Part V

Control and Data Flow

Chapter 11

Abstract Machines

Long considered to be a topic of primarily academic interest, *abstract*, or *virtual*, *machines* are now attracting renewed attention, especially by the software industry. The main idea is to define an instruction set for a "pseudo-computer", the abstract machine, that may be used as the object code for compiling a high-level language (such as ML or Java) and that may be implemented with reasonable efficiency on a wide variety of stock platforms. This means that the high-level language must be implemented only once, for the abstract machine, but that the abstract machine must itself be implemented once per platform. One advantage is that it is, in principle, much easier to port the abstract machine than it is to re-implement the language for each platform. More importantly, this architecture supports the exchange of object code across the network — if everyone implements the abstract machine, then code can migrate from one computer to another without modification. Web sites all over the world exploit this capability to tremendous advantage, using the Java Virtual Machine.

Before we get started, let us ask ourselves the question: what is an abstract machine? In other words, what is a computer? The fundamental idea of computation is the notion of step-by-step execution of *instructions* that transform the *state* of the computer in some determinate fashion.[1] Each instruction should be executable in a finite amount of time using a finite amount of information, and it should be clear how to effect the required state transformation using only physically realizable methods.[2] Execution

[1]The question of determinacy is increasingly problematic for real computers, largely because of the aggressive use of parallelism in their implementation. We will gloss over this issue here.

[2]For example, consider the instruction that, given the representation of a program, sets register zero to one iff there is an input on which that program halts when executed, and

of a program consists of initializing the machine to a known *start state*, executing instructions one-by-one until no more instructions remains; the result of the computation is the *final state*. Thus an abstract machine is essentially a *transition system* between states of that machine.

According to this definition the dynamic semantics of MinML is an abstract machine, the M machine. The states of the M machine are closed MinML expressions e, and the transitions are given by the one-step evaluation relation $e \mapsto_{\mathsf{M}} e'$ defined earlier. This machine is quite high-level in the sense that the instructions are fairly complex compared to what are found in typical concrete machines. For example, the M machine performs substitution of a value for a variable in one step, a decidedly large-scale (but nevertheless finite and effective) instruction. This machine is also odd in another sense: rather than have an analogue of a program counter that determines the next instruction to be executed, we instead have "search rules" that traverse the expression to determine what to do next. As you have no doubt observed, this can be quite an involved process, one that is not typical of real computers. We will begin to address these concerns by first looking at the management of the flow of control in an abstract machine, and then considering the management of bindings of values to variables.

11.1 Control Flow

Rather than repeatedly traverse an expression looking for the next instruction to execute, we can maintain an explicit record of what to do next in the computation using an abstract *control stack* that maintains a record of the work remaining to be done (in reverse order) to finish evaluating an expression. We will call this machine the C *machine*, to remind us that it is defined to capture the idea of control flow in a computation.

The states of the C machine have the form (k, e), where k is a control stack and e is a closed expression. Control stacks are inductively defined by the following rules:

$$\frac{}{\bullet\ \mathsf{stack}} \tag{11.1}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{f \triangleright k\ \mathsf{stack}} \tag{11.2}$$

sets it to zero otherwise. This instruction could not be regarded as the instruction of any computing device that we could ever physically realize, because of the unsolvability of the halting problem.

The set of *stack frames* is inductively defined by these rules:

$$\frac{e_2 \ \mathsf{expr}}{+(\Box, e_2) \ \mathsf{frame}} \tag{11.3}$$

$$\frac{v_1 \ \mathsf{value}}{+(v_1, \Box) \ \mathsf{frame}} \tag{11.4}$$

(There are analogous frames associated with the other primitive operations.)

$$\frac{e_1 \ \mathsf{expr} \quad e_2 \ \mathsf{expr}}{\texttt{if} \, \Box \, \texttt{then} \, e_1 \, \texttt{else} \, e_2 \, \texttt{fi} \ \mathsf{frame}} \tag{11.5}$$

$$\frac{e_2 \ \mathsf{expr}}{\texttt{apply(}\Box, e_2\texttt{)} \ \mathsf{frame}} \tag{11.6}$$

$$\frac{v_1 \ \mathsf{value}}{\texttt{apply(}v_1, \Box\texttt{)} \ \mathsf{frame}} \tag{11.7}$$

Thus a control stack is a sequence of frames $f_1 \triangleright \cdots f_n \triangleright \bullet$ (implicitly right-associated), where $\bullet$ is the empty stack and each f_i $(1 \leq i \leq n)$ is a stack frame. Each stack frame represents one step in the process of searching for the next position to evaluate in an expression.

The transition relation for the C machine is inductively defined by a set of transition rules. We begin with the rules for addition; the other primitive operations are handled similarly.

$$(k, +(e_1, e_2)) \mapsto_{\mathsf{C}} (+(\Box, e_2) \triangleright k, e_1) \tag{11.8}$$

$$(+(\Box, e_2) \triangleright k, v_1) \mapsto_{\mathsf{C}} (+(v_1, \Box) \triangleright k, e_2) \tag{11.9}$$

$$(+(n_1, \Box) \triangleright k, n_2) \mapsto_{\mathsf{C}} (k, n_1 + n_2) \tag{11.10}$$

The first two rules capture the left-to-right evaluation order for the arguments of addition. The top stack frame records the current position within the argument list; when the last argument has been evaluated, the operation is applied and the stack is popped.

Next, we consider the rules for booleans.

$$(k, \texttt{if} \, e \, \texttt{then} \, e_1 \, \texttt{else} \, e_2 \, \texttt{fi}) \mapsto_{\mathsf{C}} (\texttt{if} \, \Box \, \texttt{then} \, e_1 \, \texttt{else} \, e_2 \, \texttt{fi} \triangleright k, e) \tag{11.11}$$

$$(\texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \triangleright k, \texttt{true}) \mapsto_{\mathsf{C}} (k, e_1) \tag{11.12}$$

$$(\texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \triangleright k, \texttt{false}) \mapsto_{\mathsf{C}} (k, e_2) \tag{11.13}$$

These rules follow the same pattern. First, the test expression is evaluated, recording the pending conditional branch on the stack. Once the value of the test has been determined, we branch to the appropriate arm of the conditional.

Finally, we consider the rules for application of functions.

$$(k, \texttt{apply(}e_1, e_2\texttt{)}) \mapsto_{\mathsf{C}} (\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, e_1) \tag{11.14}$$

$$(\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, v_1) \mapsto_{\mathsf{C}} (\texttt{apply(}v_1, \Box\texttt{)} \triangleright k, e_2) \tag{11.15}$$

$$(\texttt{apply(}v_1, \Box\texttt{)} \triangleright k, v_2) \mapsto_{\mathsf{C}} (k, \{v_1, v_2/f, x\}e) \tag{11.16}$$

The last rule applies in the case that $v_1 = \texttt{fun}\,f\,\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}$. These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated.

The final states of the C machine have the form $(v, \bullet)$ consisting of the empty stack (no further work to do) and a value v.

The rules defining the C machine have *no premises* — they are all simple transitions, without any hypotheses. We've made explicit the management of the "subgoals" required for evaluating expressions using the M machine by introducing a stack of pending sub-goals that specifies the order in which they are to be considered. In this sense the C machine is less abstract than the M machine. It is interesting to examine your implementation of the M machine, and compare it to an implementation of the C machine. The M machine implementation makes heavy use of the ML runtime stack to implement the recursive calls to the MinML interpreter corresponding to premises of the evaluation rules. The runtime stack is required because the interpreter is *not* a tail recursive function. In contrast an implementation of the C machine *is* tail recursive, precisely because there are no premises on any of the transitions rules defining it.

What is the relationship between the M machine and the C machine? Do they define the same semantics for the MinML language? Indeed they do, but a rigorous proof of this fact is surprisingly tricky to get right. The hardest part is to figure out how to state the correspondence precisely; having done that, the verification is not difficult.

