Chapter 29

Control Stacks

The technique of structural dynamics is very useful for theoretical purposes, such as proving type safety, but is too high level to be directly usable in an implementation. One reason is that the use of "search rules" requires the traversal and reconstruction of an expression in order to simplify one small part of it. In an implementation we would prefer to use some mechanism to record "where we are" in the expression so that we may "resume" from that point after a simplification. This can be achieved by introducing an explicit mechanism, called a *control stack*, that keeps track of the context of an instruction step for just this purpose. By making the control stack explicit the transition rules avoid the need for any premises—every rule is an axiom. This is the formal expression of the informal idea that no traversals or reconstructions are required to implement it. In this chapter we introduce an abstract machine, $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, for the language $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. The purpose of this machine is to make control flow explicit by introducing a control stack that maintains a record of the pending sub-computations of a computation. We then prove the equivalence of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ with the structural dynamics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$.

29.1 Machine Definition

A state, s, of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ consists of a *control stack*, k, and a closed expression, e. States may take one of two forms:

1. An *evaluation* state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression, e, relative to a control stack, k.

2. A *return* state of the form $k \triangleleft e$, where e val, corresponds to the evaluation of a stack, k, relative to a closed value, e.

As an aid to memory, note that the separator "points to" the focal entity of the state, the expression in an evaluation state and the stack in a return state.

The control stack represents the context of evaluation. It records the "current location" of evaluation, the context into which the value of the current expression is to be returned. Formally, a control stack is a list of *frames*:

$$\frac{}{\epsilon \text{ stack}} \tag{29.1a}$$

$$\frac{f \text{ frame} \quad k \text{ stack}}{k; f \text{ stack}} \tag{29.1b}$$

The definition of frame depends on the language we are evaluating. The frames of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-) \text{ frame}} \tag{29.2a}$$

$$\frac{}{\mathtt{ifz}(-; e_1; x.e_2) \text{ frame}} \tag{29.2b}$$

$$\frac{}{\mathtt{ap}(-; e_2) \text{ frame}} \tag{29.2c}$$

The frames correspond to searchrules in the dynamics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. Thus, instead of relying on the structure of the transition derivation to maintain a record of pending computations, we make an explicit record of them in the form of a frame on the control stack.

The transition judgement between states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ machine is inductively defined by a set of inference rules. We begin with the rules for natural numbers.

$$\frac{}{k \triangleright \mathtt{z} \mapsto k \triangleleft \mathtt{z}} \tag{29.3a}$$

$$\frac{}{k \triangleright \mathtt{s}(e) \mapsto k; \mathtt{s}(-) \triangleright e} \tag{29.3b}$$

$$\frac{}{k; \mathtt{s}(-) \triangleleft e \mapsto k \triangleleft \mathtt{s}(e)} \tag{29.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s}(e)$, we push a frame on the stack to record the pending successor, and evaluate e; when that returns with e', we return $\mathtt{s}(e')$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \triangleright \mathtt{ifz}(e; e_1; x.e_2) \mapsto k; \mathtt{ifz}(-; e_1; x.e_2) \triangleright e} \tag{29.4a}$$

$$\frac{}{k; \mathtt{ifz}(-; e_1; x.e_2) \triangleleft \mathtt{z} \mapsto k \triangleright e_1} \tag{29.4b}$$

$$k;\mathtt{ifz}(-;e_1;x.e_2) \lhd \mathtt{s}(e) \mapsto k \rhd [e/x]e_2 \tag{29.4c}$$

First, the test expression is evaluated, recording the pending case analysis on the stack. Once the value of the test expression has been determined, we branch to the appropriate arm of the conditional, substituting the predecessor in the case of a positive number.

Finally, we consider the rules for functions and recursion.

$$k \rhd \mathtt{lam}[\tau](x.e) \mapsto k \lhd \mathtt{lam}[\tau](x.e) \tag{29.5a}$$

$$k \rhd \mathtt{ap}(e_1;e_2) \mapsto k;\mathtt{ap}(-;e_2) \rhd e_1 \tag{29.5b}$$

$$k;\mathtt{ap}(-;e_2) \lhd \mathtt{lam}[\tau](x.e) \mapsto k \rhd [e_2/x]e \tag{29.5c}$$

$$k \rhd \mathtt{fix}[\tau](x.e) \mapsto k \rhd [\mathtt{fix}[\tau](x.e)/x]e \tag{29.5d}$$

These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated. Note that evaluation of general recursion requires no stack space! (But see Chapter 39 for more on evaluation of general recursion.)

The initial and final states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are defined by the following rules:

$$\epsilon \rhd e \text{ initial} \tag{29.6a}$$

$$\frac{e \text{ val}}{\epsilon \lhd e \text{ final}} \tag{29.6b}$$

29.2 Safety

To define and prove safety for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ requires that we introduce a new typing judgement, $k : \tau$, stating that the stack k expects a value of type τ. This judgement is inductively defined by the following rules:

$$\epsilon : \tau \tag{29.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{k;f : \tau} \tag{29.7b}$$

This definition makes use of an auxiliary judgement, $f : \tau \Rightarrow \tau'$, stating that a frame f transforms a value of type τ to a value of type τ'.

$$\mathtt{s}(-) : \mathtt{nat} \Rightarrow \mathtt{nat} \tag{29.8a}$$

$$\frac{e_1 : \tau \quad x : \mathtt{nat} \vdash e_2 : \tau}{\mathtt{ifz}(-;e_1;x.e_2) : \mathtt{nat} \Rightarrow \tau} \tag{29.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-;e_2) : \mathtt{arr}(\tau_2;\tau) \Rightarrow \tau} \tag{29.8c}$$

The two forms of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ state are well-formed provided that their stack and expression components match.

