14 *Exceptions*

In Chapter 13 we saw how to extend the simple operational semantics of the pure simply typed lambda-calculus with mutable references and considered the effect of this extension on the typing rules and type safety proofs. In this chapter, we treat another extension to our original computational model: raising and handling exceptions.

Real-world programming is full of situations where a function needs to signal to its caller that it is unable to perform its task for some reason—because some calculation would involve a division by zero or an arithmetic overflow, a lookup key is missing from a dictionary, an array index went out of bounds, a file could not be found or opened, some disastrous event occurred such as the system running out of memory or the user killing the process, etc.

Some of these exceptional conditions can be signaled by making the function return a variant (or option), as we saw in §11.10. But in situations where the exceptional conditions are truly *exceptional,* we may not want to force every caller of our function to deal with the possibility that they may occur. Instead, we may prefer that an exceptional condition causes a direct transfer of control to an *exception handler* defined at some higher-level in the program—or indeed (if the exceptional condition is rare enough or if there is nothing that the caller can do anyway to recover from it) simply aborts the program. We first consider the latter case (§14.1), where an exception is a whole-program abort, then add a mechanism for trapping and recovering from exceptions (§14.2), and finally refine both of these mechanisms to allow extra programmer-specified data to be passed between exception sites and handlers (§14.3).

The systems studied in this chapter are the simply typed lambda-calculus (Figure 9-1) extended with various primitives for exceptions and exception handling (Figures 14-1 and 14-2). The OCaml implementation of the first extension is `fullerror`. The language with exceptions carrying values (Figure 14-3) is not implemented.

→ *error* *Extends* $\lambda_{\to}$ *(9-1)*

New syntactic forms

$t ::= \ldots$ *terms:*

`error` *run-time error*

New evaluation rules $t \longrightarrow t'$

$\texttt{error}\ t_2 \longrightarrow \texttt{error}$ (E-AppErr1)

$v_1\ \texttt{error} \longrightarrow \texttt{error}$ (E-AppErr2)

New typing rules $\Gamma \vdash t : T$

$\Gamma \vdash \texttt{error} : T$ (T-Error)

Figure 14-1: Errors

14.1 Raising Exceptions

Let us start by enriching the simply typed lambda-calculus with the simplest possible mechanism for signaling exceptions: a term `error` that, when evaluated, completely aborts evaluation of the term in which it appears. Figure 14-1 details the needed extensions.

The main design decision in writing the rules for `error` is how to formalize "abnormal termination" in our operational semantics. We adopt the simple expedient of letting `error` itself be the result of a program that aborts. The rules E-AppErr1 and E-AppErr2 capture this behavior. E-AppErr1 says that, if we encounter the term `error` while trying to reduce the left-hand side of an application to a value, we should immediately yield `error` as the result of the application. Similarly, E-AppErr2 says that, if we encounter an `error` while we are working on reducing the argument of an application to a value, we should abandon work on the application and immediately yield `error`.

Observe that we have *not* included `error` in the syntax of values—only the syntax of terms. This guarantees that there will never be an overlap between the left-hand sides of the E-AppAbs and E-AppErr2 rules—i.e., there is no ambiguity as to whether we should evaluate the term

```
(λx:Nat.0) error
```

by performing the application (yielding 0 as result) or aborting: only the latter is possible. Similarly, the fact that we used the metavariable v_1 (rather than t_1, ranging over arbitrary terms) in E-AppErr2 forces the evaluator to wait until the left-hand side of an application is reduced to a value before aborting

it, even if the right-hand side is `error`. Thus, a term like

```
(fix (λx:Nat.x)) error
```

will diverge instead of aborting. These conditions ensure that the evaluation relation remains deterministic.

The typing rule T-ERROR is also interesting. Since we may want to raise an exception in any context, the term `error` form is allowed to have any type whatsoever. In

```
(λx:Bool.x) error;
```

it has type `Bool`. In

```
(λx:Bool.x) (error true);
```

it has type `Bool→Bool`.

This flexibility in `error`'s type raises some difficulties in implementing a typechecking algorithm, since it breaks the property that every typable term in the language has a unique type (Theorem 9.3.3). This can be dealt with in various ways. In a language with subtyping, we can assign `error` the minimal type `Bot` (see §15.4), which can be *promoted* to any other type as necessary. In a language with parametric polymorphism (see Chapter 23), we can give `error` the polymorphic type `∀X.X`, which can be *instantiated* to any other type. Both of these tricks allow infinitely many possible types for `error` to be represented compactly by a single type.

