Part IX

Control Flow

Chapter 27

Control Stacks

The technique of specifying the dynamic semantics as a transition system 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, $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, for the language $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$. 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 $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ with the structural operational semantics of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$.

27.1 Machine Definition

A state, s, of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ 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 $k \lhd e$, 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*:

$$\frac{}{\epsilon \ \mathsf{stack}} \tag{27.1a}$$

$$\frac{f \ \mathsf{frame} \quad k \ \mathsf{stack}}{k;f \ \mathsf{stack}} \tag{27.1b}$$

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:

$$\frac{}{\mathtt{s(-)} \ \mathsf{frame}} \tag{27.2a}$$

$$\frac{}{\mathtt{ifz(-;}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} \ \mathsf{frame}} \tag{27.2b}$$

$$\frac{}{\mathtt{ap(-;}e_2\mathtt{)} \ \mathsf{frame}} \tag{27.2c}$$

The frames correspond to rules with transition premises in the dynamic semantics 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 $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ is inductively defined by a set of inference rules. We begin with the rules for natural numbers.

$$\frac{}{k \rhd \mathtt{z} \mapsto k \lhd \mathtt{z}} \tag{27.3a}$$

$$\frac{}{k \rhd \mathtt{s(}e\mathtt{)} \mapsto k;\mathtt{s(-)} \rhd e} \tag{27.3b}$$

$$\frac{}{k;\mathtt{s(-)} \lhd e \mapsto k \lhd \mathtt{s(}e\mathtt{)}} \tag{27.3c}$$

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

Next, we consider the rules for case analysis.

$$\frac{}{k \triangleright \mathtt{ifz}(e;e_1;x.e_2) \mapsto k;\mathtt{ifz}(-;e_1;x.e_2) \triangleright e} \tag{27.4a}$$

$$\frac{}{k;\mathtt{ifz}(-;e_1;x.e_2) \triangleleft \mathtt{z} \mapsto k \triangleright e_1} \tag{27.4b}$$

$$\frac{}{k;\mathtt{ifz}(-;e_1;x.e_2) \triangleleft \mathtt{s}(e) \mapsto k \triangleright [e/x]e_2} \tag{27.4c}$$

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.

$$\frac{}{k \triangleright \mathtt{lam}[\tau](x.e) \mapsto k \triangleleft \mathtt{lam}[\tau](x.e)} \tag{27.5a}$$

$$\frac{}{k \triangleright \mathtt{ap}(e_1;e_2) \mapsto k;\mathtt{ap}(-;e_2) \triangleright e_1} \tag{27.5b}$$

$$\frac{}{k;\mathtt{ap}(-;e_2) \triangleleft \mathtt{lam}[\tau](x.e) \mapsto k \triangleright [e_2/x]e} \tag{27.5c}$$

$$\frac{}{k \triangleright \mathtt{fix}[\tau](x.e) \mapsto k \triangleright [\mathtt{fix}[\tau](x.e)/x]e} \tag{27.5d}$$

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 42 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:

$$\frac{}{\epsilon \triangleright e \ \mathsf{initial}} \tag{27.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}} \tag{27.6b}$$

27.2 Safety

To define and prove safety for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ 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:

$$\frac{}{\epsilon : \tau} \tag{27.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{k;f : \tau} \tag{27.7b}$$

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'.

$$\frac{}{\texttt{s(}-\texttt{)} : \texttt{nat} \Rightarrow \texttt{nat}} \tag{27.8a}$$

$$\frac{e_1 : \tau \quad x : \texttt{nat} \vdash e_2 : \tau}{\texttt{ifz(}-\texttt{;}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : \texttt{nat} \Rightarrow \tau} \tag{27.8b}$$

$$\frac{e_2 : \tau_2}{\texttt{ap(}-\texttt{;}e_2\texttt{)} : \texttt{arr(}\tau_2\texttt{;}\tau\texttt{)} \Rightarrow \tau} \tag{27.8c}$$

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

$$\frac{k : \tau \quad e : \tau}{k \rhd e \ \mathsf{ok}} \tag{27.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \lhd e \ \mathsf{ok}} \tag{27.9b}$$

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

Theorem 27.1 (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 $s \mapsto s'$.

