Motivation

Most programming languages provide some mechanism for interrupting the normal flow of control in a program to signal some exceptional condition.

Note that it is always possible to program without exceptions — instead of raising an exception, we return `None`; instead of returning result `x` normally, we return `∃(x)`. But now we need to wrap every function application in a `case` to find out whether it returned a result or an exception.

→ much more convenient to build this mechanism into the language.

Varieties of non-local control

There are many ways of adding “non-local control flow”

- `exit(i)`
- `goto`
- `setjmp/longjmp`
- `raise/try` (or `catch/throw`) in many variations
- `callcc` / continuations

More esoteric variants (cf. many Scheme papers)
Varieties of non-local control

There are many ways of adding “non-local control flow”

- `exit(1)`
- `goto`
- `setjmp/longjmp`
- `raise/try (or catch/throw)` in many variations
- `callcc / continuations`
- more esoteric variants (cf. many Scheme papers)

Let’s begin with the simplest of these.

An “abort” primitive

First step: raising exceptions (but not catching them).

\[
\text{t ::= ... terms}
\]

\[
\text{error run-time error}
\]

Evaluation

\[
\text{error } \tau \rightarrow \text{error} \quad \text{(E-AppErr1)}
\]
\[
\nu_1 \text{ error } \rightarrow \text{error} \quad \text{(E-AppErr2)}
\]

Typing

\[
\Gamma \vdash \text{error} : T \quad \text{(T-ERROR)}
\]

Aside: Syntax-directedness

Note that this rule

\[
\Gamma \vdash \text{error} : T \quad \text{(T-ERROR)}
\]

has a problem from the point of view of implementation: it is not syntax-directed!

This will cause the Uniqueness of Types theorem to fail.

For purposes of defining the language and proving its type safety, this is not a problem — Uniqueness of Types is not critical.

Let’s think a little, though, about how the rule might be fixed...
An alternative

Can’t we just decorate the error keyword with its intended type, as we have done to fix related problems with other constructs?

\[ \Gamma \vdash \text{(error as T)} : T \quad (T\text{-ERROR}) \]

No, this doesn’t work!

E.g. (assuming our language also has numbers and booleans):

\[ \text{succ (if (error as Bool) then 5 else 7)} \rightarrow \text{succ (error as Bool)} \]

Exercise: Come up with a similar example using just functions and error.

Another alternative

In a system with universal polymorphism (like OCaml), the variability of typing for error can be dealt with by assigning it a variable type!

\[ \Gamma \vdash \text{error : } \alpha \quad (T\text{-ERROR}) \]

In effect, we are replacing the uniqueness of typing property by a weaker (but still useful) property called most general typing.

I.e., although a term may have many types, we always have a compact way of representing the set of all of its possible types.

Yet another alternative

Alternatively, in a system with subtyping (which we’ll discuss in the next lecture) and a minimal Bot type, we can give error a unique type:

\[ \Gamma \vdash \text{error : Bot} \quad (T\text{-ERROR}) \]

(Of course, what we’ve really done is just pushed the complexity of the old error rule onto the Bot type! We’ll return to this point later.)
For now...

Let’s stick with the original rule

\[ \Gamma \vdash \text{error} : T \]  \hspace{1cm} \text{(T-ERROR)}

and live with the resulting nondeterminism of the typing relation.

Type safety

The preservation theorem requires no changes when we add \text{error}: if a term of type \( T \) reduces to \text{error}, that’s fine, since \text{error} has every type \( T \).

Progress

First, note that we do not want to extend the set of values to include \text{error}, since this would make our new rule for propagating errors through applications.

\[ v_1 \; \text{error} \rightarrow \text{error} \]  \hspace{1cm} \text{(E-AppErr2)}

\[ (\lambda x : T_1 . t_{12}) \; v_2 \rightarrow [x \mapsto v_2]t_{12} \]  \hspace{1cm} \text{(E-AppAbs)}

e.g., the term

\[ (\lambda x : \text{Nat}.0) \; \text{error} \]

could evaluate to either 0 (which would be wrong) or \text{error} (which is what we intend).
Progress

Instead, we keep error as a non-value normal form, and refine the statement of progress to explicitly mention the possibility that terms may evaluate to error instead of to a value.

**Theorem [Progress]:** Suppose t is a closed, well-typed normal form. Then either t is a value or t \(\sim\) error.

Catching exceptions

t ::= ... terms

\[\text{try } t \text{ with } t\]

**Evaluation**

\[\text{try } v_1 \text{ with } t_2 \rightarrow v_1\] \(\text{(E-TRYV)}\)

\[\text{try error with } t_2 \rightarrow t_2\] \(\text{(E-TRYERROR)}\)

\[t_1 \rightarrow t'_1\]

\[\text{try } t_1 \text{ with } t_2 \rightarrow \text{try } t'_1 \text{ with } t_2\] \(\text{(E-TRY)}\)

**Typing**

\[\Gamma \vdash t_1 : T\]

\[\Gamma \vdash t_2 : T\]

\[\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T\] \(\text{(T-TRY)}\)

Exceptions carrying values

t ::= ... terms

\[\text{raise } t\]

**Evaluation**

\[\text{(raise } v_1) \rightarrow \text{raise } v_1\] \(\text{(E-APPRAISE1)}\)

\[v_1 \rightarrow \text{raise } v_2\] \(\text{(E-APPRAISE2)}\)

\[t_1 \rightarrow t'_1\]

\[\text{raise } t_1 \rightarrow \text{raise } t'_1\] \(\text{(E-RAISE)}\)

\[\text{raise } (\text{raise } v_1) \rightarrow \text{raise } v_1\] \(\text{(E-RAISERAISE)}\)
Typing

\[ \Gamma \vdash t_1 : T_{exn} \]
\[ \Gamma \vdash \text{raise } t_1 : T \]
\[ (T\text{-EXN}) \]

\[ \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_{exn} \rightarrow T \]
\[ \Gamma \vdash \text{try } t_1 \text{ with } t_2 : T \]
\[ (T\text{-TRY}) \]