29 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 signal the need for special handling in unusual circumstances. We could use conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly because the transfer to the handler is conceptually direct and immediate, rather than indirect via explicit checks.

In this chapter, we will consider two extensions of **PCF** with exceptions. The first, **FPCF**, enriches **PCF** with the simplest form of exception, called a *failure*, with no associated data. A failure can be intercepted and turned into a success (or another failure!) by transferring control to another expression. The second, **XPCF**, enriches **PCF** with *exceptions*, with associated data that is passed to an exception handler that intercepts it. The handler may analyze the associated data to determine how to recover from the exceptional condition. A key choice is to decide on the type of the data associated to an exception.

29.1 Failures

The syntax of **FPCF** is defined by the following extension of the grammar of **PCF**:

Exp	e	::=	$\mathtt{fail}$	$\mathtt{fail}$	signal a failure
			$\mathtt{catch}(e_1;e_2)$	$\mathtt{catch}\,e_1\,\mathtt{ow}\,e_2$	catch a failure

The expression $\mathtt{fail}$ aborts the current evaluation, and the expression $\mathtt{catch}(e_1;e_2)$ catches any failure in e_1 by evaluating e_2 instead. Either e_1 or e_2 may themselves abort, or they may diverge or return a value as usual in **PCF**.

The statics of **FPCF** is given by these rules:

$$\frac{}{\Gamma \vdash \mathtt{fail} : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{catch}(e_1;e_2) : \tau} \tag{29.1b}$$

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

https://doi.org/10.1017/CBO9781316576892.031 Published online by Cambridge University Press

The dynamics of **FPCF** is given using a technique called *stack unwinding*. Evaluation of a `catch` pushes a frame of the form $\texttt{catch}(-;e)$ onto the control stack that awaits the arrival of a failure. Evaluation of a `fail` expression pops frames from the control stack until it reaches a frame of the form $\texttt{catch}(-;e)$, at which point the frame is removed from the stack and the expression e is evaluated. Failure propagation is expressed by a state of the form $k \blacktriangleleft$, which extends the two forms of state considered in Chapter 28 to express failure propagation.

The **FPCF** machine extends the **PCF** machine with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail} \longmapsto k \blacktriangleleft} \tag{29.2a}$$

$$\frac{}{k \triangleright \texttt{catch}(e_1;e_2) \longmapsto k;\texttt{catch}(-;e_2) \triangleright e_1} \tag{29.2b}$$

$$\frac{}{k;\texttt{catch}(-;e_2) \triangleleft v \longmapsto k \triangleleft v} \tag{29.2c}$$

$$\frac{}{k;\texttt{catch}(-;e_2) \blacktriangleleft \longmapsto k \triangleright e_2} \tag{29.2d}$$

$$\frac{(f \neq \texttt{catch}(-;e))}{k;f \blacktriangleleft \longmapsto k \blacktriangleleft} \tag{29.2e}$$

Evaluating `fail` propagates a failure up the stack. The act of failing itself, `fail`, will, of course, give rise to a failure. Evaluating $\texttt{catch}(e_1;e_2)$ consists of pushing the handler on the control stack and evaluating e_1. If a value reaches to the handler, the handler is removed and the value is passed to the surrounding frame. If a failure reaches the handler, the stored expression is evaluated with the handler removed from the control stack. Failures propagate through all frames other than the `catch` frame.

The initial and final states of the **FPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \text{ initial}} \tag{29.3a}$$

$$\frac{e \text{ val}}{\epsilon \triangleleft e \text{ final}} \tag{29.3b}$$

$$\frac{}{\epsilon \blacktriangleleft \text{ final}} \tag{29.3c}$$

The definition of stack typing given in Chapter 28 can be extended to account for the new forms of frame so that safety can be proved in the same way as before. The only difference is that the statement of progress must be weakened to take account of failure: a well-typed expression is either a value, or may take a step, or may signal failure.

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

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

https://doi.org/10.1017/CBO9781316576892.031 Published online by Cambridge University Press

29.2 Exceptions

The language **XPCF** enriches **FPCF** with *exceptions*, failures to which a value is attached. The syntax of **XPCF** extends that of **PCF** with the following forms of expression:

Exp	e	$::=$	$\mathtt{raise}(e)$	$\mathtt{raise}(e)$	raise an exception
			$\mathtt{try}(e_1; x.e_2)$	$\mathtt{try}\, e_1\, \mathtt{ow}\, x \hookrightarrow e_2$	handle an exception

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

The statics of exceptions extends the statics of failures to account for the type of the value carried with the exception:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \tag{29.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\mathsf{exn}} \vdash e_2 : \tau}{\Gamma \vdash \mathtt{try}(e_1; x.e_2) : \tau} \tag{29.4b}$$

The type τ_{exn} is some fixed, but as yet unspecified, type of exception values. (The choice of τ_{exn} is discussed in Section 29.3.)