14.1.1 EXERCISE [⋆]: Wouldn't it be simpler just to require the programmer to annotate `error` with its intended type in each context where it is used? □

The type preservation property for the language with exceptions is the same as always: if a term has type `T` and we let it evaluate one step, the result still has type `T`. The progress property, however, needs to be refined a little. In its original form, it said that a well-typed program must evaluate to a value (or diverge). But now we have introduced a non-value normal form, `error`, which can certainly be the result of evaluating a well-typed program. We need to restate progress to allow for this.

14.1.2 THEOREM [PROGRESS]: Suppose `t` is a closed, well-typed normal form. Then either `t` is a value or `t` = `error`. □

14.2 Handling Exceptions

The evaluation rules for `error` can be thought of as "unwinding the call stack," discarding pending function calls until the `error` has propagated all

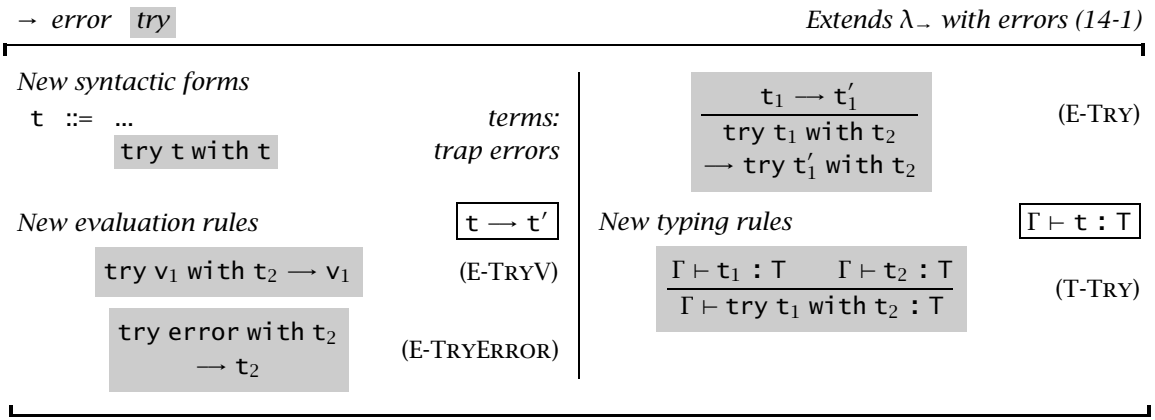

Figure 14-2: Error handling

the way to the top level. In real implementations of languages with exceptions, this is exactly what happens: the call stack consists of a set of *activation records,* one for each active function call; raising an exception causes activation records to be popped off the call stack until it becomes empty.

In most languages with exceptions, it is also possible to install *exception handlers* in the call stack. When an exception is raised, activation records are popped off the call stack until an exception handler is encountered, and evaluation then proceeds with this handler. In other words, the exception functions as a non-local transfer of control, whose target is the most recently installed exception handler (i.e., the nearest one on the call stack).

Our formulation of exception handlers, summarized in Figure 14-2, is similar to both ML and Java. The expression `try` t_1 `with` t_2 means "return the result of evaluating t_1, unless it aborts, in which case evaluate the handler t_2 instead." The evaluation rule E-TryV says that, when t_1 has been reduced to a value v_1, we may throw away the `try`, since we know now that it will not be needed. E-TryError, on the other hand, says that, if evaluating t_1 results in `error`, then we should replace the `try` with t_2 and continue evaluating from there. E-Try tells us that, until t_1 has been reduced to either a value or `error`, we should just keep working on it and leave t_2 alone.

The typing rule for `try` follows directly from its operational semantics. The result of the whole `try` can be either the result of the main body t_1 or else the result of the handler t_2; we simply need to require that these have the same type T, which is also the type of the `try`.

The type safety property and its proof remain essentially unchanged from the previous section.