The first step is to define a correspondence between C machine states and M machine states. Intuitively the control stack in the C machine corresponds to the "surrounding context" of an expression, which is saved for consideration once the expression has been evaluated. Thus a C machine state may be thought of as representing the M machine state obtained by "unravelling" the control stack and plugging in the current expression to reconstruct the entire program as a single expression. The function that does this, written $k\ @\ e$, is defined by induction on the structure of k as follows:

$$\begin{array}{rcl}
\bullet\ @\ e & = & e \\
+(\square, e_2) \triangleright k\ @\ e_1 & = & k\ @\ +(e_1, e_2) \\
+(v_1, \square) \triangleright k\ @\ e_2 & = & k\ @\ +(v_1, e_2) \\
\texttt{if}\,\square\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \triangleright k\ @\ e & = & k\ @\ \texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \\
\texttt{apply(}\square, e_2\texttt{)} \triangleright k\ @\ e & = & k\ @\ \texttt{apply(}e, e_2\texttt{)} \\
\texttt{apply(}v_1, \square\texttt{)} \triangleright k\ @\ e & = & k\ @\ \texttt{apply(}v_1, e\texttt{)}
\end{array}$$

The precise correspondence between the two machines is given by the following theorem.

Theorem 16

1. *If* $(k, e) \mapsto_{\mathsf{C}} (k', e')$, *then either* $k\ @\ e = k'\ @\ e'$, *or* $k\ @\ e \mapsto_{\mathsf{M}} k'\ @\ e'$.

2. *If* $e \mapsto_{\mathsf{M}} e'$ *and* $(k, e') \mapsto^*_{\mathsf{C}} (\bullet, v)$, *then* $(k, e) \mapsto^*_{\mathsf{C}} (\bullet, v)$.

The first part of the Theorem states that the C machine transitions are either "bookkeeping" steps that move a piece of the program onto the control stack without materially changing the overall program, or "instruction" steps that correspond to transitions in the M machine. The second part is a bit tricky to understand, at first glance. It says that if the M machine moves from a state e to a state e', and the C machine runs to completion starting from e' and an arbitrary stack k, then it also runs to completion starting from e and k.[3]

Proof:

1. By induction on the definition of the C machine. We will do the cases for application here; the remainder follow a similar pattern.

[3]Half the battle in establishing a correspondence between the two machines was to find the proper statement of the correspondence! So you should not be dismayed if it takes some time to understand what is being said here, and why.

(a) Consider the transition

$$(k, \texttt{apply(}e_1, e_2\texttt{)}) \mapsto_{\mathsf{C}} (\texttt{apply(}\square, e_2\texttt{)} \triangleright k, e_1).$$

Here $e = \texttt{apply(}e_1, e_2\texttt{)}$, $k' = \texttt{apply(}\square, e_2\texttt{)} \triangleright k$, and $e' = e_1$. It is easy to check that $k \mathbin{@} e = k' \mathbin{@} e'$.

(b) Consider the transition

$$(\texttt{apply(}\square, e_2\texttt{)} \triangleright k'', v_1) \mapsto_{\mathsf{C}} (\texttt{apply(}v_1, \square\texttt{)} \triangleright k'', e_2).$$

Here $e = v_1$, $k = \texttt{apply(}\square, e_2\texttt{)} \triangleright k''$, $e' = e_2$, and $k' = \texttt{apply(}v_1, \square\texttt{)} \triangleright k''$. It is easy to check that $k \mathbin{@} e = k' \mathbin{@} e'$.

(c) Consider the transition

$$(\texttt{apply(}v_1, \square\texttt{)} \triangleright k', v_2) \mapsto_{\mathsf{C}} (k', \{v_1, v_2/f, x\}e),$$

where $v_1 = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_2\texttt{):}\tau\ \texttt{is}\ e\ \texttt{end}$. Here $k = \texttt{apply(}v_1, \square\texttt{)} \triangleright k'$, $e = v_2$, and $e' = \{v_1, v_2/f, x\}e$. We have

$$\begin{array}{rcl} k \mathbin{@} e & = & k' \mathbin{@} \texttt{apply(}v_1, v_2\texttt{)} \\ & \mapsto & k' \mathbin{@} e' \end{array}$$

as desired. The second step follows from the observation that stacks are defined so that the M search rules "glide over" k' — the next instruction to execute in $k' \mathbin{@} \texttt{apply(}v_1, v_2\texttt{)}$ must be the application $\texttt{apply(}v_1, v_2\texttt{)}$.

2. By induction on the MinML dynamic semantics. We will do the cases for application here; the remainder follow a similar pattern.

(a) $e = \texttt{apply(}v_1, v_2\texttt{)} \mapsto_{\mathsf{M}} \{v_1, v_2/f, x\}e_2 = e'$, where the value $v_1 = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_2\texttt{):}\tau\ \texttt{is}\ e_2\ \texttt{end}$. Suppose that $(k, e') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$. By the definition of the C machine transition relation,

$$\begin{array}{rcl} (k, e) & \mapsto_{\mathsf{C}} & (\texttt{apply(}\square, v_2\texttt{)} \triangleright k, v_1) \\ & \mapsto_{\mathsf{C}} & (\texttt{apply(}v_1, \square\texttt{)} \triangleright k, v_2) \\ & \mapsto_{\mathsf{C}} & (k, e') \end{array}$$

From this, the result follows immediately.

(b) $e = \texttt{apply(}e_1, e_2\texttt{)} \mapsto_{\mathsf{M}} \texttt{apply(}e_1', e_2\texttt{)} = e'$, where $e_1 \mapsto_{\mathsf{M}} e_1'$. Suppose that $(k, e') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$. Since $e' = \texttt{apply(}e_1', e_2\texttt{)}$, and since the C machine is deterministic, this transition sequence must have the form

$$(k, e') = (k, \texttt{apply(}e_1', e_2\texttt{)}) \mapsto_{\mathsf{C}} (\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, e_1') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$$

By the inductive hypothesis, using the enlarged stack, it follows that

$$(\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, e_1) \mapsto_{\mathsf{C}}^{*} (\bullet, v).$$

Now since

$$(k, e) = (k, \texttt{apply(}e_1, e_2\texttt{)}) \mapsto_{\mathsf{C}} (\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, e_1)$$

the result follows immediately.

(c) $e = \texttt{apply(}v_1, e_2\texttt{)} \mapsto_{\mathsf{M}} \texttt{apply(}v_1, e_2'\texttt{)} = e'$, where $e_2 \mapsto_{\mathsf{M}} e_2'$. Suppose that $(k, e') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$. Since $e' = \texttt{apply(}v_1, e_2'\texttt{)}$, and since the C machine is deterministic, this transition sequence must have the form

$$(k, e') = (k, \texttt{apply(}v_1, e_2'\texttt{)}) \mapsto_{\mathsf{C}} (\texttt{apply(}v_1, \Box\texttt{)} \triangleright k, e_2') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$$

By the inductive hypothesis, using the enlarged stack, it follows that

$$(\texttt{apply(}v_1, \Box\texttt{)} \triangleright k, e_2) \mapsto_{\mathsf{C}}^{*} (\bullet, v).$$

Now since

$$(k, e) = (k, \texttt{apply(}v_1, e_2\texttt{)}) \mapsto_{\mathsf{C}} (\texttt{apply(}v_1, \Box\texttt{)} \triangleright k, e1)$$

the result follows immediately.

■

Exercise 17
Finish the proof of the theorem by giving a complete proof of part (1), and filling in the missing cases in part (2).

Corollary 18

1. *If* $(k, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*, then* $k \mathbin{@} e \mapsto_{\mathsf{M}}^{*} v$*. Hence if* $(\bullet, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*, then* $e \mapsto_{\mathsf{M}}^{*} v$*.*

2. *If* $e \mapsto_{\mathsf{M}}^{*} e'$ *and* $(k, e') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*, then* $(k, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*. Hence if* $e \mapsto_{\mathsf{M}}^{*} v$*, then* $(\bullet, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*.*

Proof:

1. By induction on the transition sequence, making use of part (1) of the theorem, then taking $k = \bullet$. For the induction we have two cases to consider, one for each rule defining multi-step transition:

   (a) Reflexivity. In this case $k = \bullet$ and $e = v$. It follows that $k \mathbin{@} e = v \mapsto^* v$, as required.

