer node in the heap.

(3.43)
$$\begin{array}{|llllllll|}\hline & o & \texttt{Mkint} : i & s & d & n : v & h & m \\ \Longrightarrow & o & i & a : s & d & v & h[a : \texttt{NNum}\ n] & m \\ \hline\end{array}$$

To perform the inverse operation we use Get. This is specified by two transitions. In the first we see how Get treats a boolean on top of the stack.

(3.44)
$$\begin{array}{|llllllll|}\hline & o & \texttt{Get} : i & a : s & d & v & h[a : \texttt{NConstr}\ t\ []] & m \\ \Longrightarrow & o & i & s & d & t : v & h & m \\ \hline\end{array}$$

In the second, we see how Get treats a number.

(3.45)
$$\begin{array}{|llllllll|}\hline & o & \texttt{Get} : i & a : s & d & v & h[a : \texttt{NNum}\ n] & m \\ \Longrightarrow & o & i & s & d & n : v & h & m \\ \hline\end{array}$$

Finally, to make use of booleans on the V-stack we use a simplified Casejump instruction that inspects the V-stack to determine which instruction stream to use. This new instruction is called Cond, and is specified by the following two transitions. In the first – with the value on top of the V-stack being true – we select the first code sequence: t.

(3.46)
$$\begin{array}{|llllllll|}\hline & o & \texttt{Cond}\ t\ f : i & s & d & 2 : v & h & m \\ \Longrightarrow & o & t \mathbin{+\!\!+} i & s & d & v & h & m \\ \hline\end{array}$$

In the second transition we see that with a false value on top of the V-stack Cond selects its second code sequence: f.

(3.47)
$$\begin{array}{|llllllll|}\hline & o & \texttt{Cond}\ t\ f : i & s & d & 1 : v & h & m \\ \Longrightarrow & o & f \mathbin{+\!\!+} i & s & d & v & h & m \\ \hline\end{array}$$

Exercise 3.40. Extend the instruction data type, redefine the showInstruction function, implement the new instruction transitions and modify the dispatch function.

We now consider the – rather extensive – modifications to the compiler.

3.9.4 The compiler

Because of the extra state component, the compile function must initialise the V-stack component to be empty.

```
> compile :: CoreProgram -> GmState
> compile program
>      = ([], initialCode, [], [], [], heap, globals, statInitial)
>      where (heap, globals) = buildInitialHeap program
```

Strictly speaking, this is all that is necessary to make the machine work, but we have introduced the V-stack so that we can compile arithmetic functions 'in-line', so this is what we intend our code to do.

```
> buildInitialHeap :: CoreProgram -> (GmHeap, GmGlobals)
> buildInitialHeap program
>      = mapAccuml allocateSc hInitial compiled
>      where compiled = map compileSc (preludeDefs ++ program ++ primitives)
```

Because of the changes to the transitions for the primitive instruction, we must change the code for each compiled primitive. Instead of hand compiling this code – as we did for the Mark 6 machine – we can instead give this job to the compiler. This of course relies on the fact that the compiler is clever enough to optimise the code it produces, otherwise we never generate any `Add` instructions!

```
> primitives :: [(Name,[Name],CoreExpr)]
> primitives
>      =       [("+", ["x","y"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "y"))),
>               ("-", ["x","y"], (EAp (EAp (EVar "-") (EVar "x")) (EVar "y"))),
>               ("*", ["x","y"], (EAp (EAp (EVar "*") (EVar "x")) (EVar "y"))),
>               ("/", ["x","y"], (EAp (EAp (EVar "/") (EVar "x")) (EVar "y"))),
```

We also need to add the negation function.

```
>               ("negate", ["x"], (EAp (EVar "negate") (EVar "x"))),
```