→ *exceptions* *Extends* $\lambda_{\to}$ *(9-1)*

New syntactic forms

$t ::= \ldots$ *terms:*
$\texttt{raise}\ t$ *raise exception*
$\texttt{try}\ t\ \texttt{with}\ t$ *handle exceptions*

New evaluation rules $\boxed{t \longrightarrow t'}$

$(\texttt{raise}\ v_{11})\ t_2 \longrightarrow \texttt{raise}\ v_{11}$ (E-AppRaise1)

$v_1\ (\texttt{raise}\ v_{21}) \longrightarrow \texttt{raise}\ v_{21}$ (E-AppRaise2)

$$\frac{t_1 \longrightarrow t_1'}{\texttt{raise}\ t_1 \longrightarrow \texttt{raise}\ t_1'}$$ (E-Raise)

$\texttt{raise}\ (\texttt{raise}\ v_{11}) \longrightarrow \texttt{raise}\ v_{11}$ (E-RaiseRaise)

$\texttt{try}\ v_1\ \texttt{with}\ t_2 \longrightarrow v_1$ (E-TryV)

$\texttt{try raise}\ v_{11}\ \texttt{with}\ t_2 \longrightarrow t_2\ v_{11}$ (E-TryRaise)

$$\frac{t_1 \longrightarrow t_1'}{\texttt{try}\ t_1\ \texttt{with}\ t_2 \longrightarrow \texttt{try}\ t_1'\ \texttt{with}\ t_2}$$ (E-Try)

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_{exn}}{\Gamma \vdash \texttt{raise}\ t_1 : T}$$ (T-Exn)

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T_{exn} \to T}{\Gamma \vdash \texttt{try}\ t_1\ \texttt{with}\ t_2 : T}$$ (T-Try)

Figure 14-3: Exceptions carrying values

14.3 Exceptions Carrying Values

The mechanisms introduced in §14.1 and §14.2 allow a function to signal to its caller that "something unusual happened." It is generally useful to send back some extra information about *which* unusual thing has happened, since the action that the handler needs to take—either to recover and try again or to present a comprehensible error message to the user—may depend on this information.

Figure 14-3 shows how our basic exception handling constructs can be enriched so that each exception carries a value. The type of this value is written T_{exn}. For the moment, we leave the precise nature of this type open; below, we discuss several alternatives.

The atomic term `error` is replaced by a term constructor $\texttt{raise}\ t$, where t is the extra information that we want to pass to the exception handler. The syntax of `try` remains the same, but the handler t_2 in $\texttt{try}\ t_1\ \texttt{with}\ t_2$ is now interpreted as a *function* that takes the extra information as an argument.

The evaluation rule E-TryRaise implements this behavior, taking the extra information carried by a `raise` from the body t_1 and passing it to the handler t_2. E-AppRaise1 and E-AppRaise2 propagate exceptions through applications, just like E-AppErr1 and E-AppErr2 in Figure 14-1. Note, however, that these

rules are allowed to propagate only exceptions whose extra information is a value; if we attempt to evaluate a `raise` with extra information that itself requires some evaluation, these rules will block, forcing us to use E-RAISE to evaluate the extra information first. E-RAISERAISE propagates exceptions that may occur *while* we are evaluating the extra information that is to be sent along in some other exception. E-TRYV tells us that we can throw away a `try` once its main body has reduced to a value, just as we did in §14.2. E-TRY directs the evaluator to work on the body of a `try` until it becomes either a value or a `raise`.

The typing rules reflect these changes in behavior. In T-RAISE we demand that the extra information has type T_{exn}; the whole `raise` can then be given any type `T` that may be required by the context. In T-TRY, we check that the handler $\mathtt{t}_2$ is a function that, given the extra information of type T_{exn}, yields a result of the same type as $\mathtt{t}_1$.

Finally, let us consider some alternatives for the type T_{exn}.

1. We can take T_{exn} to be just `Nat`. This corresponds to the `errno` convention used, for example, by Unix operating system functions: each system call returns a numeric "error code," with `0` signaling success and other values reporting various exceptional conditions.

2. We can take T_{exn} to be `String`, which avoids looking up error numbers in tables and allows exception-raising sites to construct more descriptive messages if they wish. The cost of this extra flexibility is that error handlers may now have to *parse* these strings to find out what happened.

3. We can keep the ability to pass more informative exceptions while avoiding string parsing if we define T_{exn} to be a *variant type:*

   ```
   T_exn  =  <divideByZero:     Unit,
              overflow:         Unit,
              fileNotFound:     String,
              fileNotReadable:  String,
              ... >
   ```

   This scheme allows a handler to distinguish between kinds of exceptions using a simple `case` expression. Also, different exceptions can carry different types of additional information: exceptions like `divideByZero` need no extra baggage, `fileNotFound` can carry a string indicating which file was being opened when the error occurred, etc.

   The problem with this alternative is that it is rather inflexible, demanding that we fix *in advance* the complete set of exceptions that can be raised by

any program (i.e., the set of tags of the variant type T_{exn}). This leaves no room for programmers to declare application-specific exceptions.