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \ \mathsf{ok}} \tag{29.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \triangleleft e \ \mathsf{ok}} \tag{29.9b}$$

We leave the proof of safety of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ as an exercise.

Theorem 29.1 (Safety). *1. If s ok and $s \mapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \mapsto s'$.

29.3 Correctness of the Control Machine

It is natural to ask whether $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ correctly implements $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. If we evaluate a given expression, e, using $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, do we get the same result as would be given by $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, and *vice versa*?

Answering this question decomposes into two conditions relating $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ to $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$:

Completeness If $e \mapsto^* e'$, where e' val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$.

Soundness If $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$, then $e \mapsto^* e'$ with e' val.

Let us consider, in turn, what is involved in the proof of each part.

For completeness it is natural to consider a proof by induction on the definition of multistep transition, which reduces the theorem to the following two lemmas:

1. If e val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e$.
2. If $e \mapsto e'$, then, for every v val, if $\epsilon \triangleright e' \mapsto^* \epsilon \triangleleft v$, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$.

The first can be proved easily by induction on the structure of e. The second requires an inductive analysis of the derivation of $e \mapsto e'$, giving rise to two complications that must be accounted for in the proof. The first complication is that we cannot restrict attention to the empty stack, for if e is, say, $\mathtt{ap}(e_1;e_2)$, then the first step of the machine is

$$\epsilon \triangleright \mathtt{ap}(e_1;e_2) \mapsto \epsilon;\mathtt{ap}(-;e_2) \triangleright e_1,$$

and so we must consider evaluation of e_1 on a non-empty stack.

A natural generalization is to prove that if $e \mapsto e'$ and $k \triangleright e' \mapsto^* k \triangleleft v$, then $k \triangleright e \mapsto^* k \triangleleft v$. Consider again the case $e = \mathtt{ap}(e_1; e_2)$, $e' = \mathtt{ap}(e_1'; e_2)$, with $e_1 \mapsto e_1'$. We are given that $k \triangleright \mathtt{ap}(e_1'; e_2) \mapsto^* k \triangleleft v$, and we are to show that $k \triangleright \mathtt{ap}(e_1; e_2) \mapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

$$k \triangleright \mathtt{ap}(e_1'; e_2) \mapsto k; \mathtt{ap}(-; e_2) \triangleright e_1'.$$

We would like to apply induction to the derivation of $e_1 \mapsto e_1'$, but to do so we must have a v_1 such that $e_1' \mapsto^* v_1$, which is not immediately at hand.

This means that we must consider the ultimate value of each sub-expression of an expression in order to complete the proof. This information is provided by the evaluation dynamics described in Chapter 9, which has the property that $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

Lemma 29.2. *If $e \Downarrow v$, then for every k stack, $k \triangleright e \mapsto^* k \triangleleft v$.*

The desired result follows by the analogue of Theorem 9.2 on page 83 for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, which states that $e \Downarrow v$ iff $e \mapsto^* v$.

For the proof of soundness, it is awkward to reason inductively about the multistep transition from $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$ machine state as encoding an expression, and show that $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$ transitions are simulated by $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ transitions under this encoding.

Specifically, we define a judgement, $s \looparrowright e$, stating that state s "unravels to" expression e. It will turn out that for initial states, $s = \epsilon \triangleright e$, and final states, $s = \epsilon \triangleleft e$, we have $s \looparrowright e$. Then we show that if $s \mapsto^* s'$, where s' final, $s \looparrowright e$, and $s' \looparrowright e'$, then e' val and $e \mapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \looparrowright e$ and s final, then e val.
2. If $s \mapsto s'$, $s \looparrowright e$, $s' \looparrowright e'$, and $e' \mapsto^* v$, where v val, then $e \mapsto^* v$.

The first is quite simple, we need only observe that the unravelling of a final state is a value. For the second, it is enough to show the following lemma.

Lemma 29.3. *If $s \mapsto s'$, $s \looparrowright e$, and $s' \looparrowright e'$, then $e \mapsto^* e'$.*

Corollary 29.4. *$e \mapsto^* \overline{n}$ iff $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft \overline{n}$.*

The remainder of this section is devoted to the proofs of the soundness and completeness lemmas.

29.3.1 Completeness

Proof of Lemma 29.2 on the previous page. The proof is by induction on an evaluation dynamics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau_2](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1; e_2) \Downarrow v} \tag{29.10}$$

For an arbitrary control stack, k, we are to show that $k \triangleright \mathtt{ap}(e_1; e_2) \mapsto^* k \triangleleft v$. Applying both of the inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1; e_2) &\mapsto k; \mathtt{ap}(-; e_2) \triangleright e_1 \\ &\mapsto^* k; \mathtt{ap}(-; e_2) \triangleleft \mathtt{lam}[\tau_2](x.e) \\ &\mapsto k \triangleright [e_2/x]e \\ &\mapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

29.3.2 Soundness

The judgement $s \not\mapsto e'$, where s is either $k \triangleright e$ or $k \triangleleft e$, is defined in terms of the auxiliary judgement $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \not\mapsto e'} \tag{29.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \not\mapsto e'} \tag{29.11b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{29.12a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{k; \mathtt{s}(-) \bowtie e = e'} \tag{29.12b}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1; e_2; x.e_3) = e'}{k; \mathtt{ifz}(-; e_2; x.e_3) \bowtie e_1 = e'} \tag{29.12c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1; e_2) = e}{k; \mathtt{ap}(-; e_2) \bowtie e_1 = e} \tag{29.12d}$$

These judgements both define total functions.