The dynamics of **XPCF** is similar to that of **FPCF**, except that the failure state $k \blacktriangleleft$ is replaced by the exception state $k \blacktriangleleft e$ which passes an exception value e to the stack k. There is only one notion of exception, but the associated value can be used to identify the source of the exception. We use a by-value interpretation to avoid the problem of *imprecise exceptions* that arises under a by-name interpretation.

The stack frames of the **PCF** machine are extended to include $\mathtt{raise}(-)$ and $\mathtt{try}(-; x.e_2)$. These are used in the following rules:

$$\frac{}{k \triangleright \mathtt{raise}(e) \longmapsto k;\mathtt{raise}(-) \triangleright e} \tag{29.5a}$$

$$\frac{}{k;\mathtt{raise}(-) \triangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5b}$$

$$\frac{}{k \triangleright \mathtt{try}(e_1; x.e_2) \longmapsto k;\mathtt{try}(-; x.e_2) \triangleright e_1} \tag{29.5c}$$

$$\frac{}{k;\mathtt{try}(-; x.e_2) \triangleleft e \longmapsto k \triangleleft e} \tag{29.5d}$$

$$\frac{}{k;\mathtt{try}(-; x.e_2) \blacktriangleleft e \longmapsto k \triangleright [e/x]e_2} \tag{29.5e}$$

$$\frac{(f \neq \mathtt{try}(-; x.e_2))}{k; f \blacktriangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5f}$$

https://doi.org/10.1017/CBO9781316576892.031 Published online by Cambridge University Press

The main difference compared to rules (29.2) is that an exception passes a values to the stack, whereas a failure does not.

The initial and final states of the **XPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \triangleright e \text{ initial}} \tag{29.6a}$$

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

$$\frac{}{\epsilon \blacktriangleleft e \text{ final}} \tag{29.6c}$$

Theorem 29.2 (Safety for **XPCF**). *1. If s ok and $s \longmapsto s'$, then s' ok.*
2. If s ok, then either s final or there exists s' such that $s \longmapsto s'$.

29.3 Exception Values

The statics of **XPCF** is parameterized by the type τ_{exn} of values associated to exceptions. The choice of τ_{exn} is important because it determines how the source of an exception is identified in a program. If τ_{exn} is the one-element type `unit`, then exceptions degenerate to failures, which are unable to identify their source. Thus, τ_{exn} must have more than one value to be useful.

This fact suggests that τ_{exn} should be a finite sum. The classes of the sum identify the sources of exceptions, and the classified value carries information about the particular instance. For example, τ_{exn} might be a sum type of the form

$$[\mathtt{div} \hookrightarrow \mathtt{unit}, \mathtt{fnf} \hookrightarrow \mathtt{string}, \ldots].$$

Here 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 that was not found.

Using a sum means that an exception handler can dispatch on the class of the exception value to identify its source and cause. For example, we might write

```
handle e1 ow x ↪
  match x {
    div ⟨⟩ ↪ ediv
  | fnf s ↪ efnf }
```

to handle the exceptions specified by the above sum type. Because the exception and its associated data are coupled in a sum type, there is no possibility of misinterpreting the data associated to one exception as being that of another.

https://doi.org/10.1017/CBO9781316576892.031 Published online by Cambridge University Press

The disadvantage of choosing a finite sum for τ_{exn} is that it specifies a *closed world* of possible exception sources. All sources must be identified for the entire program, which impedes modular development and evolution. A more modular approach admits an *open world* of exception sources that can be introduced as the program evolves and even as it executes. A generalization of finite sums, called *dynamic classification*, defined in Chapter 33, is required for an open world. (See that Chapter for further discussion.)

When τ_{exn} is a type of classified values, its classes are often called *exceptions*, so that one may speak of "the `fnf` exception" in the above example. This terminology is harmless, and all but unavoidable, but it invites confusion between two separate ideas:

1. Exceptions as a *control mechanism* that allows the course of evaluation to be altered by raising and handling exceptions.
2. Exceptions as a *data value* associated with such a deviation of control that allows the source of the deviation to be identified.