Comparison functions are almost identical to the dyadic arithmetic functions.

```
>               ("==", ["x","y"], (EAp (EAp (EVar "==") (EVar "x")) (EVar "y"))),
>               ("~=", ["x","y"], (EAp (EAp (EVar "~=") (EVar "x")) (EVar "y"))),
>               (">=", ["x","y"], (EAp (EAp (EVar ">=") (EVar "x")) (EVar "y"))),
>               (">",  ["x","y"], (EAp (EAp (EVar ">")  (EVar "x")) (EVar "y"))),
>               ("<=", ["x","y"], (EAp (EAp (EVar "<=") (EVar "x")) (EVar "y"))),
>               ("<",  ["x","y"], (EAp (EAp (EVar "<")  (EVar "x")) (EVar "y"))),
```

Finally, we ought to include the conditional function, and some supercombinators to represent boolean values.

```
>               ("if",  ["c","t","f"],
>                      (EAp (EAp (EAp (EVar "if") (EVar "c")) (EVar "t")) (EVar "f"))),
>               ("True",  [], (EConstr 2 0)),
>               ("False", [], (EConstr 1 0))]
```

$\mathcal{B}[\![e]\!]\ \rho$ compiles code that evaluates an expression e to WHNF, in an environment ρ, leaving the result on the V stack.

$$
\begin{array}{rcl}
\mathcal{B}[\![i]\!]\ \rho & = & [\texttt{Pushbasic}\ i] \\
\multicolumn{3}{l}{\mathcal{B}[\![\texttt{let}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ \\
 & & \mathcal{B}[\![e]\!]\ \rho' +\!\!+ [\texttt{Pop}\ n] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\multicolumn{3}{l}{\mathcal{B}[\![\texttt{letrec}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
 & = & [\texttt{Alloc}\ n] +\!\!+ \\
 & & \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{Update}\ n-1] +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{Update}\ 0] +\!\!+ \\
 & & \mathcal{B}[\![e]\!]\ \rho' +\!\!+ [\texttt{Pop}\ n] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\mathcal{B}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho & = & \mathcal{B}[\![e_1]\!]\ \rho +\!\!+ \mathcal{B}[\![e_0]\!]\ \rho +\!\!+ [\texttt{Add}] \\
 & & \qquad \text{And similarly for other arithmetic expressions} \\
\mathcal{B}[\![e_0\ \texttt{==}\ e_1]\!]\ \rho & = & \mathcal{B}[\![e_1]\!]\ \rho +\!\!+ \mathcal{B}[\![e_0]\!]\ \rho +\!\!+ [\texttt{Eq}] \\
 & & \qquad \text{And similarly for other comparison expressions} \\
\mathcal{B}[\![\texttt{negate}\ e]\!]\ \rho & = & \mathcal{B}[\![e]\!]\ \rho +\!\!+ [\texttt{Neg}] \\
\mathcal{B}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\ \rho & = & \mathcal{B}[\![e_0]\!]\ \rho +\!\!+ [\texttt{Cond}\ (\mathcal{B}[\![e_1]\!]\ \rho)\ (\mathcal{B}[\![e_2]\!]\ \rho)] \\
\mathcal{B}[\![e]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Get}] \qquad\qquad \text{the default case}
\end{array}
$$

Figure 3.18: The $\mathcal{B}$ compilation scheme

The $\mathcal{B}$ compilation scheme

The $\mathcal{B}$ scheme, shown in Figure 3.18, constitutes another type of context. To be compiled by the $\mathcal{B}$ scheme, an expression must not only be known to need evaluation to WHNF, it must also be an expression of type integer or boolean. The following expressions will propagate through the $\mathcal{B}$ scheme:

- If `let(rec)` Δ `in` e occurs in a $\mathcal{B}$-strict context then the expression e is also in a $\mathcal{B}$-strict context.
- If the expression `if` e_0 e_1 e_2 occurs in a $\mathcal{B}$-strict context then the expressions e_0, e_1 and e_2 also occur in $\mathcal{B}$-strict contexts.
- If $e_0 \odot e_1$ occurs in a $\mathcal{B}$-strict context, with $\odot$ being a comparison or arithmetic operator, then the expressions e_0 and e_1 also occur in $\mathcal{B}$-strict contexts.
- If `negate` e occurs in a $\mathcal{B}$-strict context then so does the expression e.