Lemma 29.5. *The judgement $s \not\mapsto e$ has mode $(\forall, \exists!)$, and the judgement $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$.*

That is, each state unravels to a unique expression, and the result of wrapping a stack around an expression is uniquely determined. We are therefore justified in writing $k \bowtie e$ for the unique e' such that $k \bowtie e = e'$.

The following lemma is crucial. It states that unravelling preserves the transition relation.

Lemma 29.6. *If $e \mapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \mapsto d'$.*

Proof. The proof is by rule induction on the transition $e \mapsto e'$. The inductive cases, in which the transition rule has a premise, follow easily by induction. The base cases, in which the transition is an axiom, are proved by an inductive analysis of the stack, k.

For an example of an inductive case, suppose that $e = \mathtt{ap}(e_1; e_2)$, $e' = \mathtt{ap}(e_1'; e_2)$, and $e_1 \mapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from Rules (29.12) that $k; \mathtt{ap}(-; e_2) \bowtie e_1 = d$ and $k; \mathtt{ap}(-; e_2) \bowtie e_1' = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\mathtt{lam}[\tau_2](x.e); e_2)$ and $e' = [e_2/x]e$ with $e \mapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \mapsto d'$. We proceed by an inner induction on the structure of k. If $k = \epsilon$, the result follows immediately. Consider, say, the stack $k = k'; \mathtt{ap}(-; c_2)$. It follows from Rules (29.12) that $k' \bowtie \mathtt{ap}(e; c_2) = d$ and $k' \bowtie \mathtt{ap}(e'; c_2) = d'$. But by the SOS rules $\mathtt{ap}(e; c_2) \mapsto \mathtt{ap}(e'; c_2)$, so by the inner inductive hypothesis we have $d \mapsto d'$, as desired. □

We are now in a position to complete the proof of Lemma 29.3 on page 277.

Proof of Lemma 29.3 on page 277. The proof is by case analysis on the transitions of $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$. In each case after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Suppose that $s = k \triangleright \mathtt{s}(e)$ and $s' = k; \mathtt{s}(-) \triangleright e$. Note that $k \bowtie \mathtt{s}(e) = e'$ iff $k; \mathtt{s}(-) \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = k; \mathtt{ap}(\mathtt{lam}[\tau](x.e_1); -) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k; \mathtt{ap}(\mathtt{lam}[\tau](x.e_1); -) \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathtt{ap}(\mathtt{lam}[\tau](x.e_1); e_2) = e'$. The result follows from Lemma 29.6.

□

29.4 Notes

The abstract machine considered here is typical of a wide class of machines that make control flow explicit in the state. The prototype is Landin's SECD machine [49], which may be seen as a linearization of a structural operational semantics [75]. An advantage of a machine model is that the explicit treatment of control is natural for languages that allow the control state to be explicitly manipulated (see Chapter 31 for a prime example). A disadvantage is that one is required to make explicit the control state of the computation, rather than leave it implicit as in structural operational semantics. Which is better depends wholly on the situation at hand, though historically there has been greater emphasis on abstract machines than on structural semantics.

Chapter 30

Exceptions

Exceptions effect a non-local transfer of control from the point at which the exception is *raised* to an 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.

30.1 Failures

A *failure* is a control mechanism that permits a computation to refuse to return a value to the point of its evaluation. Failure can be detected by *catching* it, diverting evaluation to another expression, called a *handler*. Failure can be turned into success, provided that the handler does not itself fail.

The following grammar defines the syntax of failures:

Expr	e	::=	`fail`	`fail` failure
			`catch`$(e_1;e_2)$	`catch` e_1 `ow` e_2 handler

The expression `fail` aborts the current evaluation, and the expression `catch`$(e_1;e_2)$ handles any failure in e_1 by evaluating e_2 instead.

The statics of failures is straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{30.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1;e_2) : \tau} \tag{30.1b}$$

A failure can have any type, because it never returns. The two expressions in a `catch` expression must have the same type, since either might determine the value of that expression.

The dynamics of failures may be given using *stack unwinding*. Evaluation of a `catch` installs a handler on the control stack. Evaluation of a `fail` unwinds the control stack by popping frames until it reaches the nearest enclosing handler, to which control is passed. The handler is evaluated in the context of the surrounding control stack, so that failures within it propagate further up the stack.

Stack unwinding can be defined directly using structural dynamics, but we prefer to make use of the stack machine defined in Chapter 29. In addition to states of the form $k \triangleright e$, which evaluates the expression e on the stack k, and $k \triangleleft e$, which passes the value e to the stack k, we make use of an additional form of state, $k \blacktriangleleft$, which passes a failure up the stack to the nearest enclosing handler.