   (b) Reverse execution. Here we have $(k', e') \mapsto_{\mathsf{C}} (k, e) \mapsto^*_{\mathsf{C}} (\bullet, v)$. By induction $k \mathbin{@} e \mapsto^*_{\mathsf{M}} v$, and by Theorem 16 $k' \mathbin{@} e' \mapsto^*_{\mathsf{M}} k \mathbin{@} e$, so $k' \mathbin{@} e' \mapsto^*_{\mathsf{M}} v$.

2. By induction on transition sequence, making use of part (2) of the theorem, then taking $e' = v$ and $k = \bullet$. We have two cases:

   (a) Reflexivity. In this case $e = e'$ and the result is immediate.

   (b) Reverse execution. Here $e \mapsto_{\mathsf{M}} e'' \mapsto^*_{\mathsf{M}} e'$ and $(k, e') \mapsto^*_{\mathsf{C}} (\bullet, v)$. By induction $(k, e'') \mapsto^*_{\mathsf{C}} (\bullet, v)$ and by Theorem 16 we have $(k, e) \mapsto^*_{\mathsf{C}} (\bullet, v)$, as required.

■

To facilitate comparison with the E machine described below, it is useful to restructure the C machine in the following manner. First, we introduce an "auxiliary" state of the form (v, k), which represents the process of passing the value v to the stack k. Second, we "link" these two states by the transition rule

$$(k, v) \mapsto_{\mathsf{C}} (v, k). \tag{11.17}$$

That is, when encountering a value, pass it to the stack. Finally, we modify the transition relation so that all analysis of the stack is performed using the auxiliary state. Note that transitions now have one of four forms:

$$\begin{array}{lll l}
(k, e) & \mapsto_{\mathsf{C}} & (k', e') & \textit{process expression} \\
(k, v) & \mapsto_{\mathsf{C}} & (v, k) & \textit{pass value to stack} \\
(v, k) & \mapsto_{\mathsf{C}} & (v', k') & \textit{pass value up stack} \\
(v, k) & \mapsto_{\mathsf{C}} & (k', e') & \textit{process pending expression}
\end{array}$$

Exercise 19
Complete the suggested re-formulation of the C machine, and show that it is equivalent to the orginal formulation.

11.2 Environments

The C machine is still quite "high level" in that function application is performed by substitution of the function itself and its argument into the body of the function, a rather complex operation. This is unrealistic for two reasons. First, substitution is a complicated process, not one that we would ordinarily think of as occurring as a single step of execution of a computer. Second, and perhaps more importantly, the use of substitution means that the program itself, and not just the data it acts upon, changes during evaluation. This is a radical departure from more familiar models of computation, which maintain a rigorous separation between program and data. In this section we will present another abstraction machine, the E machine, which avoids substitution by introducing an *environment* that records the bindings of variables.

The basic idea is simple: rather than *replace* variables by their bindings when performing a function application, we instead *record* the bindings of variables in a data structure, and, correspondingly, *look up* the bindings of variables when they are used. In a sense we are performing substitution "lazily", rather than "eagerly", to avoid unnecessary duplication and to avoid modifying the program during execution. The main complication introduced by environments is that we must exercise considerable caution to ensure that we do not confuse the scopes of variables.[4] It is remarkably easy, if we are not careful, to confuse the bindings of variables that happen to have the same name. We avoid difficulties by introducing *closures*, data structures that package an expression together with an environment.

To see the point, let's first sketch out the structure of the E machine. A state of the E machine has the form (K, E, e), where K is a *machine stack*, E is an *environment*, a finite function mapping variables to *machine values*, and e is an open expression such that $\mathrm{FV}(e) \subseteq \mathrm{dom}(E)$. Machine values are values "inside the machine", distinct from the syntactic notion of value used in the M and C machines. The reason for the distinction arises from the replacement of substitution by binding.

Since the M and C machines perform function application by substitution, there is never any need to consider expressions with free variables in them; the invariant that the expression part of the state is closed is maintained throughout evaluation. The whole point of the E machine, however, is to avoid substitution by maintaining an environment that records

[4]In fact, the notion of "dynamic scope" arose as a result of an error in the original Lisp interpreter (circa 1960) that confused the scopes of variables.

the bindings of free variables. When a function is called, the parameter is bound to the argument, the function name is bound to the function itself, and the body is evaluated; when that is complete the bindings of the function name and parameter can be released, and evaluation continues.

This suggests that the environment is a global, stack-like data structure onto which arguments are pushed and popped during evaluation — values are pushed on function call and popped on function return. In fact, the environment might be called the *data stack* for precisely this reason. However, a moment's thought reveals that this characterization is a tad too simplistic, because it overlooks a crucial issue in the implementation of functional languages, namely the ability to return functions as results of function applications. Suppose that f is a function of type $\texttt{int}\to\texttt{int}\to\texttt{int}$. When applied to an integer n, the result $\texttt{apply(}f,n\texttt{)}$ yields a function of type $\texttt{int}\to\texttt{int}$. For example, f might be the following function:

$$\texttt{fun}\,\_\,\texttt{(}x\texttt{:int):int}\to\texttt{int is fun}\,\_\,\texttt{(}y\texttt{:int):int is}\ x\ \texttt{end end},$$

Observe that the function returned by f contains a free occurrence of the parameter x of f. If we follow the simple stack-like discipline of function call and return, we will, upon calling f, bind x to 1, yielding the value

$$\texttt{fun}\,\_\,\texttt{(}y\texttt{:int):int is}\ x\ \texttt{end},$$

then pop the binding of x from the environment. *But wait a minute!* The returned value is a function that contains a free occurrence of x, and we've just deleted the binding for x from the environment! Subsequent uses of this function will either capture some other binding for x that happens to be in the environment at the time it is used, violating the static scoping principle,[5], or incur an unbound variable error if no binding for x happens to be available.

This problem is avoided by the use of *closures*. The value returned by the application $\texttt{apply(}f,1\texttt{)}$ is the closure[6]

$$\texttt{fun}\,\_\,\texttt{(}y\texttt{:int):int is}\ x\ \texttt{end}[E[x\mapsto 1]]$$

where E is the environment in effect at the point of the call. When f returns the binding for x is indeed popped from the *global* environment, but a *local*

[5]This is the error in the original implementation of Lisp referred to earlier.

[6]In this case the rest of the environment, E, is superfluous. In general we can cut down the closing environment to just those variables that actually occur in the body of the function. We will ignore this optimization for the time being.

copy of it is retained in the closure returned by f. This way no confusion or capture is possible, and the static scoping discipline is maintained, even in the absence of substitution.

The need for closures motivates the distinction between syntactic values and machine values. The latter are inductively defined by the following rules:

$$\frac{}{n\ \mathsf{mvalue}} \qquad \frac{}{\texttt{true}\ \mathsf{mvalue}} \tag{11.18}$$

$$\frac{}{\texttt{true}\ \mathsf{mvalue}} \tag{11.19}$$

$$\frac{}{\texttt{false}\ \mathsf{mvalue}} \tag{11.20}$$

$$\frac{x\ \mathsf{var} \quad y\ \mathsf{var} \quad e\ \mathsf{expr}}{\texttt{fun}\,x\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}[E]\ \mathsf{mvalue}} \tag{11.21}$$

An *environment*, E, is a finite function mapping variables to machine values.