27.3 Correctness of the Control Machine

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

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

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, $\texttt{ap}(e_1;e_2)$, then the first step of the machine is

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

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 $e = \texttt{ap}(e_1;e_2)$, $e' = \texttt{ap}(e_1';e_2)$, with $e_1 \mapsto e_1'$. We are given that $k \triangleright \texttt{ap}(e_1';e_2) \mapsto^* k \triangleleft v$, and we are to show that $k \triangleright \texttt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

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

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 semantics described in Chapter 12, which has the property that $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

Lemma 27.2. *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 12.2 on page 97 for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, 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 $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ machine state as encoding an expression, and show that $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ transitions are simulated by $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ 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 27.3. *If $s \mapsto s'$, $s \looparrowright e$, and $s' \looparrowright e'$, then $e \mapsto^* e'$.*

Corollary 27.4. *$e \mapsto^* \overline{n}$ 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.

27.3.1 Completeness

Proof of Lemma 27.2. The proof is by induction on an evaluation semantics for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \texttt{lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{)} \quad [e_2/x]e \Downarrow v}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow v} \tag{27.10}$$

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

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

The other cases of the proof are handled similarly. □

27.3.2 Soundness

The judgement $s \looparrowright e'$, 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:

$$\frac{k \bowtie e = e'}{k \triangleright e \looparrowright e'} \tag{27.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \looparrowright e'} \tag{27.11b}$$

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

$$\frac{}{\epsilon \bowtie e = e} \tag{27.12a}$$

$$\frac{k \bowtie \texttt{s}(e) = e'}{k;\texttt{s}(-) \bowtie e = e'} \tag{27.12b}$$

$$\frac{k \bowtie \texttt{ifz}(e_1;e_2;x.e_3) = e'}{k;\texttt{ifz}(-;e_2;x.e_3) \bowtie e_1 = e'} \tag{27.12c}$$

$$\frac{k \bowtie \texttt{ap}(e_1;e_2) = e}{k;\texttt{ap}(-;e_2) \bowtie e_1 = e} \tag{27.12d}$$

These judgements both define total functions.

Lemma 27.5. *The judgement $s \looparrowright e$ 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 27.6. *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 $e = \texttt{ap}(e_1;e_2)$, $e' = \texttt{ap}(e_1';e_2)$, and $e_1 \mapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from Rules (27.12) that $k;\texttt{ap}(-;e_2) \bowtie e_1 = d$ and $k;\texttt{ap}(-;e_2) \bowtie e_1' = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \texttt{ap}(\texttt{lam}[\tau_2](x.e);e_2)$ 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 $k = k';\texttt{ap}(-;c_2)$. It follows from Rules (27.12) that $k' \bowtie \texttt{ap}(e;c_2) = d$ and $k' \bowtie \texttt{ap}(e';c_2) = d'$. But by the SOS rules $\texttt{ap}(e;c_2) \mapsto \texttt{ap}(e';c_2)$, 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 27.3 on page 246.

Proof of Lemma 27.3 on page 246. The proof is by case analysis on the transitions of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$. In each case after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$.

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

Suppose that $s = k;\texttt{ap}(\texttt{lam}[\tau](x.e_1);-) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k;\texttt{ap}(\texttt{lam}[\tau](x.e_1);-) \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \texttt{ap}(\texttt{lam}[\tau](x.e_1);e_2) = e'$. The result follows from Lemma 27.6.

□

27.4 Exercises

Chapter 28

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. All too often explicit checks are omitted (by design or neglect), whereas exceptions cannot be ignored.

28.1 Failures

To begin with let us consider a simple control mechanism, which permits the evaluation of an expression to *fail* by passing control to the nearest enclosing handler, which is said to *catch* the failure. Failures are a simplified form of exception in which no value is associated with the failure. This allows us to concentrate on the control flow aspects, and to treat the associated value separately.

The following grammar describes an extension to $\mathcal{L}\{\rightarrow\}$ to include failures:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	$::=$	$\texttt{fail[}\tau\texttt{]}$	$\texttt{fail}$
		$\mid$	$\texttt{catch(}e_1\texttt{;}e_2\texttt{)}$	$\texttt{try}\,e_1\,\texttt{ow}\,e_2$

The expression $\texttt{fail[}\tau\texttt{]}$ aborts the current evaluation. The expression $\texttt{catch(}e_1\texttt{;}e_2\texttt{)}$

evaluates e_1. If it terminates normally, its value is returned; if it fails, its value is the value of e_2.