The set of frames defined in Chapter 29 is extended with the additonal form $\texttt{catch}(-;e_2)$. The transition rules given in Chapter 29 are extended with the following additional rules:

$$\overline{k \triangleright \texttt{fail} \mapsto k \blacktriangleleft} \tag{30.2a}$$

$$\overline{k \triangleright \texttt{catch}(e_1;e_2) \mapsto k;\texttt{catch}(-;e_2) \triangleright e_1} \tag{30.2b}$$

$$\overline{k;\texttt{catch}(-;e_2) \triangleleft v \mapsto k \triangleleft v} \tag{30.2c}$$

$$\overline{k;\texttt{catch}(-;e_2) \blacktriangleleft \mapsto k \triangleright e_2} \tag{30.2d}$$

$$\overline{k;f \blacktriangleleft \mapsto k \blacktriangleleft} \tag{30.2e}$$

As a notational convenience, we require that Rule (30.2e) apply only if none of the preceding rules apply. Evaluating `fail` propagates a failure up the stack. The act of raising an exception may itself raise an exception. Evaluating $\texttt{catch}(e_1;e_2)$ consists of pushing the handler onto the control stack and evaluating e_1. If a value is propagated to the handler, the handler is removed and the value continues to propagate upwards. If a failure is propagated to the handler, the stored expression is evaluated with the handler removed from the control stack. All other frames propagate failures.

The definition of initial state remains the same as for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\epsilon \lhd e\ \mathsf{final}} \tag{30.3a}$$

$$\epsilon \blacktriangleleft\ \mathsf{final} \tag{30.3b}$$

The first of these is as before, corresponding to a normal result with the specified value. The second is new, corresponding to an uncaught exception propagating through the entire program.

It is a straightforward exercise the extend the definition of stack typing given in Chapter 29 to account for the new forms of frame. Using this, safety can be proved by standard means. Note, however, that the meaning of the progress theorem is now significantly different: a well-typed program does not get stuck, but it may well result in an uncaught failure!

Theorem 30.1 (Safety). *1. If s ok and $s \mapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \mapsto s'$.

30.2 Exceptions

Failures are simplistic in that they do not distinguish different causes, and hence do not permit handlers to react differently to different circumstances. An *exception* is a generalization of a failure that associates a value with the failure. This value is passed to the handler, allowing it to discriminate between various forms of failures, and to pass data appropriate to that form of failure. The type of values associated with exceptions is discussed in Section 30.3 on the next page. For now, we simply assume that there is some type, τ_{exn}, of values associated with a failure.

The syntax of exceptions is given by the following grammar:

Expr	e	::=	$\mathtt{raise}[\tau](e)$	$\mathtt{raise}(e)$	exception
			$\mathtt{handle}(e_1; x.e_2)$	$\mathtt{handle}\, e_1\, \mathtt{ow}\, x \Rightarrow e_2$	handler

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\mathtt{handle}(e_1; x.e_2)$ binds a variable, x, in the handler, e_2, to which the associated value of the exception is bound, should an exception be raised during the execution of e_1.

The statics of exceptions generalizes that of failures:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \mathtt{raise}[\tau](e) : \tau} \tag{30.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \mathtt{handle}(e_1; x.e_2) : \tau} \tag{30.4b}$$

The dynamics of exceptions is a mild generalization of the dynamics of failures in which we generalize the failure state, $k \blacktriangleleft$, to the exception state, $k \blacktriangleleft e$, which passes a value of type τ_{exn} along with the failure. The syntax of stack frames is extended to include $\mathtt{raise}[\tau](-)$ and $\mathtt{handle}(-; x.e_2)$. The dynamics of exceptions is specified by the following rules:

$$\overline{k \triangleright \mathtt{raise}[\tau](e) \mapsto k; \mathtt{raise}[\tau](-) \triangleright e} \tag{30.5a}$$

$$\overline{k; \mathtt{raise}[\tau](-) \triangleleft e \mapsto k \blacktriangleleft e} \tag{30.5b}$$

$$\overline{k; \mathtt{raise}[\tau](-) \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{30.5c}$$

$$\overline{k \triangleright \mathtt{handle}(e_1; x.e_2) \mapsto k; \mathtt{handle}(-; x.e_2) \triangleright e_1} \tag{30.5d}$$

$$\overline{k; \mathtt{handle}(-; x.e_2) \triangleleft e \mapsto k \triangleleft e} \tag{30.5e}$$

$$\overline{k; \mathtt{handle}(-; x.e_2) \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{30.5f}$$

$$\frac{(f \neq \mathtt{handle}(-; x.e_2))}{k; f \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{30.5g}$$

It is a straightforward exercise to extend the safety theorem given in Section 30.1 on page 281 to exceptions.

30.3 Exception Type

The statics of exceptions is parameterized by the type of exception values, τ_{exn}. This type may be chosen arbitrarily, but it must be shared by all exceptions in a program to ensure type safety. For otherwise a handler cannot tell what type of value to expect from an exception, compromising safety.

But how do we choose the type of exceptions? A very naïve choice would be to take τ_{exn} to be the type `str`, so that, for example, one may write

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault. This is fine as far as it goes, but a handler for such an exception would have to interpret the string if it is to distinguish one exception from another!

Motivated by this, we might choose τ_{exn} to be `nat`, which amounts to saying that exceptional conditions are coded as natural numbers.[1] This does allow the handler to distinguish one source of failure from another, but makes no provision for associating data with the failure. Moreover, it forces the programmer to impose a single, global convention for indexing the causes of failure, compromising modular development and evolution.

The first concern—how to associate data specific to the type of failure—can be addressed by taking τ_{exn} to be a labelled sum type whose classes are the forms of failure, and whose associated types determine the form of the data attached to the exception. For example, the type τ_{exn} might have the form

$$\tau_{exn} = \texttt{[div:unit,fnf:string,...]}.$$

The class `div` might represent an arithmetic fault, with no associated data, and the class `fnf` might represent a "file not found" error, with associated data being the name of the file.