The set of *machine stacks* is inductively defined by the following rules:

$$\frac{}{\bullet\ \mathsf{mstack}} \tag{11.22}$$

$$\frac{F\ \mathsf{mframe} \quad K\ \mathsf{mstack}}{F \triangleright K\ \mathsf{mstack}}\ . \tag{11.23}$$

Here F is a *machine frame*. The set of machine frames is inductively defined by these rules:

$$\frac{e_2\ \mathsf{expr}}{+(\Box, e_2)[E]\ \mathsf{mframe}} \tag{11.24}$$

$$\frac{V_1\ \mathsf{mvalue}}{+(V_1, \Box)\ \mathsf{mframe}} \tag{11.25}$$

$$\frac{e_1\ \mathsf{expr} \quad e_2\ \mathsf{expr}}{\texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}[E]\ \mathsf{mframe}} \tag{11.26}$$

$$\frac{e_2\ \mathsf{expr}}{\texttt{apply(}\Box, e_2\texttt{)}[E]\ \mathsf{mframe}} \tag{11.27}$$

$$\frac{V_1\ \mathsf{mvalue}}{\texttt{apply(}V_1, \Box\texttt{)}\ \mathsf{mframe}} \tag{11.28}$$

The notation for E machine frames is deceptively similar to the notation for C machine frames. Note, however, that E machine frames involve machine values, and that in many cases the frame is closed with respect to an environment recording the bindings of the free variables in the expressions stored in the frame. The second form of addition and application frames need no environment; do you see why?

The E machine has two kinds of states: (K, E, e), described earlier, and "auxiliary" states of the form (V, K), where K is a machine stack and V is a machine value. The auxiliary state represents the passage of a machine value to the top frame of the machine stack. (In the C machine this is accomplished by simply filling the hole in the stack frame, but here a bit more work is required.)

The E machine is inductively defined by a set of rules for transitions of one of the following four forms:

$$\begin{array}{rcll} (K, E, e) & \mapsto_{\mathsf{E}} & (K', E', e') & \textit{process expression} \\ (K, E, v) & \mapsto_{\mathsf{E}} & (V', K') & \textit{pass value to stack} \\ (V, K) & \mapsto_{\mathsf{E}} & (V', K') & \textit{pass value up stack} \\ (V, K) & \mapsto_{\mathsf{E}} & (K', E', e') & \textit{process pending expression} \end{array}$$

We will use the same transition relation for all four cases, relying on the form of the states to disambiguate which is intended.

To evaluate a variable x, we look up its binding and pass the associated value to the top frame of the control stack.

$$(K, E, x) \mapsto_{\mathsf{E}} (E(x), K) \tag{11.29}$$

Similarly, to evaluate numeric or boolean constants, we simply pass them to the control stack.

$$(K, E, n) \mapsto_{\mathsf{E}} (n, K) \tag{11.30}$$

$$(K, E, \texttt{true}) \mapsto_{\mathsf{E}} (\texttt{true}, K) \tag{11.31}$$

$$(K, E, \texttt{false}) \mapsto_{\mathsf{E}} (\texttt{false}, K) \tag{11.32}$$

To evaluate a function expression, we close it with respect to the current environment to ensure that its free variables are not inadvertently captured, and pass the resulting closure to the control stack.

$$(K, E, \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}) \mapsto_{\mathsf{E}} (\texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}[E], K) \tag{11.33}$$

To evaluate a primitive operation, we start by evaluating its first argument, pushing a frame on the control stack that records the need to evaluate its remaining arguments.

$$(K, E, +(e_1, e_2)) \mapsto_{\mathsf{E}} (+(\Box, e_2)[E] \triangleright K, E, e_1) \tag{11.34}$$

Notice that the frame is closed in the current environment to avoid capture of free variables in the remaining arguments.

To evaluate a conditional, we evaluate the test expression, pushing a frame on the control stack to record the two pending branches, once again closed with respect to the current environment.

$$(K, E, \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}) \mapsto_{\mathsf{E}} (\texttt{if}\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}[E] \triangleright K, E, e) \tag{11.35}$$

To evaluate an application, we begin by evaluating the function position, pushing a frame to record the pending evaluation of the argument, closed with respect to the current environment.

$$(K, E, \texttt{apply(}e_1, e_2\texttt{)}) \mapsto_{\mathsf{E}} (\texttt{apply(}\Box, e_2\texttt{)}[E] \triangleright K, E, e_1) \tag{11.36}$$

To complete the definition of the E machine, we must define the transitions governing the auxiliary states.

Pending argument evaluations for primitive operations are handled as follows. If more arguments remain to be evaluated, we switch states to process the next argument.

$$(V_1, +(\Box, e_2)[E] \triangleright K) \mapsto_{\mathsf{E}} (+(V_1, \Box) \triangleright K, E, e_2) \tag{11.37}$$

Notice that the environment of the frame is used to evaluate the next argument. If no more arguments remain to be evaluated, we pass the result of executing the primitive operation to the rest of the stack.

$$(n_2, +(n_1, \Box) \triangleright K) \mapsto_{\mathsf{E}} (n_1 + n_2, K) \tag{11.38}$$

Pending conditional branches are handled in the obvious manner.

$$(\texttt{true}, \texttt{if}\,\square\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}[E] \triangleright K) \mapsto_{\mathsf{E}} (K, E, e_1) \tag{11.39}$$

$$(\texttt{false}, \texttt{if}\,\square\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}[E] \triangleright K) \mapsto_{\mathsf{E}} (K, E, e_2) \tag{11.40}$$

Notice that the environment of the frame is restored before evaluating the appropriate branch of the conditional.

Pending function applications are handled as follows.

$$(V, \texttt{apply(}\square, e_2\texttt{)}[E] \triangleright K) \mapsto_{\mathsf{E}} (\texttt{apply(}V, \square\texttt{)} \triangleright K, E, e_2) \tag{11.41}$$

Observe that the environment of the frame is restored before evaluating the argument of the application, and that the function value (which is, presumbly, a closure) is stored intact in the new top frame of the stack.

Once the argument has been evaluated, we call the function.

$$(V_2, \texttt{apply(}V, \square\texttt{)} \triangleright K) \mapsto_{\mathsf{E}} (K, E[f \mapsto V][x \mapsto V_2], e) \tag{11.42}$$

where

$$V = \texttt{fun}\,f\,\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}[E].$$

To call the function we *bind* f to V and x to V_2 in the environment of the closure, continuing with the evaluation of the body of the function. Observe that since we use the environment of the closure, extended with bindings for the function and its parameter, we ensure that the appropriate bindings for the free variables of the function are employed.

The final states of the E machine have the form $(V, \bullet)$, with final result V. Notice that the result is a machine value. If the type of the entire program is `int` or `bool`, then V will be a numeral or a boolean constant, respectively. Otherwise the value will be a closure.

A correspondence between the E and the C machine along the lines of the correspondence between the C machine and the M machine may be established. However, since the technical details are rather involved, we will not pursue a rigorous treatment of the relationship here. Suffice it to say that if e is a closed MinML program of base type (`int` or `bool`), then $(\bullet, e) \mapsto^*_{\mathsf{C}} (\bullet, v)$ iff $(\bullet, \emptyset, e) \mapsto^*_{\mathsf{E}} (v, \bullet)$. (The restriction to base type is necessary if we are to claim that both machines return the *same* value.)

Chapter 12

Continuations

The treatment of exceptions in terms of a handler and control stack relies on the *reification* of control stacks as values that can be pushed on the handler stack. At first glance this appears to be a rather heavyweight operation that would involve copying the entire control stack when establishing a handler, and restoring it when raising an exception. However, we observed that the machine satisfies a crucial invariant, namely that the saved control stack is always an initial segment of the current control stack. This allows us to reify a control stack as a "finger" in the control stack, and to install it by popping the stack back to the finger. This is a formal justification of an implementation based on the `setjmp` and and `longjmp` constructs of the C language. Unlike `setjmp` and `longjmp`, the exception mechanism is completely safe — it is impossible to return past the "finger" yet later attempt to "pop" the control stack to that point. In C the fingers are kept as addresses (pointers) in memory, and there is no discipline for ensuring that the set point makes any sense when invoked later in a computation.

The idea of reification of control stacks can be taken a step further, by allowing them to be passed as values within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *first-class continuations*, where the qualification "first class" stresses that they are ordinary values with an indefinite lifetime that can be passed and returned at will in a computation. In contrast to set points in C first-class continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus first-class continuations support unlimited "time travel" — we can go back to a previous point in the computation and then return to some point in its future, at will.