The static semantics of failures is quite straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail[}\tau\texttt{]} : \tau} \tag{28.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch(}e_1\texttt{;}e_2\texttt{)} : \tau} \tag{28.1b}$$

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

The dynamic semantics of failures uses a technique called *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.

This behavior is naturally specified using the abstract machine $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ from Chapter 27, because it makes the control stack explicit. We introduce a new form of state, $k \blacktriangleleft$, which passes a failure to the stack, k, in search of the nearest enclosing handler. A state of the form $\epsilon \blacktriangleleft$ is considered final, rather than stuck; it corresponds to an "uncaught failure" making its way to the top of the stack.

The set of frames is extended with the following additional rule:

$$\frac{e_2 \ \mathsf{exp}}{\texttt{catch(}-\texttt{;}e_2\texttt{)} \ \mathsf{frame}} \tag{28.2}$$

The transition rules of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ are extended with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail[}\tau\texttt{]} \mapsto k \blacktriangleleft} \tag{28.3a}$$

$$\frac{}{k \triangleright \texttt{catch(}e_1\texttt{;}e_2\texttt{)} \mapsto k\texttt{;catch(}-\texttt{;}e_2\texttt{)} \triangleright e_1} \tag{28.3b}$$

$$\frac{}{k\texttt{;catch(}-\texttt{;}e_2\texttt{)} \triangleleft v \mapsto k \triangleleft v} \tag{28.3c}$$

$$\frac{}{k\texttt{;catch(}-\texttt{;}e_2\texttt{)} \blacktriangleleft \ \mapsto k \triangleright e_2} \tag{28.3d}$$

$$\frac{(f \neq \texttt{catch(}-\texttt{;}e_2\texttt{)})}{k\texttt{;}f \blacktriangleleft \ \mapsto k \blacktriangleleft} \tag{28.3e}$$

Evaluating `fail[`τ`]` propagates a failure up the stack. Evaluating `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 $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\epsilon \lhd e\ \mathsf{final}} \tag{28.4a}$$

$$\frac{}{\epsilon \blacktriangleleft\ \mathsf{final}} \tag{28.4b}$$

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 27 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 28.1 (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 $s \mapsto s'$.

28.2 Exceptions

Let us now consider enhancing the simple failures mechanism of the preceding section with an exception mechanism that permits a value to be associated with the failure, which is then passed to the handler as part of the control transfer. The syntax of exceptions is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	`raise[`τ`](`e`)`	`raise(`e`)`
		|	`handle(`e_1`;`x`.`e_2`)`	`try` e_1 `ow` $x \Rightarrow e_2$

The argument to `raise` is evaluated to determine the value passed to the handler. The expression `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 dynamic semantics of exceptions is a mild generalization of that of failures given in Section 28.1 on page 249. The failure state, $k \blacktriangleleft$, is extended to permit passing a value along with the failure, $k \blacktriangleleft e$, where e val. Stack frames include these two forms:

$$\frac{}{\texttt{raise[}\tau\texttt{](}-\texttt{)}\ \mathsf{frame}} \tag{28.5a}$$

$$\frac{}{\texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)}\ \mathsf{frame}} \tag{28.5b}$$

The rules for evaluating exceptions are as follows:

$$\frac{}{k \triangleright \texttt{raise[}\tau\texttt{](}e\texttt{)} \mapsto k\texttt{;raise[}\tau\texttt{](}-\texttt{)} \triangleright e} \tag{28.6a}$$

$$\frac{}{k\texttt{;raise[}\tau\texttt{](}-\texttt{)} \triangleleft e \mapsto k \blacktriangleleft e} \tag{28.6b}$$

$$\frac{}{k\texttt{;raise[}\tau\texttt{](}-\texttt{)} \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{28.6c}$$

$$\frac{}{k \triangleright \texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto k\texttt{;handle(}-\texttt{;}x\texttt{.}e_2\texttt{)} \triangleright e_1} \tag{28.6d}$$

$$\frac{}{k\texttt{;handle(}-\texttt{;}x\texttt{.}e_2\texttt{)} \triangleleft e \mapsto k \triangleleft e} \tag{28.6e}$$

$$\frac{}{k\texttt{;handle(}-\texttt{;}x\texttt{.}e_2\texttt{)} \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{28.6f}$$

$$\frac{(f \neq \texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)})}{k\texttt{;}f \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{28.6g}$$

The static semantics of exceptions generalizes that of failures.