Using a sum type for τ_{exn} makes it easy for the handler to discriminate on the source of the failure, and to recover the associated data without fear of a type safety violation. For example, we might write

```
try e1 ow x ⇒
  match x {
    div ⟨⟩ ⇒ e_div
  | fnf s ⇒ e_fnf }
```

to handle the exceptions specified by the sum type given in the preceding paragraph.

The problem with choosing a sum type for τ_{exn} is that it imposes a *static classification* of the sources of failure in a program. There must be one, globally agreed-upon type that classifies all possible forms of failure, and specifies their associated data. Using sums in this manner impedes modular

[1] In Unix these are called `errno`'s, for *error numbers*.

development and evolution, since all of the modules comprising a system must agree on the one, central type of exception values. A better approach is to use *dynamic classification* for exception values by choosing τ_{exn} to be an *extensible sum*, one to which new classes may be added at execution time. This allows separate program modules to introduce their own failure classification scheme without worrying about interference with one another; the initialization of the module generates new classes at run-time that are guaranteed to be distinct from all other classes previously or subsequently generated. (See Chapter 36 for more on dynamic classification.)

30.4 Encapsulation

It is sometimes useful to distinguish expressions that can fail or raise an exception from those that cannot. An expression is called *fallible*, or *exceptional*, if it can fail or raise an exception during its evaluation, and is *infallible*, or *unexceptional*, otherwise. The concept of fallibility is intentionally permissive in that an infallible expression may be considered to be (vacuously) fallible, whereas infallibility is intended to be strict in that an infallible expression cannot fail. Consequently, if e_1 and e_2 are two infallible expressions both of whose values are required in a computation, we may evaluate them in either order without affecting the outcome. If, on the other hand, one or both are fallible, then the outcome of the computation is sensitive to the evaluation order (whichever fails first determines the overall result).

To formalize this distinction we distinguish two *modes* of expression, the fallible and the infallible, linked by a *modality* classifying the fallible expressions of a type.

Type	τ	$::=$ $\texttt{fallible}(\tau)$	$\tau\ \texttt{fallible}$	fallible
Fall	f	$::=$ $\texttt{fail}$	$\texttt{fail}$	failure
		$\texttt{succ}(e)$	$\texttt{succ}\ e$	success
		$\texttt{try}(e; x.f_1; f_2)$	$\texttt{let fall}(x)\ \texttt{be}\ e\ \texttt{in}\ f_1\ \texttt{ow}\ f_2$	handler
Infall	e	$::=$ x	x	variable
		$\texttt{fall}(f)$	$\texttt{fall}\ f$	fallible
		$\texttt{try}(e; x.e_1; e_2)$	$\texttt{let fall}(x)\ \texttt{be}\ e\ \texttt{in}\ e_1\ \texttt{ow}\ e_2$	handler

The type τ `fallible` is the type of encapsulated fallible expressions of type τ. Fallible expressions include failures, successes (infallible expressions thought of as vacuously fallible), and handlers that intercept failures,

but which may itself fail. Infallible expressions include variables, encapsulated fallible expressions, and handlers that intercepts failures, always yielding an infallible result.

The statics of encapsulated failures consists of two judgement forms, $\Gamma \vdash e : \tau$ for infallible expressions and $\Gamma \vdash f \sim \tau$ for fallible expressions. These judgements are defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{30.6a}$$

$$\frac{\Gamma \vdash f \sim \tau}{\Gamma \vdash \mathtt{fall}(f) : \mathtt{fallible}(\tau)} \tag{30.6b}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{try}(e; x.e_1; e_2) : \tau'} \tag{30.6c}$$

$$\frac{}{\Gamma \vdash \mathtt{fail} \sim \tau} \tag{30.6d}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{succ}(e) \sim \tau} \tag{30.6e}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash f_1 \sim \tau' \quad \Gamma \vdash f_2 \sim \tau'}{\Gamma \vdash \mathtt{try}(e; x.f_1; f_2) \sim \tau'} \tag{30.6f}$$

Rule (30.6c) specifies that a handler may be used to turn a fallible expression (encapsulated by e) into an infallible computation, provided that the result is infallible regardless of whether the encapsulated expression succeeds or fails.

The dynamics of encapsulated failures is readily derived, though some care must be taken with the elimination form for the modality.

$$\frac{}{\mathtt{fall}(f)\ \mathtt{val}} \tag{30.7a}$$

$$\frac{}{k \triangleright \mathtt{try}(e; x.e_1; e_2) \mapsto k; \mathtt{try}(-; x.e_1; e_2) \triangleright e} \tag{30.7b}$$

$$\frac{}{k; \mathtt{try}(-; x.e_1; e_2) \triangleleft \mathtt{fall}(f) \mapsto k; \mathtt{try}(-; x.e_1; e_2); \mathtt{fall}(-) \triangleright f} \tag{30.7c}$$

$$\frac{}{k \triangleright \mathtt{fail} \mapsto k \blacktriangleleft} \tag{30.7d}$$

$$k \triangleright \mathtt{succ}(e) \mapsto k;\mathtt{succ}(-) \triangleright e \tag{30.7e}$$

$$k;\mathtt{succ}(-) \triangleleft e \mapsto k \triangleleft \mathtt{succ}(e) \tag{30.7f}$$

$$k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-) \triangleleft \mathtt{succ}(e) \mapsto k \triangleright [e/x]e_1 \quad (e \text{ val}) \tag{30.7g}$$

$$k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-) \blacktriangleleft \mapsto k \triangleright e_2 \tag{30.7h}$$

We have omitted the rules for the fallible form of handler; they are similar to Rules (30.7b) to (30.7b) and (30.7g) to (30.7h), albeit with infallible subexpressions e_1 and e_2 replaced by fallible subexpressions f_1 and f_2.