How is this achieved? The key to implementing first-class continuations is to arrange that control stacks are *persistent* data structures, just like any other data structure in ML that does not involve mutable references. By a persistent data structure we mean one for which operations on it yield a "new" version of the data structure without disturbing the old version. For example, lists in ML are persistent in the sense that if we cons an element to the front of a list we do not thereby destroy the original list, but rather yield a new list with an additional element at the front, retaining the possibility of using the old list for other purposes. In this sense persistent data structures allow time travel — we can easily switch between several versions of a data structure without regard to the temporal order in which they were created. This is in sharp contrast to more familiar *ephemeral* data structures for which operations such as insertion of an element irrevocably mutate the data structure, preventing any form of time travel.

Returning to the case in point, the standard implementation of a control stack is as an ephemeral data structure, a pointer to a region of mutable storage that is overwritten whenever we push a frame. This makes it impossible to maintain an "old" and a "new" copy of the control stack at the same time, making time travel impossible. If, however, we represent the control stack as a persistent data structure, then we can easily reify a control stack by simply binding it to a variable, and continue working. If we wish we can easily return to that control stack by referring to the variable that is bound to it. This is achieved in practice by representing the control stack as a list of frames in the heap so that the persistence of lists can be extended to control stacks. While we will not be specific about implementation strategies in this note, it should be born in mind when considering the semantics outlined below.

Why are first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor. In Chapter 3 we will show how to build a threads package for concurrent programming using continuations.

12.1 Informal Overview of Continuations

We will extend MinML with the type τ cont of continuations accepting values of type τ. A continuation will, in fact, be a control stack of type τ stack, but rather than expose this representation to the programmer, we will regard τ cont as an abstract type supporting two operations, letcc x in e and throw e_1 to e_2.[1]

Informally, evaluation of letcc x in e binds the *current continuation*[2] to x and evaluates e. The current continuation is, as we've discussed, a reification of the current control stack, which represents the current point in the evaluation of the program. The type of x is τ cont, where τ is the type of e. The intuition is that the current continuation is the point to which e returns when it completes evaluation. Consequently, the control stack expects a value of type τ, which then determines how execution proceeds. Thus x is bound to a stack expecting a value of type τ, that is, a value of type τ cont. Note that this is the *only* way to obtain a value of type τ cont; there are no expressions that evaluate to continuations. (This is similar to our treatment of references — values of type τ ref are locations, but locations can only be obtained by evaluating a ref expression.)

We may "jump" to a saved control point by *throwing* a value to a continuation, written throw e_1 to e_2. The expression e_2 must evaluate to a τ_1 cont, and e_1 must evaluate to a value of type τ_1. The current control stack is abandoned in favor of the reified control stack resulting from the evaluation of e_2; the value of e_1 is then passed to that stack.

Here is a simple example, written in Standard ML notation. The idea is to multiply the elements of a list, short-circuiting the computation in case 0 is encountered. Here's the code:

```
fun mult_list (l:int list):int =
    letcc ret in
      let fun mult nil = 1
            | mult (0::_) = throw 0 to ret
            | mult (n::l) = n * mult l
      in  mult l end )
```

[1]Close relatives of these primitives are available in SML/NJ in the following forms: for letcc x in e, write SMLofNJ.Cont.callcc (fn x => e), and for throw e_1 to e_2, write SMLofNJ.Cont.throw e_2 e_1.

[2]Hence the name "letcc".

Ignoring the `letcc` for the moment, the body of `mult_list` is a `let` expression that defines a recursive procedure `mult`, and applies it to the argument of `mult_list`. The job of `mult` is to return the product of the elements of the list times the value of the accumulator; by calling `mult` with `l` and `1`, we obtain the product of the elements of `l`. Ignoring the second line of `mult`, it should be clear why and how this code works.

Now let's consider the second line of `mult`, and the outer use of `letcc`. Intuitively, the purpose of the second line of `mult` is to short circuit the multiplication, returning `0` immediately in the case that a `0` occurs in the list. This is achieved by throwing the value `0` (the final answer) to the continuation bound to the variable `ret`. This variable is bound by `letcc` surrounding the body of `mult_list`. What continuation is it? It's the continuation that runs upon completion of the body of `mult_list`. This continuation would be executed in the case that no `0` is encountered and evaluation proceeds normally. In the unusual case of encountering a `0` in the list, we branch directly to the return point, passing the value `0`, effecting an early return from the procedure with result value `0`.

Here's another formulation of the same function:

```
fun mult_list l =
    let fun mult nil ret = 1
          | mult (0::_) ret = throw 0 to ret
          | mult (n::l) ret = n * mult l ret
     in letcc ret in (mult l) ret end
```

Here the inner loop is parameterized by the return continuation for early exit. The multiplication loop is obtained by calling `mult` with the current continuation at the exit point of `mult_list` so that throws to `ret` effect an early return from `mult_list`, as desired.

Exercise 20
Study this example carefully to be sure you understand why it works!

Let's look at another example: given a continuation k of type τ `cont` and a function f of type $\tau'\to\tau$, return a continuation k' of type τ' `cont` with the following behavior: throwing a value v' of type τ' to k' throws the value $f(v')$ to k. This is called *composition of a function with a continuation*. We wish to fill in the following template:

```
fun compose (f:τ'->τ,k:τ cont):τ' cont = ...
```

The function `compose` will have type

$$((\tau' \texttt{ -> } \tau) \texttt{ * } \tau \texttt{ cont}) \texttt{ -> } \tau' \texttt{ cont}$$

Exercise 21
This is a very difficult programming problem! But please take a few moments to try to solve it before reading on. The solution is very instructive, but is, for most people, rather hard to think up.

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` f`(...) to` k. This is the continuation that, when given a value v', applies f to it, and throws the result to k. We can seize this continuation using `letcc`, writing

$$\texttt{throw } f\texttt{(letcc } x\texttt{:}\tau' \texttt{ cont in ...) to } k$$

At the point of the ellipsis the variable x is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting evaluation above! We don't want to actually throw a value to this continuation (yet), instead we wish to abort it and return it as the result. Here's the final code:

```
fun compose (f, k) =
    letcc ret in
      throw (f (letcc r in throw ret r)) to k
```

The type of `return` is τ' `cont cont`, a continuation expecting a continuation expecting a value of type τ'!

We can do without first-class continuations by "rolling our own". The idea is that we can perform (by hand or automatically) a systematic program transformation in which a "copy" of the control stack is maintained as a function, called a continuation. Every function takes as an argument the control stack to which it is to pass its result by applying given stack (represented as a function) to the result value. Functions never return in the usual sense; they pass their result to the given continuation. Programs written in this form are said to be in *continuation-passing style*, or *CPS* for short.

Here's the code to multiply the elements of a list (without short-circuiting) in continuation-passing style:

```
    fun cps_mult nil k = k 1
      | cps_mult (n::l) k = cps_mult l (fn r => k (n * r))
    fun mult l = cps_mult l (fn r => r)
```

It's easy to implement the short-circuit form by passing an additional continuation, the one to invoke for short-circuiting the result:

```
fun cps_mult_list l k =
    let fun cps_mult nil k0 k = k 1
          | cps_mult (0::_) k0 k = k0 0
          | cps_mult (n::l) k0 k =
              cps_mult k0 l (fn p => k (n*p))
    in cps_mult l k k end
```

The continuation `k0` never changes; it is always the return continuation for `cps_mult_list`. The argument continuation to `cps_mult_list` is duplicated on the call to `cps_mult`.

Observe that the type of the first version of `cps_mult` becomes

$$\texttt{int}\,\texttt{list}\to(\texttt{int}\to\alpha)\to\alpha,$$

and that the type of the second version becomes

$$\texttt{int}\,\texttt{list}\to(\texttt{int}\to\alpha)\to(\texttt{int}\to\alpha)\to\alpha,$$

These transformations are representative of the general case.

12.2 Semantics of Continuations

The informal description of evaluation is quite complex, as you no doubt have observed. Here's an example where a formal semantics is *much* clearer, and can serve as a useful guide for understanding how all of this works. The semantics is suprisingly simple and intuitive.