If we cannot recognise any of the special cases of expression, then the compiled code will evaluate the expression using the $\mathcal{E}$ scheme, and then perform a `Get` instruction. The `Get` instruction unboxes the value left on top of the stack by the $\mathcal{E}$ scheme and moves it to the V-stack.

This has left unspecified how we know that an expression is initially in a $\mathcal{B}$-strict context. The usual situation is that we generate a $\mathcal{B}$-strict context from the usual strict context: $\mathcal{E}$, with the additional knowledge that the value is of type integer or boolean.

Exercise 3.41. Implement the $\mathcal{B}$ compiler scheme.

The $\mathcal{E}$ compilation scheme

The $\mathcal{E}$ scheme – defined in Figure 3.19 – specifies that we should give special treatment to arithmetic and comparison function calls. It differs from the version we used for the Mark 6 machine because we make use of the $\mathcal{B}$ scheme to perform the arithmetic and comparison operations. Again, if there are no special cases, then we must use a default method to compile the expression. This is simply to build the graph, using the $\mathcal{C}$ scheme, and then place an `Eval` instruction in the code stream. This will ensure that the graph is evaluated to WHNF.

Exercise 3.42. Implement the new $\mathcal{E}$ compiler scheme.

The $\mathcal{R}$ compilation scheme

We also take this opportunity to improve the $\mathcal{R}$ scheme. Firstly, we wish to create opportunities for the $\mathcal{B}$ scheme to be used. We are also attempting to reduce the number of instructions that are executed at the end of a function's code sequence. The new compilation schemes for the $\mathcal{R}$ scheme are given in Figure 3.20. It has been expanded from the version we used in the Mark 6 machine; in that it now works like a *context*. We refer to this context as $\mathcal{R}$-strict. An expression being compiled in a $\mathcal{R}$-strict context will be evaluated to WHNF, and it will then be used to overwrite the current redex. It obeys the following rules of propagation.

- The expression in the body of a supercombinator definition is in an $\mathcal{R}$-strict context.
- If `let(rec)` Δ `in` e occurs in an $\mathcal{R}$-strict context then the expression e is also in an $\mathcal{R}$-strict context.
- If the expression `if` e_0 e_1 e_2 occurs in an $\mathcal{R}$-strict context then the expressions e_1 and e_2 are also in an $\mathcal{R}$-strict context. (The expression e_0 will now appear in a $\mathcal{B}$-strict context.)
- If `case` e `of` *alts* occurs in an $\mathcal{R}$-strict context then the expression e is in a strict context. Furthermore, the expression part of each alternative will occur in an $\mathcal{R}$-strict context.

Exercise 3.43. Implement the $\mathcal{R}$ scheme in Figure 3.20. Note that you can use the generality of the `compileAlts` function to implement both the $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{E}$ schemes.

One point worth noting about the Mark 7 machine is that we did not define the `Eval` instruction to save the current V-stack on the dump. This is in contrast to the G-machine described in [Peyton Jones 1987]. Whether you do this is really a matter of taste in the abstract machines we have produced in this book. In compiling code for a real machine we would be likely to try to minimise the number of stacks. When this is the case, we will wish to use the following alternative transition for `Eval`.

$\mathcal{E}[\![e]\!]\ \rho$ compiles code that evaluates an expression e to WHNF in environment ρ, leaving a pointer to the expression on top of the stack.