An initial state has the form $k \triangleright e$, where e is an infallible expression, and k is a stack of suitable type. Consequently, a fallible expression, f, can only be evaluated on a stack of the form

$$k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-)$$

in which a handler for any failure that may arise from f is present. Therefore, a final state has the form $\epsilon \triangleleft e$, where e val; no uncaught failure can arise.

30.5 Notes

Various forms of exceptions were explored in the many dialects of Lisp (see, for example, [90]). The original formulation of ML as a metalanguage for mechanized logic [33] made extensive use of exceptions (called "failures") to implement tactics and tacticals. Most modern languages now include an exception mechanism of the kind considered here.

The essential distinction between the exception *mechanism* and exception *values* is often misunderstood. The two have no relationship to one another. Exception values are often dynamically classified (see Chapter 36), but dynamic classification has many more uses than just exception values. Another common misconception is to link exceptions with fluid binding (for which see Chapter 35). As the account given here makes clear, there is absolutely no relationship between exceptions and fluid binding. Exceptions are simply a particular use of the monad associated with option types, as described in Section 30.4 on page 286.

Chapter 31

Continuations

The semantics of many control constructs (such as exceptions and co-routines) can be expressed in terms of *reified* control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value 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 *continuations*; they are values that can be passed and returned at will in a computation. Continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus 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.

Why are continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using 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.

31.1 Informal Overview

We will extend $\mathcal{L}\{\rightarrow\}$ with the type `cont(`τ`)` of continuations accepting values of type τ. The introduction form for `cont(`τ`)` is `letcc[`τ`](`$x.e$`)`, which binds the *current continuation* (that is, the current control stack) to the variable x, and evaluates the expression e. The corresponding elimination

form is `throw[τ](e1;e2)`, which restores the value of e_1 to the control stack that is the value of e_2.

To illustrate the use of these primitives, consider the problem of multiplying the first n elements of an infinite sequence q of natural numbers, where q is represented by a function of type `nat → nat`. If zero occurs among the first n elements, we would like to effect an "early return" with the value zero, rather than perform the remaining multiplications. This problem can be solved using exceptions (we leave this as an exercise), but we will give a solution that uses continuations in preparation for what follows.

Here is the solution in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, without short-cutting:

```
fix ms is
  λ q : nat ⇀ nat.
    λ n : nat.
      case n {
        z ⇒ s(z)
      | s(n') ⇒ (q z) × (ms (q ∘ succ) n')
      }
```

The recursive call composes q with the successor function to shift the sequence by one step.

Here is the version with short-cutting:

```
λ q : nat ⇀ nat.
  λ n : nat.
    letcc ret : nat cont in
      let ms be
        fix ms is
          λ q : nat ⇀ nat.
            λ n : nat.
              case n {
                z ⇒ s(z)
              | s(n') ⇒
                case q z {
                  z ⇒ throw z to ret
                | s(n'') ⇒ (q z) × (ms (q ∘ succ) n')
                }
              }
      in
        ms q n
```

The `letcc` binds the return point of the function to the variable `ret` for use within the main loop of the computation. If zero is encountered, control is thrown to `ret`, effecting an early return with the value zero.

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 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

```
throw f(letcc x:τ' 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:τ' → τ, k:τ cont):τ' cont =
    letcc ret:τ' cont cont in
      throw (f (letcc r in throw r to ret)) to k
```

The type of `ret` is that of a continuation-expecting continuation!

31.2 Semantics of Continuations

We extend the language of $\mathcal{L}\{\to\}$ expressions with these additional forms:

Type	τ	::=	`cont(`τ`)`	τ `cont`	continuation
Expr	e	::=	`letcc[`τ`](`$x.e$`)`	`letcc` x `in` e	mark
			`throw[`τ`](`$e_1;e_2$`)`	`throw` e_1 `to` e_2	goto
			`cont(`k`)`	`cont(`k`)`	continuation

The expression `cont(`k`)` is a reified control stack, which arises during evaluation.

The statics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{31.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau'](e_1; e_2) : \tau'} \tag{31.1b}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call.

The statics of continuation values is given by the following rule:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{31.2}$$

A continuation value $\texttt{cont}(k)$ has type $\texttt{cont}(\tau)$ exactly if it is a stack accepting values of type τ.

To define the dynamics we extend $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{e_2 \text{ exp}}{\texttt{throw}[\tau](-; e_2) \text{ frame}} \tag{31.3a}$$

$$\frac{e_1 \text{ val}}{\texttt{throw}[\tau](e_1; -) \text{ frame}} \tag{31.3b}$$

Every reified control stack is a value:

$$\frac{k \text{ stack}}{\texttt{cont}(k) \text{ val}} \tag{31.4}$$

The transition rules for the continuation constructs are as follows:

$$k \triangleright \texttt{letcc}[\tau](x.e) \mapsto k \triangleright [\texttt{cont}(k)/x]e \tag{31.5a}$$

$$k \triangleright \texttt{throw}[\tau](e_1; e_2) \mapsto k; \texttt{throw}[\tau](-; e_2) \triangleright e_1 \tag{31.5b}$$

$$\frac{e_1 \text{ val}}{k; \texttt{throw}[\tau](-; e_2) \triangleleft e_1 \mapsto k; \texttt{throw}[\tau](e_1; -) \triangleright e_2} \tag{31.5c}$$

$$k; \texttt{throw}[\tau](v; -) \triangleleft \texttt{cont}(k') \mapsto k' \triangleleft v \tag{31.5d}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