First, the abstract syntax. We extend the language of MinML types with continuation types of the form $\tau\,\texttt{cont}$. We extend the language of MinML expressions with these additional forms:

$$e \ ::= \ \ldots \mid \texttt{letcc}\,x\,\texttt{in}\,e \mid \texttt{throw}\,e_1\,\texttt{to}\,e_2 \mid K$$

In the expression $\texttt{letcc}\,x\,\texttt{in}\,e$ the variable x is bound in e. As usual we rename bound variables implicitly as convenient. We include control stacks K as expressions for the sake for the sake of the dynamic semantics, much as we included locations as expressions when considering reference types. We define continuations thought of as expressions to be values:

$$\frac{K\ \mathsf{stack}}{K\ \mathsf{value}} \tag{12.1}$$

Stacks are as defined for the C machine, extended with these additional frames:

$$\frac{e_2\ \mathsf{expr}}{\texttt{throw}\,\Box\,\texttt{to}\,e_2\ \mathsf{frame}} \tag{12.2}$$

$$\frac{v_1\ \mathsf{value}}{\texttt{throw}\,v_1\,\texttt{to}\,\Box\ \mathsf{frame}} \tag{12.3}$$

Second, the static semantics. The typing rules governing the continuation primitives are these:

$$\frac{\Gamma[x{:}\tau\,\texttt{cont}] \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\,x\,\texttt{in}\,e : \tau} \tag{12.4}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\,\texttt{cont}}{\Gamma \vdash \texttt{throw}\,e_1\,\texttt{to}\,e_2 : \tau'} \tag{12.5}$$

The result type of a $\texttt{throw}$ expression is arbitrary because it does not return to the point of the call. The typing rule for continuation values is as follows:

$$\frac{\vdash K : \tau\,\texttt{stack}}{\Gamma \vdash K : \tau\,\texttt{cont}} \tag{12.6}$$

That is, a continuation value K has type $\tau\,\texttt{cont}$ exactly if it is a stack accepting values of type τ. This relation is as defined in our treatment of exceptions, extended to include the additional frames mentioned above.

Finally, the dynamic semantics. We use the C machine as a basis. We extend the language of expressions to include control stacks K as values. Like locations, these arise only during execution; there is no explicit notation for continuations in the language. The key transitions are as follows:

$$\frac{}{(K, \texttt{letcc}\,x\,\texttt{in}\,e) \mapsto (K, \{K/x\}e)} \tag{12.7}$$

$$\frac{}{(\texttt{throw}\,v\,\texttt{to}\,\Box \triangleright K, K') \mapsto (K', v)} \qquad (12.8)$$

In addition we specify the order of evaluation of arguments to `throw`:

$$\frac{}{(K, \texttt{throw}\,e_1\,\texttt{to}\,e_2) \mapsto (\texttt{throw}\,\Box\,\texttt{to}\,e_2 \triangleright K, e_1)} \qquad (12.9)$$

$$\frac{}{(\texttt{throw}\,\Box\,\texttt{to}\,e_2 \triangleright K, v_1) \mapsto (\texttt{throw}\,v_1\,\texttt{to}\,\Box \triangleright K, e_2)} \qquad (12.10)$$

Notice that evaluation of `letcc` *duplicates* the control stack, and that evaluation of `throw` *eliminates* the current control stack.

Exercise 22
Simulate the evaluation of `compose (`f`, ` k`)` *on the empty stack. Observe that the control stack substituted for x is*

$$\texttt{apply(}f, \Box\texttt{)} \triangleright \texttt{throw}\,\Box\,\texttt{to}\,k \triangleright \bullet \qquad (12.11)$$

This stack is returned from `compose`*. Next, simulate the behavior of throwing a value v' to this continuation. Observe that the above stack is reinstated and that v' is passed to it.*

The safety of this extension of MinML may be established using familiar techniques. First we must define well-formedness for machine states. As before, we define (K, e) ok iff $\vdash K : \tau\,\texttt{stack}$ and $\vdash e : \tau$.

The preservation theorem is stated as follows:

Theorem 23 (Preservation)
If (K, e) ok and $(K, e) \mapsto (K', e')$, then (K', e') ok.

Proof: The proof is by induction on evaluation. The verification is left as an exercise. ■

To establish progress we need the following extension to the canonical forms lemma:

Lemma 24 (Canonical Forms)
If $\vdash v : \tau\,\texttt{cont}$, then $v = K$ for some control stack K such that $\vdash K : \tau\,\texttt{stack}$.

Finally, progress is stated as follows:

Theorem 25 (Progress)
If (K, e) ok *then either* $K = \bullet$ *and* e value*, or there exists* K' *and* e' *such that* $(K, e) \mapsto (K', e')$.

Proof: By induction on typing. The verification is left as an exercise. ■

12.3 Coroutines

Some problems are naturally implemented using *coroutines*, two (or more) routines that interleave their execution by an explicit hand-off of control from one to the other. In contrast to conventional sub-routines neither routine is "in charge", with one calling the other to execute to completion. Instead, the control relationship is symmetric, with each yielding control to the other during excecution.

A classic example of coroutining is provided by the producer-consumer model of interaction. The idea is that there is a common, hidden resource that is supplied by the producer and utilized by the consumer. Production of the resource is interleaved with its consumption by an explicit hand-off from producer to consumer. Here is an outline of a simple producer-consumer relationship, writting in Standard ML.

```
val buf : int ref = ref 0
fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))
fun consume (prod:state) =
    (print (!buf); consume (resume prod))
```

There the producer and consumer share an integer buffer. The producer fills it with successive integers; the consumer retrieves these values and prints them. The producer yields control to the consumer after filling the buffer; the consumer yields control to the producer after printing its contents. Since the handoff is explicit, the producer and consumer run in strict synchrony, alternating between production and consumption.

The key to completing this sketch is to detail the handoff protocol. The overall idea is to represent the state of a coroutine by a continuation, the point at which it should continue executing when it is resumed by another coroutine. The function `resume` captures the current continuation and throws it to the argument continuation, transferring control to the other coroutine and, simultaneously, informing it how to resume the caller. This means that the state of a coroutine is a continuation accepting the state of (another) coroutine, which leads to a recursive type. This leads to the following partial solution in terms of the SML/NJ continuation primitives:

```
datatype state = S of state cont
fun resume (S k : state) : state =
    callcc (fn k' : state cont => throw k (S k'))
val buf : int ref = ref 0
fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))
fun consume (prod:state) =
    (print (Int.toString(!buf)); consume (resume prod))
```

All that remains is to initialize the coroutines. It is natural to start by executing the producer, but arranging to pass it a coroutine state corresponding to the consumer. This can be achieved as follows:

```
fun run () =
    consume (callcc (fn k : state cont =>
                      produce (0, S k)))
```

Because of the call-by-value semantics of function application, we first seize the continuation corresponding to passing an argument to `consume`, then invoke `produce` with initial value `0` and this continuation. When `produce` yields control, it throws its state to the continuation that invokes `consume` with that state, at which point the coroutines have been initialized — further hand-off's work as described earlier.

This is, admittedly, a rather simple-minded example. However, it illustrates an important idea, namely the symmetric hand-off of control between routines. The difficulty with this style of programming is that the hand-off protocol is "hard wired" into the code. The producer yields control to the consumer, and *vice versa*, in strict alternating order. But what if there are multiple producers? Or multiple consumers? How would we handle priorities among them? What about asynchronous events such as arrival of a network packet or completion of a disk I/O request?

An elegant solution to these problems is to generalize the notion of a coroutine to the notion of a *user-level thread*. As with coroutines, threads enjoy a symmetric relationship among one another, but, unlike coroutines, they do not explicitly hand off control amongst themselves. Instead threads run as coroutines of a *scheduler* that mediates interaction among the threads, deciding which to run next based on considerations such as priority relationships or availability of data. Threads yield control to the scheduler, which determines which other thread should run next, rather than explicitly handing control to another thread.

Here is a simple interface for a user-level threads package:

```
signature THREADS = sig
  exception NoMoreThreads
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> 'a
end
```

The function `fork` is called to create a new thread executing the body of the given function. The function `yield` is called to cede control to another thread, selected by the thread scheduler. The function `exit` is called to terminate a thread.