As a control mechanism, exceptions can be eliminated using explicit *exception passing*. A computation of type τ that may raise an exception is interpreted as an exception-free computation of type $\tau + \tau_{\mathsf{exn}}$; see Exercise **29.5** for more on this method.

29.4 Notes

Various forms of exceptions were considered in Lisp (Steele, 1990). The original formulation of ML (Gordon et al., 1979) as a metalanguage for mechanized logic used failures to implement backtracking proof search. Most modern languages now have exceptions, but differ in the forms of data that may be associated with them.

Exercises

29.1. Prove Theorem 29.2. Are any properties of τ_{exn} required for the proof?

29.2. Give an evaluation dynamics for **XPCF** using the following judgment forms:
- Normal evaluation: $e \Downarrow v$, where $e : \tau$, $v : \tau$, and v val.
- Exceptional evaluation: $e \Uparrow v$, where $e : \tau$, and $v : \tau_{\mathsf{exn}}$, and v val.

The first states that e evaluates normally to value v, the second that e raises an exception with value v.

29.3. Give a structural operational dynamics to **XPCF** by inductively defining the following judgment forms:
- $e \longmapsto e'$, stating that expression e transitions to expression e';
- e val, stating that expression e is a value.

https://doi.org/10.1017/CBO9781316576892.031 Published online by Cambridge University Press

Ensure that $e \Downarrow v$ iff $e \longmapsto^* v$, and $e \Uparrow v$ iff $e \longmapsto^* \mathtt{raise}(v)$, where v val in both cases.

29.4. The closed world assumption on exceptions amounts to choosing the type of exception values to be a finite sum type shared by the entire program. Under such an assumption, it is possible to track exceptions by placing an upper bound on the possible classes of an exception value.

Type refinements (defined in Chapter 25) can be used for exception tracking in a closed-world setting. Define *finite sum refinements* by the rule

$$\frac{X' \subseteq X \quad (\forall x \in X')\ \phi_x \sqsubseteq \tau_x}{[\phi_x]_{x \in X'} \sqsubseteq [\tau_x]_{x \in X}}.$$

In particular, the refinement $\emptyset$ is the vacuous sum refinement [] satisfied by no value. Entailment of finite sum refinements is defined by the rule

$$\frac{X' \subseteq X'' \quad (\forall x \in X')\ \phi_x \leq \phi'_x}{[\phi_x]_{x \in X'} \leq [\phi'_x]_{x \in X''}}$$

So, in particular, $\emptyset \leq \phi$ for all sum refinements ϕ of τ_{exn}. Entailment weakens knowledge of the class of a value of sum type, which is crucial to their application to exception tracking.

The goal of this exercise is to develop a system of type refinements for the modal formulation of exceptions in **MPCF** using sum refinements to perform exception tracking.

(a) Define the command refinement judgment $m \in_\tau \phi\ \mathtt{ow}\ \chi$, where $m \div \tau$, $\phi \sqsubseteq \tau$, and $\chi \sqsubseteq \tau_{\mathsf{exn}}$, to mean that if m returns e, then $e \in_\tau \phi$, and if m raises e, then $e \in_{\tau_{\mathsf{exn}}} \chi$.

(b) Define satisfaction and entailment for the expression refinement $\mathtt{cmd}(\phi; \chi) \sqsubseteq \mathtt{cmd}(\tau)$, where $\phi \sqsubseteq \tau$ and $\chi \sqsubseteq \tau_{\mathsf{exn}}$. This refinement classifies encapsulated commands that satisfy the stated value and exception refinements in the sense of the preceding problem.

29.5. Show that exceptions in **MPCF** can be eliminated by a translation into **PCF** enriched with sum types by what is called the *exception-passing style* transformation. Each command $m \div \tau$ of **MPCF** is translated to a pure expression $\widehat{m}$ of type $\widehat{\tau} + \tau_{\mathsf{exn}}$ whose value is either $\mathtt{l} \cdot e$, where $e : \tau$, for normal return, or $\mathtt{r} \cdot e$, where $e : \tau_{\mathsf{exn}}$, for an exceptional return. The command translation is extended to an expression translation $\widehat{e}$ that replaces occurrences of $\mathtt{cmd}(m)$ by $\widehat{m}$. The corresponding type translation, $\widehat{\tau}$, replaces $\mathtt{cmd}(\tau)$ by $\widehat{\tau} + \tau_{\mathsf{exn}}$. Define the command translation from **MPCF** to **PCF** enriched with sums, and show that it has the required type and correctly simulates the behavior of exceptions.

https://doi.org/10.1017/CBO9781316576892.031 Published online by Cambridge University Press