The safety of this extension of $\mathcal{L}\{\rightarrow\}$ may be established by a simple extension to the safety proof for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ given in Chapter 29.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \texttt{cont}(\tau)}{\texttt{throw}[\tau']\,(-;e_2) : \tau \Rightarrow \tau'} \tag{31.6a}$$

$$\frac{e_1 : \tau \quad e_1\ \texttt{val}}{\texttt{throw}[\tau']\,(e_1;-) : \texttt{cont}(\tau) \Rightarrow \tau'} \tag{31.6b}$$

The rest of the definitions remain as in Chapter 29.

Lemma 31.1 (Canonical Forms). *If $e : \texttt{cont}(\tau)$ and e val, then $e = \texttt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 31.2 (Safety). *1. If s ok and $s \mapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \mapsto s'$.

31.3 Coroutines

The distinction between a routine and a subroutine is the distinction between a manager and a worker. The routine calls upon the subroutine to accomplish a piece of work, and the subroutine returns to the routine when its work is done. The relationship is asymmetric in that there is a clear distinction between the *caller*, the main routine, and the *callee*, the subroutine. Often it is useful to consider a symmetric situation in which two routines each call the other to help accomplish a task. Such a pair of routines are called *coroutines*; their relationship to one another is symbiotic rather than parasitic.

The key to implementing a subroutine is for the caller to pass to the callee a continuation representing the return point of the subroutine call. When the subroutine is finished, it calls the continuation passed to it by the calling routine. Since the subroutine is finished at that point, there is no need for the callee to pass a continuation back to the caller. The key to implementing coroutines is to have each routine treat the other as a subroutine of itself. In particular, whenever a coroutine cedes control to its caller, it provides a continuation that the caller may use to cede control back to the callee, in the process providing a continuation for itself. (This raises an interesting question of how the whole process gets started. We'll return to this shortly.)

To see how a pair of coroutines is implemented, let us consider the type of each routine in the pair. A routine is a continuation accepting two arguments, a datum to be passed to that routine when it is resumed, and a

continuation to be resumed when the routine is finished its task. The datum represents the state of the computation, and the continuation is a coroutine that accepts arguments of the same form. Thus, the type of a coroutine must satisfy the type isomorphism

$$\tau\,\texttt{coro} \cong (\tau \times \tau\,\texttt{coro})\,\texttt{cont}.$$

So we may take $\tau\,\texttt{coro}$ to be the recursive type

$$\tau\,\texttt{coro} \triangleq \mu t\,.\,(\tau \times t)\,\texttt{cont}.$$

Up to isomorphism, the type $\tau\,\texttt{coro}$ is the type of continuations that accept a value of type τ, representing the state of the coroutine, and the partner coroutine, a value of the same type.

A coroutine, r, passes control to another coroutine, r', by evaluating the expression $\texttt{resume}(\langle s, r'\rangle)$, where s is the current state of the computation. Doing so creates a new coroutine whose entry point is the return point (calling site) of the application of `resume`. Therefore the type of `resume` is

$$\tau \times \tau\,\texttt{coro} \to \tau \times \tau\,\texttt{coro}.$$

The definition of `resume` is as follows:

$$\lambda\,(\langle s, r'\rangle : \tau \times \tau\,\texttt{coro}.\,\texttt{letcc}\,k\,\texttt{in}\,\texttt{throw}\,\langle s, \texttt{fold}(k)\rangle\,\texttt{to}\,\texttt{unfold}(r'))$$

When applied, `resume` seizes the current continuation, and passes the state, s, and the seized continuation (packaged as a coroutine) to the called coroutine.

But how do we create a system of coroutines in the first place? Since the state is explicitly passed from one routine to the other, a coroutine may be defined as a state transformation function that, when activated with the current state, determines the next state of the computation. A system of coroutines is created by establishing a joint exit point to which the result of the system is thrown, and creating a pair of coroutines that iteratively transform the state and pass control to the partner routine. If either routine wishes to terminate the computation, it does so by throwing a result value to their common exit point. Thus, a coroutine may be specified by a function of type

$$(\rho, \tau)\,\texttt{rout} \triangleq \rho\,\texttt{cont} \to \tau \to \tau,$$

where ρ is the result type and τ is the state type of the system of coroutines.

The set up a system of coroutines we define a function `run` that, given two routines, creates a function of type $\tau \to \rho$ that, when applied to the

initial state, computes a result of type ρ. The computation consists of a cooperating pair of routines that share a common exit point. The definition of `run` begins as follows:

$$\lambda\,\langle r_1, r_2\rangle.\,\lambda\,s_0.\,\texttt{letcc}\,x_0\,\texttt{in}\,\texttt{let}\,r_1'\,\texttt{be}\,r_1(x_0)\,\texttt{in}\,\texttt{let}\,r_2'\,\texttt{be}\,r_2(x_0)\,\texttt{in}\ldots$$

Given two routines, `run` establishes their common exit point, and passes this continuation to both routines. By throwing to this continuation either routine may terminate the computation with a result of type ρ. The body of the `run` function continues as follows:

$$\texttt{rep}(r_2')(\texttt{letcc}\,k\,\texttt{in}\,\texttt{rep}(r_1')(\langle s_0, \texttt{fold}(k)\rangle))$$

The auxiliary function `rep` creates an infinite loop that transforms the state and passes control to the other routine:

$$\lambda\,t.\,\texttt{fix}\,l\,\texttt{is}\,\lambda\,\langle s, r\rangle.\,l(\texttt{resume}(\langle t(s), r\rangle)).$$

The system is initialized by starting routine r_1 with the initial state, and arranging that, when it cedes control to its partner, it starts routine r_2 with the resulting state. At that point the system is bootstrapped: each routine will resume the other on each iteration of the loop.