User-level threads are naturally implemented as continuations. A thread is a value of type `unit cont`. The scheduler maintains a queue of threads that are ready to execute. To dispatch the scheduler dequeues a thread from the ready queue and invokes it by throwing `()` to it. Forking is implemented by creating a new thread. Yielding is achieved by enqueueing the current thread and dispatching; exiting is a simple dispatch, abandoning the current thread entirely. This implementation is suggestive of a slogan suggested by Olin Shivers: "A thread is a trajectory through continuation space". During its lifetime a thread of control is represented by a succession of continuations that are enqueued onto and dequeued from the ready queue.

Here is a simple implementation of threads:

```
structure Threads :> THREADS = struct
    open SMLofNJ.Cont
    exception NoRunnableThreads
    type thread = unit cont
    val readyQueue : thread Queue.queue = Queue.mkQueue()
    fun dispatch () =
        let
            val t = Queue.dequeue readyQueue
                    handle Queue.Dequeue
                           => raise NoRunnableThreads
        in
            throw t ()
        end
    fun exit () = dispatch()
    fun enqueue t = Queue.enqueue (readyQueue, t)
    fun fork f =
        callcc (fn parent => (enqueue parent; f ();
                              exit()))
    fun yield () =
        callcc (fn parent => (enqueue parent;
                              dispatch()))
end
```

Using the above thread interface we may implement the simple producer-consumer example as follows:

```
structure Client = struct
    open Threads
    val buffer : int ref = ref (˜1)
    fun producer (n) =
        (buffer := n ; yield () ; producer (n+1))
    fun consumer () =
        (print (Int.toString (!buffer)); yield ();
         consumer())
    fun run () =
        (fork (consumer); producer 0)
end
```

This example is excessively naïve, however, in that it relies on the strict FIFO ordering of threads by the scheduler, allowing careful control over the order of execution. If, for example, the producer were to run several times in a row before the consumer could run, several numbers would be omitted from the output.

Here is a better solution that avoids this problem (but does so by "busy waiting"):

```
structure Client = struct
    open Threads
    val buffer : int option ref = ref NONE
    fun producer (n) =
        (case !buffer
           of NONE => (buffer := SOME n ; yield() ;
                        producer (n+1))
            | SOME _ => (yield (); producer (n)))
    fun consumer () =
        (case !buffer
           of NONE => (yield (); consumer())
            | SOME n =>
              (print (Int.toString n);
               buffer := NONE; yield(); consumer()))
    fun run () =
        (fork (consumer); producer 0)
end
```

There is much more to be said about threads! We will return to this later in the course. For now, the main idea is to give a flavor of how first-class continuations can be used to implement a user-level threads package with very little difficulty. A more complete implementation is, of course, somewhat more complex, but not much more. We can easily provide all that is necessary for sophisticated thread programming in a few hundred lines of ML code.

Chapter 13

Exceptions

Exceptions effects a non-local transfer of control from the point at which the exception is *raised* to a dynamically enclosing *handler* for that exception. This transfer interrupts the normal flow of control in a program in response to unusual conditions. For example, exceptions can be used to signal an error condition, or to indicate the need for special handling in certain circumstances that arise only rarely. To be sure, one could use explicit conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly since the transfer to the handler is direct and immediate, rather than indirect via a series of explicit checks. All too often explicit checks are omitted (by design or neglect), whereas exceptions cannot be ignored.

We'll consider the extension of MinML with an exception mechanism similar to that of Standard ML, with the significant simplification that no value is associated with the exception — we simply signal the exception and thereby invoke the nearest dynamically enclosing handler. We'll come back to consider value-passing exceptions later.

The following grammar describes the extensions to MinML to support valueless exceptions:

$$e \ ::= \ \ldots \mid \texttt{fail} \mid \texttt{try}\, e_1\, \texttt{ow}\, e_2$$

The expression `fail` raises an exception. The expression $\texttt{try}\, e_1\, \texttt{ow}\, e_2$ evaluates e_1. If it terminates normally, we return its value; otherwise, if it fails, we continue by evaluating e_2.

The static semantics of exceptions is quite straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \qquad (13.1)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{ow}\, e_2 : \tau} \tag{13.2}$$

Observe that a failure can have any type, precisely because it never returns. Both clauses of a handler must have the same type, to allow for either possible outcome of evaluation.

The dynamic semantics of exceptions is given in terms of the C machine with an explicit control stack. The set of frames is extended with the following additional clause:

$$\frac{e_2\ \mathsf{expr}}{\texttt{try}\,\Box\,\texttt{ow}\,e_2\ \mathsf{frame}} \tag{13.3}$$

The evaluation rules are extended as follows:

$$\frac{}{(K, \texttt{try}\, e_1\, \texttt{ow}\, e_2) \mapsto (\texttt{try}\,\Box\,\texttt{ow}\,e_2 \triangleright K, e_1)} \tag{13.4}$$

$$\frac{}{(\texttt{try}\,\Box\,\texttt{ow}\,e_2 \triangleright K, v) \mapsto (K, v)} \tag{13.5}$$

$$\frac{}{(\texttt{try}\,\Box\,\texttt{ow}\,e_2 \triangleright K, \texttt{fail}) \mapsto (K, e_2)} \tag{13.6}$$

$$\frac{(F \neq \texttt{try}\,\Box\,\texttt{ow}\,e_2)}{(F \triangleright K, \texttt{fail}) \mapsto (K, \texttt{fail})} \tag{13.7}$$

To evaluate $\texttt{try}\, e_1\, \texttt{ow}\, e_2$ we begin by evaluating e_1. If it achieves a value, we "pop" the pending handler and yield that value. If, however, it fails, we continue by evaluating the "otherwise" clause of the nearest enclosing handler. Notice that we explicitly "pop" non-handler frames while processing a failure; this is sometimes called *unwinding* the control stack. Finally, we regard the state $(\bullet, \texttt{fail})$ as a final state of computation, corresponding to an uncaught exception.

Exercise 26
Hand-simulate the evaluation of a few simple expressions with exceptions and handlers to get a feeling for how it works.

To prove safety we define well-formedness of machine states by the following rule:

$$\frac{\vdash K : \tau\,\texttt{stack} \qquad \vdash e : \tau}{(K, e)\ \mathsf{ok}} \tag{13.8}$$

That is, a state (K, e) is well-formed iff e is an expression of type τ and K is a τ-accepting control stack. The latter is defined by the following rules:

$$\frac{}{\vdash \bullet : \tau\,\texttt{stack}} \tag{13.9}$$

$$\frac{\vdash F : \texttt{(}\tau\texttt{,}\tau'\texttt{)}\,\texttt{frame} \quad \vdash K : \tau'\,\texttt{stack}}{\vdash F \triangleright K : \tau\,\texttt{stack}} \tag{13.10}$$

A stack is well-formed iff its frames compose properly. The type of a frame is defined by the following rules:

$$\frac{\vdash e_2 : \texttt{int}}{\vdash +(\Box, e_2) : \texttt{(int,int)}\,\texttt{frame}} \tag{13.11}$$

$$\frac{v_1\ \mathsf{value} \quad \vdash v_1 : \texttt{int}}{\vdash +(v_1, \Box) : \texttt{(int,int)}\,\texttt{frame}} \tag{13.12}$$

$$\frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} : \texttt{(bool,}\tau\texttt{)}\,\texttt{frame}} \tag{13.13}$$

$$\frac{\vdash e_2 : \tau_2}{\vdash \texttt{apply(}\Box, e_2\texttt{)} : \texttt{(}\tau_2{\to}\tau\texttt{,}\tau\texttt{)}\,\texttt{frame}} \tag{13.14}$$

$$\frac{v_1\ \mathsf{value} \quad \vdash v_1 : \tau_2{\to}\tau}{\vdash \texttt{apply(}v_1, \Box\texttt{)} : \texttt{(}\tau_2\texttt{,}\tau\texttt{)}\,\texttt{frame}} \tag{13.15}$$

$$\frac{\vdash e_2 : \tau}{\vdash \texttt{try}\,\Box\,\texttt{ow}\,e_2 : \texttt{(}\tau\texttt{,}\tau'\texttt{)}\,\texttt{frame}} \tag{13.16}$$

Intuitively, a frame of type $\texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}\ \texttt{frame}$ takes an "argument" of type τ_1 and yields a "result" of type τ_2. The argument is represented by the "$\Box$" in the frame; the result is the type of the frame once its hole has been filled with an expression of the given type.