$$
\begin{array}{rcl}
\mathcal{E}[\![i]\!]\ \rho & = & [\texttt{Pushint}\ i] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{let}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ \\
 & & \mathcal{E}[\![e]\!]\ \rho' +\!\!+ [\texttt{Slide}\ n] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{letrec}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho} \\
 & = & [\texttt{Alloc}\ n] +\!\!+ \\
 & & \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{Update}\ n-1] +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{Update}\ 0] +\!\!+ \\
 & & \mathcal{E}[\![e]\!]\ \rho' +\!\!+ [\texttt{Slide}\ n] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Casejump}\ \mathcal{D}_{\mathcal{E}}[\![alts]\!]\ \rho] \\
\mathcal{E}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}\ e_1\ \ldots\ e_a]\!]\ \rho & = & \mathcal{C}[\![e_a]\!]\ \rho +\!\!+ \ldots \mathcal{C}[\![e_1]\!]\ \rho +\!\!+ [\texttt{Pack}\ t\ a] \\
\mathcal{E}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho & = & \mathcal{B}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho +\!\!+ [\texttt{Mkint}] \\
 & & \qquad\qquad \text{And similarly for other arithmetic expressions} \\
\mathcal{E}[\![e_0\ \texttt{==}\ e_1]\!]\ \rho & = & \mathcal{B}[\![e_0\ \texttt{==}\ e_1]\!]\ \rho +\!\!+ [\texttt{Mkbool}] \\
 & & \qquad\qquad \text{And similarly for other comparison expressions} \\
\mathcal{E}[\![\texttt{negate}\ e]\!]\ \rho & = & \mathcal{B}[\![\texttt{negate}\ e]\!]\ \rho +\!\!+ [\texttt{Mkint}] \\
\mathcal{E}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\ \rho & = & \mathcal{B}[\![e_0]\!]\ \rho +\!\!+ [\texttt{Cond}\ (\mathcal{E}[\![e_1]\!]\ \rho)\ (\mathcal{E}[\![e_2]\!]\ \rho)] \\
\mathcal{E}[\![e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho +\!\!+ [\texttt{Eval}] \qquad\qquad \text{the default case}
\end{array}
$$

$\mathcal{D}_{\mathcal{E}}[\![alts]\!]\ \rho$ compiles the code for the alternatives in a `case` expression.

$$
\mathcal{D}_{\mathcal{E}}[\![alt_1\ \ldots\ alt_n]\!]\ \rho \ = \ [\mathcal{A}_{\mathcal{E}}[\![alt_1]\!]\ \rho,\ \ldots,\ \mathcal{A}_{\mathcal{E}}[\![alt_n]\!]\ \rho]
$$

$\mathcal{A}_{\mathcal{E}}[\![alt]\!]\ \rho$ compiles the code for an alternative in a `case` expression.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathcal{A}_{\mathcal{E}}[\![\texttt{<}t\texttt{>}\ x_1 \ldots x_n\ \texttt{->}\ body]\!]\ \rho} \\
 & = & t\ \texttt{->}\ [\texttt{Split}\ n] +\!\!+ \mathcal{E}[\![body]\!]\ \rho' +\!\!+ [\texttt{Slide}\ n] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto 0 \ldots x_n \mapsto n-1]
\end{array}
$$

Figure 3.19: The Mark 7 $\mathcal{E}$, $\mathcal{D}_{\mathcal{E}}$ and $\mathcal{A}_{\mathcal{E}}$ compilation schemes

$\mathcal{R}[\![e]\!]\ \rho\ d$ generates code which instantiates the expression e in environment ρ, for a supercombinator of arity d, and then proceeds to unwind the resulting stack.

$$
\begin{array}{rcl}
\mathcal{R}[\![\texttt{let}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho\ d & & \\
& = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
& & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ \\
& & \mathcal{R}[\![e]\!]\ \rho'\ (n+d) \\
& & \qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\mathcal{R}[\![\texttt{letrec}\ x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n\ \texttt{in}\ e]\!]\ \rho\ d & & \\
& = & [\texttt{Alloc}\ n] +\!\!+ \\
& & \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{Update}\ n-1] +\!\!+ \ldots +\!\!+ \\
& & \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{Update}\ 0] +\!\!+ \\
& & \mathcal{R}[\![e]\!]\ \rho'(n+d) \\
& & \qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0] \\
\mathcal{R}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\ \rho\ d & = & \mathcal{B}[\![e_0]\!]\ \rho +\!\!+ [\texttt{Cond}\ [\mathcal{R}[\![e_1]\!]\ \rho\ d,\ \mathcal{R}[\![e_2]\!]\ \rho\ d]] \\
\mathcal{R}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho\ d & = & \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Casejump}\ \mathcal{D}_{\mathcal{R}}[\![alts]\!]\ \rho\ d] \\
\mathcal{R}[\![e]\!]\ \rho\ d & = & \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Update}\ d, \texttt{Pop}\ d, \texttt{Unwind}] \qquad \text{the default case}
\end{array}
$$