A good example of coroutining arises whenever we wish to interleave input and output in a computation. We may achieve this using a coroutine between a *producer* routine and a *consumer* routine. The producer emits the next element of the input, if any, and passes control to the consumer with that element removed from the input. The consumer processes the next data item, and returns control to the producer, with the result of processing attached to the output. The input and output are modeled as lists of type τ_i `list` and τ_o `list`, respectively, which are passed back and forth between the routines.[1] The routines exchange messages according to the following protocol. The message $\texttt{OK}(\langle i, o\rangle)$ is sent from the consumer to producer to acknowledge receipt of the previous message, and to pass back the current state of the input and output channels. The message $\texttt{EMIT}(\langle v, \langle i, o\rangle\rangle)$, where v is a value of type τ_i `opt`, is sent from the producer to the consumer to emit the next value (if any) from the input, and to pass the current state of the input and output channels to the consumer.

This leads to the following implementation of the producer/consumer model. The type τ of the state maintained by the routines is the labelled

[1] In practice the input and output state are implicit, but we prefer to make them explicit for the sake of clarity.

sum type

$$[\mathtt{OK} : \tau_i\ \mathtt{list} \times \tau_o\ \mathtt{list}, \mathtt{EMIT} : \tau_i\ \mathtt{opt} \times (\tau_i\ \mathtt{list} \times \tau_o\ \mathtt{list})].$$

This type specifies the message protocol between the producer and the consumer described in the preceding paragraph.

The producer, P, is defined by the expression

$$\lambda\, x_0.\, \lambda\, msg.\, \mathtt{case}\, msg\, \{b_1 \mid b_2 \mid b_3\},$$

where the first branch, b_1, is

$$\mathtt{OK} \cdot \langle \mathtt{nil}, os \rangle \Rightarrow \mathtt{EMIT} \cdot \langle \mathtt{null}, \langle \mathtt{nil}, os \rangle \rangle$$

and the second branch, b_2, is

$$\mathtt{OK} \cdot \langle \mathtt{cons}(i; is), os \rangle \Rightarrow \mathtt{EMIT} \cdot \langle \mathtt{just}(i), \langle is, os \rangle \rangle,$$

and the third branch, b_3, is

$$\mathtt{EMIT} \cdot \_ \Rightarrow \mathtt{error}.$$

In words, if the input is exhausted, the producer emits the value `null`, along with the current channel state. Otherwise, it emits `just(`i`)`, where i is the first remaining input, and removes that element from the passed channel state. The producer cannot see an `EMIT` message, and signals an error if it should occur.

The consumer, C, is defined by the expression

$$\lambda\, x_0.\, \lambda\, msg.\, \mathtt{case}\, msg\, \{b'_1 \mid b'_2 \mid b'_3\},$$

where the first branch, b'_1, is

$$\mathtt{EMIT} \cdot \langle \mathtt{null}, \langle \_, os \rangle \rangle \Rightarrow \mathtt{throw}\, os\, \mathtt{to}\, x_0,$$

the second branch, b'_2, is

$$\mathtt{EMIT} \cdot \langle \mathtt{just}(i), \langle is, os \rangle \rangle \Rightarrow \mathtt{OK} \cdot \langle is, \mathtt{cons}(f(i); os) \rangle,$$

and the third branch, b'_3, is

$$\mathtt{OK} \cdot \_ \Rightarrow \mathtt{error}.$$

The consumer dispatches on the emitted datum. If it is absent, the output channel state is passed to x_0 as the ultimate value of the computation. If

it is present, the function f (unspecified here) of type $\tau_i \to \tau_o$ is applied to transform the input to the output, and the result is added to the output channel. If the message `OK` is received, the consumer signals an error, as the producer never produces such a message.

The initial state, s_0, has the form $\texttt{OK} \cdot \langle is, os \rangle$, where is and os are the initial input and output channel state, respectively. The computation is created by the expression

$$\texttt{run}(\langle P, C \rangle)(s_0),$$

which sets up the coroutines as described earlier.

While it is relatively easy to visualize and implement coroutines involving only two partners, it is more complex, and less useful, to consider a similar pattern of control among $n \geq 2$ participants. In such cases it is more common to structure the interaction as a collection of n routines, each of which is a coroutine of a central *scheduler*. When a routine resumes its partner, it passes control to the scheduler, which determines which routine to execute next, again as a coroutine of itself. When structured as coroutines of a scheduler, the individual routines are called *threads*. A thread *yields* control by resuming its partner, the scheduler, which then determines which thread to execute next as a coroutine of itself. This pattern of control is called *cooperative multi-threading*, since it is based on explicit yields, rather than implicit yields imposed by asynchronous events such as timer interrupts.

31.4 Notes

Continuations are a ubiquitous notion in programming languages. Reynolds's survey [83] provides an excellent account of the history and literature on continuations. The account given here draws on the work of Felleisen [27].