Exceptions (Chapter 14)

Motivation

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

Examples:

- `exit(1)`
- `goto` which is now usually discouraged
- `raise` (or `catch/throw`) in many variations
- `callcc` in continuations
- More esoteric variants (cf. many Scheme papers)

Note that it is always possible to program without exceptions — instead of raising an exception, we return `#e` instead of returning `result` normally, we return `#e` instead of returning a function application in a `case` to find out whether it returned a result or an exception.

Varieties of non-local control

There are many ways of adding "non-local control flow"
Varieties of non-local control

There are many ways of addressing "non-local control flow" in languages. Let's begin with the simplest of these.

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

```c
if x > 0 then true else error

and

if x < 0 then 5 else error
```

This means that both

```c
L : error : f
```

Note that the typing rule for `error` allows us to give it any type.

L : error : f

Typing errors

```
Typing

Typing
```

What if we had booleans and numbers in the language?

```
<table>
<thead>
<tr>
<th>Error</th>
<th>error</th>
</tr>
</thead>
</table>
```

There are many ways of adding "non-local control flow"
Aside: Syntax-directedness

Note that this rule

\[ T \rightarrow \text{error as } T \]

has a problem from the point of view of implementation: It is not

\[ \text{error as } T \]

done to fix related problems with other constructs.

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

No, this doesn't work!

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

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

\[ \text{succ}(\text{error as Bool}) \]

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

\[ T \rightarrow \text{error as } T \]

\[ L \rightarrow \text{error} : T \]

\[ L \rightarrow \text{error} : T \]

Lett's think a little harder about how the type might be fixed...

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

\[ \text{succ}(\text{error as Bool}) \]

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

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

Progress, though, requires a little more care.

Progress

Theorem [Progress]: Suppose t is a closed, well-typed normal form. Then either t is a value or t = error.

Proof: Suppose t is a normal form, and define the statement error overlaps with our existing computation rule for applications:

\[ \text{error} \rightarrow \text{error} \]

\[ \text{error} \rightarrow \text{error} \]

since this would make our new rule for propagating errors through applications overlap with our existing one. First, note that we do not want to extend the set of values to include error.

Progress

Type safety

The preservation theorem requires no changes when we add error: if a term of type T reduces to error, that’s fine, since error has every type T.
Exceptions carrying values

```
t ::= ... raise t
```