4. The same idea can be refined to leave room for user-defined exceptions by taking T_{exn} to be an *extensible variant type*. ML adopts this idea, providing a single extensible variant type called `exn`.[1] The ML declaration `exception l of T` can be understood, in the present setting, as "make sure that `l` is different from any tag already present in the variant type T_{exn},[2] and from now on let T_{exn} be $\langle l_1{:}T_1 \ldots l_n{:}t_n, l{:}T\rangle$, where $l_1{:}T_1$ through $l_n{:}t_n$ were the possible variants before this declaration."

   The ML syntax for raising exceptions is `raise l(t)`, where `l` is an exception tag defined in the current scope. This can be understood as a combination of the tagging operator and our simple `raise`:

   $$\texttt{raise l(t)} \stackrel{\text{def}}{=} \texttt{raise (<l=t> as } T_{exn}\texttt{)}$$

   Similarly, the ML `try` construct can be desugared using our simple `try` plus a `case`.

   $$\begin{aligned}\texttt{try t with l(x)} \rightarrow \texttt{h} \stackrel{\text{def}}{=}\ & \texttt{try t with} \\ & \quad \lambda \texttt{e:}T_{exn}\texttt{.\ case e of} \\ & \qquad\qquad \texttt{<l=x>} \Rightarrow \texttt{h} \\ & \qquad\quad \texttt{|\ \_} \Rightarrow \texttt{raise e}\end{aligned}$$

   The `case` checks whether the exception that has been raised is tagged with `l`. If so, it binds the value carried by the exception to the variable `x` and evaluates the handler `h`. If not, it falls through to the `else` clause, which *re-raises* the exception. The exception will keep propagating (and perhaps being caught and re-raised) until it either reaches a handler that wants to deal with it, or else reaches the top level and aborts the whole program.

5. Java uses classes instead of extensible variants to support user-defined exceptions. The language provides a built-in class `Throwable`; an instance of `Throwable` or any of its subclasses can be used in a `throw` (same as our `raise`) or `try...catch` (same as our `try...with`) statement. New exceptions can be declared simply by defining new subclasses of `Throwable`.

   There is actually a close correspondence between this exception-handling mechanism and that of ML. Roughly speaking, an exception object in Java

1. One can go further and provide extensible variant types as a general language feature, but the designers of ML have chosen to simply treat `exn` as a special case.
2. Since the `exception` form is a binder, we can always ensure that `l` is different from the tags already used in T_{exn} by alpha-converting it if necessary.

is represented at run time by a tag indicating its class (which corresponds directly to the extensible variant tag in ML) plus a record of instance variables (corresponding to the extra information labeled by this tag).

Java exceptions go a little further than ML in a couple of respects. One is that there is a natural partial order on exception tags, generated by the *subclass* ordering. A handler for the exception `I` will actually trap all exceptions carrying an object of class `I` or any subclass of `I`. Another is that Java distinguishes between *exceptions* (subclasses of the built-in class `Exception`—a subclass of `Throwable`), which application programs might want to catch and try to recover from, and *errors* (subclasses of `Error`—also a subclass of `Throwable`), which indicate serious conditions that should normally just terminate execution. The key difference between the two lies in the typechecking rules, which demand that methods explicitly declare which exceptions (but *not* which errors) they might raise.

14.3.1 EXERCISE [⋆⋆⋆]: The explanation of extensible variant types in alternative 4 above is rather informal. Show how to make it precise. □

14.3.2 EXERCISE [⋆⋆⋆⋆]: We noted above that Java exceptions (those that are subclasses of `Exception`) are a bit more strictly controlled than exceptions in ML (or the ones we have defined here): every exception that might be raised by a method must be declared in the method's type. Extend your solution to Exercise 14.3.1 so that the type of a function indicates not only its argument and result types, but also the set of exceptions that it may raise. Prove that your system is typesafe. □

14.3.3 EXERCISE [⋆⋆⋆]: Many other control constructs can be formalized using techniques similar to the ones we have seen in this chapter. Readers familiar with the "call with current continuation" (`call/cc`) operator of Scheme (see Clinger, Friedman, and Wand, 1985; Kelsey, Clinger, and Rees, 1998; Dybvig, 1996; Friedman, Wand, and Haynes, 2001) may enjoy trying to formulate typing rules based on a type `Cont T` of *T-continuations*—i.e., continuations that expect an argument of type `T`. □