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise[}\tau\texttt{](}e\texttt{)} : \tau} \tag{28.7a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : \tau} \tag{28.7b}$$

These rules are parameterized by the type of values associated with exceptions, τ_{exn}. But 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 the expression that it guards. If different exceptions could have different associated values, the handler could not predict (statically) what type of value to expect, and hence could not dispatch on it without violating type safety.

The reason to associate data with an exception is to communicate to the handler some information about the use of the exceptional condition. But what should the type of this data be? A very naïve suggestion might be to choose 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. The trouble with this, of course, is that all information to be passed to the handler must be encoded as a string, and the handler must parse the string to recover that information!

Another all-too-familiar choice of τ_{exn} is the type `nat`. Exception conditions are encoded, by convention, as natural numbers.[1] This is obviously an impractical approach, since it requires that each system maintain a global assignment of numbers to error conditions, impeding or even precluding modular development. Moreover, the decoding of the error numbers is tedious and error prone. Surely there is a better way!

A more practical choice for τ_{exn} would be a distinguished labelled sum type of the form

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

with one class for each exceptional condition and an associated data value of the type associated to that class in τ_{exn}. This allows the handler to perform a simple symbolic case analysis on the class of the exception to recover the underlying data. For example, we might write

```
try $e_1$ ow x $\Rightarrow$
  case x {
    div $\langle\rangle$ $\Rightarrow$ $e_{div}$
  | fnf $s$ $\Rightarrow$ $e_{fnf}$
  | ... }
```

to recover from the exceptions specified in τ_{exn}.

The chief difficulty with this approach is that, like error numbers, it requires a single global commitment to the type τ_{exn} that must be shared by all components of the program. This impedes separate development, and

[1] In Unix these are called `errno`'s, for *error numbers*, with `0` being the number for "no error."

requires all modules to be aware of all exceptions that may be raised anywhere within the program. The solution to this is to employ a *dynamically extensible* sum type for τ_{exn} that allows new classes to be generated from anywhere within the program in such a way that each component is assured to be allocated different classes from those generated elsewhere in the program.

Since extensible sums have application beyond serving as the type of exception values, we defer a detailed discussion to Chapter 36, which discusses them in isolation from exceptions.

28.3 Exercises

Chapter 29

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 *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. 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.

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.

29.1 Informal Overview

We will extend $\mathcal{L}\{\rightarrow\}$ with the type $\texttt{cont}(\tau)$ of continuations accepting values of type τ. The introduction form for $\texttt{cont}(\tau)$ is $\texttt{letcc}[\tau](x.e)$, 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](e_1;e_2), 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 $\rightarrow$ 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 $\mathcal{L}\{$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 $\tau' \rightarrow \tau$, 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!

29.2 Semantics of Continuations

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	`cont(`τ`)`	τ `cont`
Expr	e	$::=$	`letcc[`τ`](`$x.e$`)`	`letcc` x `in` e
		$\mid$	`throw[`τ`](`$e_1;e_2$`)`	`throw` e_1 `to` e_2
		$\mid$	`cont(`k`)`	

The expression `cont(`k`)` is a reified control stack; they arise during evaluation, but are not available as expressions to the programmer.

The static semantics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau'](e_1;e_2) : \tau'} \tag{29.1b}$$

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

The static semantics of continuation values is given by the following rule:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{29.2}$$

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 dynamic semantics, we extend $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{e_2 \ \mathsf{exp}}{\texttt{throw}[\tau](-;e_2) \ \mathsf{frame}} \tag{29.3a}$$

$$\frac{e_1 \ \mathsf{val}}{\texttt{throw}[\tau](e_1;-) \ \mathsf{frame}} \tag{29.3b}$$