With this in place we can state and prove safety.

Theorem 27 (Preservation)
If (K, e) ok *and* $(K, e) \mapsto (K', e')$*, then* (K, e) ok.

Proof: By induction on evaluation. ■

Exercise 28
Prove Theorem 27.

Theorem 29 (Progress)
If (K, e) ok then either

1. $K = \bullet$ *and* e value*, or*

2. $K = \bullet$ *and* $e = \texttt{fail}$*, or*

3. *there exists* K' *and* e' *such that* $(K, e) \mapsto (K', e')$.

Proof: By induction on typing. ∎

Exercise 30
Prove Theorem 29.

Exercise 31
Combine the treatment of references and exceptions to form a language with both of these features. You will face a choice of how to define the interaction between mutation and exceptions:

1. *As in ML, mutations are irrevocable, even in the face of exceptions that "backtrack" to a surrounding handler.*

2. *Invocation of a handler rolls back the memory to the state at the point of installation of the handler.*

Give a dynamic semantics for each alternative, and argue for and against each choice.

The dynamic semantics of exceptions is somewhat unsatisfactory because of the explicit unwinding of the control stack to find the nearest enclosing handler. While this does effect a non-local transfer of control, it does so by rather crude means, rather than by a direct "jump" to the handler. In practice exceptions are implemented as jumps, using the following ideas. A dedicated register is set aside to contain the "current" exception handler. When an exception is raised, the current handler is retrieved from the exception register, and control is passed to it. Before doing so, however, we must reset the exception register to contain the nearest handler enclosing the new handler. This ensures that if the handler raises an exception the correct handler is invoked. How do we recover this handler? We maintain a stack of pending handlers that is pushed whenever a handler is installed, and popped whenever a handler is invoked. The exception register is the top element of this stack. Note that we must restore the control stack to the point at which the handler was installed before invoking the handler!

This can be modelled by a machine with states of the form (H, K, e), where

- H is a handler stack;
- K is a control stack;
- e is a closed expression

A handler stack consists of a stack of pairs consisting of a handler together its associated control stack:

$$\frac{}{\bullet \ \mathsf{hstack}} \tag{13.17}$$

$$\frac{K \ \mathsf{stack} \quad e \ \mathsf{expr} \quad H \ \mathsf{hstack}}{(K, e) \triangleright H \ \mathsf{hstack}} \tag{13.18}$$

A handler stack element consists of a "freeze dried" control stack paired with a pending handler.

The key transitions of the machine are given by the following rules. On failure we pop the control stack and pass to the exception stack:

$$\frac{}{((K', e') \triangleright H, K, \mathtt{fail}) \mapsto (H, K', e')} \tag{13.19}$$

We pop the handler stack, "thaw" the saved control stack, and invoke the saved handler expression. If there is no pending handler, we stop the machine:

$$\frac{}{(\bullet, K, \mathtt{fail}) \mapsto (\bullet, \bullet, \mathtt{fail})} \tag{13.20}$$

To install a handler we preserve the handler code and the current control stack:

$$\frac{}{(H, K, \mathtt{try}\, e_1 \, \mathtt{ow}\, e_2) \mapsto ((K, e_2) \triangleright H, \mathtt{try}\, \square \, \mathtt{ow}\, e_2 \triangleright K, e_1)} \tag{13.21}$$

We "freeze dry" the control stack, associate it with the unevaluated handler, and push it on the handler stack. We also push a frame on the control stack to remind us to remove the pending handler from the handler stack in the case of normal completion of evaluation of e_1:

$$\frac{}{((K, e_2) \triangleright H, \mathtt{try}\, \square \, \mathtt{ow}\, e_2 \triangleright K, v_1) \mapsto (H, K, v_1)} \tag{13.22}$$

Exercise 32
State and prove the safety of this formulation of exceptions.

The idea of "freeze-drying" an entire control stack and "thawing" it later may seem like an unusually heavy-weight operation. However, a key invariant governing a machine state (H, K, e) is the following *prefix property*: if $H = (K', e') \triangleright H'$, then K' is a prefix of K. This means that we can store a control stack by simply keeping a "finger" on some initial segment of it, and can restore a saved control stack by popping up to that finger.

Exercise 33
Prove that the prefix property is preserved by every step of evaluation.

Finally, let us consider value-passing exceptions such as are found in Standard ML. The main idea is to replace the failure expression, `fail`, by a more general *raise* expression, `raise(`e`)`, which associates a value (that of e) with the failure. Handlers are generalized so that the "otherwise" clause is a function accepting the value associated with the failure, and yielding a value of the same type as the "try" clause. Here is a sketch of the static semantics for this variation:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \tag{13.23}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_{\mathsf{exn}} \to \tau}{\Gamma \vdash \mathtt{try}\, e_1\, \mathtt{ow}\, e_2 : \tau} \tag{13.24}$$

These rules are parameterized by the type of values associated with exceptions, τ_{exn}.

The question is: what should be the type τ_{exn}? The first thing to observe is that *all* exceptions should be of the same type, otherwise we cannot guarantee type safety. The reason is that a handler might be invoked by *any* raise expression occurring during the execution of its "try" clause. If one exception raised an integer, and another a boolean, the handler could not safely dispatch on the exception value. Given this, we must choose a type τ_{exn} that supports a flexible programming style.

For example, we might choose, say, `string`, for τ_{exn}, with the idea that the value associated with an exception is a description of the cause of the exception. For example, we might write

```
fun div (m, 0) = raise "Division by zero attempted."
  | div (m, n) = ... raise "Arithmetic overflow occurred." ...
```

However, consider the plight of the poor handler, which may wish to distinguish between division-by-zero and arithmetic overflow. How might it do that? If exception values were strings, it would have to parse the string, relying on the message to be in a standard format, and dispatch based on the parse. This is manifestly unworkable. For similar reasons we wouldn't choose τ_{exn} to be, say, `int`, since that would require coding up exceptions as numbers, much like "error numbers" in Unix. Again, completely unworkable in practice, and completely unmodular (different modules are bound to conflict over their numbering scheme).

A more reasonable choice would be to define τ_{exn} to be a given datatype `exc`. For example, we might have the declaration

```
datatype exc = Div | Overflow | Match | Bind
```

as part of the implicit prelude of every program. Then we'd write

```
fun div (m, 0) = raise Div
  | div (m, n) = ... raise Overflow ...
```

Now the handler can easily dispatch on `Div` or `Overflow` using pattern matching, which is much better. However, this choice restricts all programs to a *fixed* set of exceptions, the value constructors associated with the predeclared `exc` datatype.

To allow extensibility Standard ML includes a special *extensible datatype* called `exn`. Values of type `exn` are similar to values of a datatype, namely they are constructed from other values using a constructor. Moreover, we may pattern match against values of type `exn` in the usual way. But, in addition, we may introduce *new* constructors of type `exn` "on the fly", rather than declare a fixed set at the beginning of the program. Such new constructors are introduced using an exception declaration such as the following:

```
exception Div
exception Overflow
```

Now `Div` and `Overflow` are constructors of type `exn`, and may be used in a `raise` expression or matched against by an exception handler. Exception declarations can occur anywhere in the program, and are guaranteed (by α-conversion) to be distinct from all other exceptions that may occur elsewhere in the program, even if they happen to have the same name. If two modules declare an exception named `Error`, then these are *different* exceptions; no confusion is possible.

The interesting thing about the `exn` type is that *it has nothing whatsoever to do with the exception mechanism* (beyond the fact that it is the type of values associated with exceptions). In particular, the `exception` declaration introduces a value constructor that has no inherent connection with the exception mechanism. We may use the `exn` type for other purposes; indeed, Java has an analogue of the type `exn`, called `Object`. This is the basis for downcasting and so-called typecase in Java.