$\mathcal{D}_{\mathcal{R}}[\![alts]\!]\ \rho\ d$ compiles the code for the alternatives in a `case` expression.

$$
\mathcal{D}_{\mathcal{R}}[\![alt_1\ \ldots\ alt_n]\!]\ \rho\ d \ = \ [\mathcal{A}_{\mathcal{R}}[\![alt_1]\!]\ \rho\ d,\ \ldots,\ \mathcal{A}_{\mathcal{R}}[\![alt_n]\!]\ \rho\ d]
$$

$\mathcal{A}_{\mathcal{R}}[\![alt]\!]\ \rho\ d$ compiles the code for an alternative in a `case` expression.

$$
\begin{array}{rcl}
\mathcal{A}_{\mathcal{R}}[\![\texttt{<}t\texttt{>}\ x_1 \ldots x_n\ \texttt{->}\ body]\!]\ \rho\ d & & \\
& = & t\ \texttt{->}\ [\texttt{Split}\ n] +\!\!+ \mathcal{R}[\![body]\!]\ \rho'\ (n+d) \\
& & \qquad \text{where } \rho' = \rho^{+n}[x_1 \mapsto 0 \ldots x_n \mapsto n-1]
\end{array}
$$

Figure 3.20: The Mark 7 $\mathcal{R}$, $\mathcal{D}_{\mathcal{R}}$, and $\mathcal{A}_{\mathcal{R}}$ compilation schemes

$$
(3.48)\quad
\begin{array}{|cclcllll|}
\hline
 & o & \texttt{Eval} : i & a : s & d & v & h & m \\
\Longrightarrow & o & [\texttt{Unwind}] & [a] & \langle i,\ s,\ v\rangle : d & [] & h & m \\
\hline
\end{array}
$$

Exercise 3.44. Implement this alternative transition for `Eval`. What are the other instructions that will need to be changed to allow for this new transition?

Exercise 3.45. What compilation rules would you change to produce optimised code for the boolean operators `&`, `|` and `not`?

Exercise 3.46. Add a `Return` instruction with transition:

$$
(3.49)\quad
\begin{array}{|ccccllll|}
\hline
 & o & [\texttt{Return}] & [a_0, \ldots, a_k] & \langle i,\ s\rangle : d & v & h & m \\
\Longrightarrow & o & i & a_k : s & d & v & h & m \\
\hline
\end{array}
$$

This is used by the $\mathcal{R}$ scheme in place of `Unwind` when the item on top of the stack is known to be in WHNF. Modify `compileR` to generate this new instruction.

Exercise 3.47. Write out the transition for `UpdateInt n`. This instruction performs the same actions as the sequence `[Mkint, Update n]`. Implement this transition and a similar one for `UpdateBool n`. Modify `compileR` to generate these instructions instead of the original sequences.

Why are the new instructions preferred to the original sequences? (Hint: use the statistics from some sample programs.)

3.10 Conclusions

The approach that we have taken in this chapter is a very useful one when designing large pieces of software. First we start with something very simple, and then by a number of gradual changes we produce a very large and complicated piece of software. It would be misleading to claim that this is always possible by a process of small, incremental, changes. In fact the material presented as part of the Mark 1 machine was specifically designed with the Mark 7 machine in mind.

For the moment, however, we have produced a reasonably efficient machine. In the next chapter we will look at the TIM.

```
> module Tim where
> import Utils
> import Language
```