Every reified control stack is a value:

$$\frac{k \ \mathsf{stack}}{\texttt{cont}(k) \ \mathsf{val}} \tag{29.4}$$

The transition rules for the continuation constructs are as follows:

$$\frac{}{k \triangleright \texttt{letcc}[\tau](x.e) \mapsto k \triangleright [\texttt{cont}(k)/x]e} \tag{29.5a}$$

$$\frac{}{k;\texttt{throw}[\tau](v;-) \triangleleft \texttt{cont}(k') \mapsto k' \triangleleft v} \tag{29.5b}$$

$$\frac{}{k \triangleright \texttt{throw}[\tau](e_1;e_2) \mapsto k;\texttt{throw}[\tau](-;e_2) \triangleright e_1} \tag{29.5c}$$

$$\frac{e_1 \ \mathsf{val}}{k;\texttt{throw}[\tau](-;e_2) \triangleleft e_1 \mapsto k;\texttt{throw}[\tau](e_1;-) \triangleright e_2} \tag{29.5d}$$

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 $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ given in Chapter 27.

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

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}[\tau](-;e_2) : \tau \Rightarrow \tau'} \tag{29.6a}$$

$$\frac{e_1 : \tau \quad e_1\ \mathsf{val}}{\mathtt{throw}[\tau](e_1;-) : \mathtt{cont}(\tau) \Rightarrow \tau'} \tag{29.6b}$$

The rest of the definitions remain as in Chapter 27.

Lemma 29.1 (Canonical Forms). *If e : $\mathtt{cont}(\tau)$ and e val, then $e = \mathtt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 29.2 (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 $s \mapsto s'$.*

29.3 Coroutines

A *subroutine* is a pattern of control flow in a program in which one routine passes control to another by passing to it a data value, the *argument*, to the subroutine, and a *return point* at which to resume control when finished. This arrangement is asymmetric in that the caller passes a return point to the callee, but upon return the callee simply branches to that control point without passing any return information. A *coroutine* is a symmetric pattern of control flow in which each routine passes to the other a return point of the call. The asymmetric call/return pattern is symmetrized to a call/call pattern in which each routine is effectively a subroutine of the other. (This raises an interesting question of how the interaction commences, which we will discuss in more detail below.)

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.

To see how coroutines are implemented in terms of continuations, it is best to think of the "steady state" interaction between the two routines, leaving the initialization phase to be discussed separately. A routine is represented by a continuation that, when invoked, is passed a data item, whose type is shared between the two routines, and a return continuation, which represents the partner routine. Crucially, the argument type of the other continuation is again of the very same form, consisting of a data item and another return continuation. If we think of the coroutine as a *trajectory* through a succession of such continuations, then the *state* of the continuation (which changes as the interaction progresses) satisfies the type isomorphism

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

where τ is the type of the data values exchanged by the routines. The solution to such an isomorphism is, of course, the recursive type

$$\texttt{state} = \mu t\,.\,(\tau \times t)\,\texttt{cont}.$$

Thus a state, s, encapsulates a pair consisting of a value of type τ together with another state.

The routines pass control from one to the other by calling the function `resume` of type

$$\tau \times \texttt{state} \rightarrow \tau \times \texttt{state}.$$

That is, given a datum, d, a state, s, the application $\texttt{resume}(\langle d,s\rangle)$ passes d and its own return address to the routine represented by the state s. The function `resume` is defined by the following expression:

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

When applied, this function seizes the current continuation, and passes the given datum and this continuation to the partner routine, using the isomorphism between `state` and $(\tau \times \texttt{state})\,\texttt{cont}$.

The general form of a coroutine consists of an infinite loop which, on each iteration, takes a datum, d, and a state, s, performs a transformation on d, resuming its partner routine with the result, d', of the transformation. The function `corout` builds a coroutine from a data transformation routine; it has type

$$(\tau \rightarrow \tau) \rightarrow (\tau \times \texttt{state}) \rightarrow \tau'.$$

The result type, τ', is arbitrary, since the routine never returns to the call site. (A coroutine is shut down by an explicit exit operation, which will be specified shortly.) The function `corout` is defined by the following expression (with types omitted for concision):

$$\lambda \mathit{next}.\,\texttt{fix}\,\mathit{loop}\,\texttt{is}\,\lambda\langle d,s\rangle.\,\mathit{loop}\texttt{(resume(}\langle \mathit{next}\texttt{(}d\texttt{)},s\rangle\texttt{))}.$$

Each time through the loop, the partner routine, s, is resumed with the updated datum given by applying *next* to the current datum, d.

Let ρ be the ultimate type of a computation consisting of two interacting coroutines that exchanges values of type τ during their execution. The function `run`, which has type

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

takes an initial value of type τ and two routines, each of type

$$\rho\,\texttt{cont} \to \tau \to \tau,$$

and builds a coroutine of type ρ from them. The first argument to each routine is the exit point, and the result is a data transformation operation. The definition of `run` begins as follows:

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

First, `run` establishes an exit point that is passed to the two routines to obtain their data transformation components. This allows either or both of the routines to terminate the computation by throwing the ultimate result value to `exit`. The implementation of `run` continues as follows:

$$\texttt{corout(}r_2'\texttt{)(letcc}\,k\,\texttt{in}\,\texttt{corout(}r_1'\texttt{)(}\langle \mathit{init},\texttt{fold(}k\texttt{)}\rangle\texttt{))}$$

The routine r_1' is called with the initial datum, *init*, and the state `fold(`k`)`, where k is the continuation corresponding to the call to r_2'. The first `resume` from the coroutine built from r_1' will cause the coroutine built from r_2' to be initiated. At this point the steady state behavior is in effect, with the two routines exchanging control using `resume`. Either may terminate the computation by throwing a result value, v, of type ρ to the continuation *exit*.

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 $\tau_i\,\texttt{list}$ and $\tau_o\,\texttt{list}$, 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 $\tau_i\,\texttt{opt}$, 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 data exchanged by the routines is the labelled sum type

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

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

The producer, `producer`, is defined by the expression

$$\lambda \mathit{exit}.\,\lambda \mathit{msg}.\,\texttt{case}\,\mathit{msg}\,\{b_1 \mid b_2 \mid b_3\},$$

where the first branch, b_1, is

$$\texttt{in[OK]}(\langle\texttt{nil}, \mathit{os}\rangle) \Rightarrow \texttt{in[EMIT]}(\langle\texttt{null}, \langle\texttt{nil}, \mathit{os}\rangle\rangle)$$

and the second branch, b_2, is

$$\texttt{in[OK]}(\langle\texttt{cons}(i;\mathit{is}), \mathit{os}\rangle) \Rightarrow \texttt{in[EMIT]}(\langle\texttt{just}(i), \langle\mathit{is}, \mathit{os}\rangle\rangle),$$

and the third branch, b_3, is

$$\texttt{in[EMIT]}(\_) \Rightarrow \texttt{error}.$$

In words, if the input is exhausted, the producer emits the value `null`, along with the current channel state. Otherwise, it emits $\texttt{just}(i)$, where i is the first remaining input, and removes that element from the passed channel

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

state. The producer cannot see an `EMIT` message, and signals an error if it should occur.

The consumer, `consumer`, is defined by the expression

$$\lambda exit.\,\lambda msg.\,\texttt{case}\,msg\,\{b_1 \mid b_2 \mid b_3\},$$

where the first branch, b_1, is

$$\texttt{in[EMIT]}(\langle\texttt{null},\langle\_,os\rangle\rangle) \Rightarrow \texttt{throw}\,os\,\texttt{to}\,exit,$$

the second branch, b_2, is

$$\texttt{in[EMIT]}(\langle\texttt{just}(i),\langle is,os\rangle\rangle) \Rightarrow \texttt{in[OK]}(\langle is,\texttt{cons}(f(i);os)\rangle),$$

and the third branch, b_3, is

$$\texttt{in[OK]}(\_) \Rightarrow \texttt{error}.$$

The consumer dispatches on the emitted datum. If it is absent, the output channel state is passed to *exit* as the ultimate value of the computation. If it is present, the function f (unspecified here) of type $\tau_i \rightarrow \tau_o$ 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 datum, `init`, has the form $\texttt{in[OK]}(\langle is,os\rangle)$, where *is* and *os* are the initial input and output channel state, respectively. The computation is created by the expression

$$\texttt{run(init)}(\langle\texttt{producer},\texttt{consumer}\rangle),$$

which sets up the coroutines as described earlier.

29.4 Exercises

1. Study the short-circuit multiplication example carefully to be sure you understand why it works!
2. Attempt to solve the problem of composing a continuation with a function yourself, before reading the solution.
3. Simulate the evaluation of `compose` (f, k) on the empty stack. Observe that the control stack substituted for x is

$$\epsilon;\texttt{throw}[\tau](-;k);\texttt{ap}(f;-)$$

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