CIS 500

Software Foundations

Fall 2003

29 October

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.

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 $\exists$`(x)`. But now we need to wrap every function application in a `case` to find out whether it returned a result or an exception.

$\longrightarrow$ 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(1)`
- `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).

t	::=	...	terms
		error	run-time error

Evaluation

$$\texttt{error}\ \texttt{t}_2 \longrightarrow \texttt{error} \qquad \text{(E-AppErr1)}$$

$$\texttt{v}_1\ \texttt{error} \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

Typing

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

Typing errors

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

$$\Gamma \vdash \texttt{error} : \texttt{T} \qquad \text{(T-ERROR)}$$

This means that both

```
if x>0 then 5 else error
```

and

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

will typecheck.

Aside: Syntax-directedness

Note that this rule

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

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?

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

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?

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

No, this doesn't work!

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

```
    succ (if (error as Bool) then 5 else 7)
⟶  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!

$$\Gamma \vdash \texttt{error} : \texttt{'a} \qquad \text{(T-Error)}$$

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:

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

(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

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

and live with the resulting *nondeterminism* of the typing relation.

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`.

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`.

Progress, though, requires a litte more care.

Progress

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

$$\text{v}_1\ \texttt{error} \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

overlap with our existing computation rule for applications:

$$(\lambda \text{x:T}_{11}.\text{t}_{12})\ \text{v}_2 \longrightarrow [\text{x} \mapsto \text{v}_2]\text{t}_{12} \qquad \text{(E-AppAbs)}$$

e.g., the term

```
(λx:Nat.0) error
```

could evaluate to either `0` (which would be wrong) or `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` = `error`.

Catching exceptions

$$\texttt{t} \quad ::= \quad \ldots \qquad \text{terms}$$

$$\texttt{try t with t} \qquad \text{trap errors}$$

Evaluation

$$\texttt{try } \texttt{v}_1 \texttt{ with } \texttt{t}_2 \longrightarrow \texttt{v}_1 \qquad \text{(E-TryV)}$$

$$\texttt{try error with } \texttt{t}_2 \longrightarrow \texttt{t}_2 \qquad \text{(E-TryError)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{try } \texttt{t}_1 \texttt{ with } \texttt{t}_2 \longrightarrow \texttt{try } \texttt{t}_1' \texttt{ with } \texttt{t}_2} \qquad \text{(E-Try)}$$

Typing

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}}{\Gamma \vdash \texttt{try } \texttt{t}_1 \texttt{ with } \texttt{t}_2 : \texttt{T}} \qquad \text{(T-Try)}$$

Exceptions carrying values

$\mathtt{t}$	::=	...	terms
		$\mathtt{raise\ t}$	raise exception

Evaluation

$$(\mathtt{raise}\ \mathtt{v}_{11})\ \mathtt{t}_2 \longrightarrow \mathtt{raise}\ \mathtt{v}_{11} \qquad \text{(E-AppRaise1)}$$

$$\mathtt{v}_1\ (\mathtt{raise}\ \mathtt{v}_{21}) \longrightarrow \mathtt{raise}\ \mathtt{v}_{21} \qquad \text{(E-AppRaise2)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{raise}\ \mathtt{t}_1 \longrightarrow \mathtt{raise}\ \mathtt{t}_1'} \qquad \text{(E-Raise)}$$

$$\mathtt{raise}\ (\mathtt{raise}\ \mathtt{v}_{11}) \longrightarrow \mathtt{raise}\ \mathtt{v}_{11} \qquad \text{(E-RaiseRaise)}$$

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

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

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

Typing

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_{exn}}{\Gamma \vdash \texttt{raise } \texttt{t}_1 : \texttt{T}} \qquad \text{(T-Exn)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}_{exn} \to \texttt{T}}{\Gamma \vdash \texttt{try } \texttt{t}_1 \texttt{ with } \texttt{t}_2 : \texttt{T}} \qquad \text{(T-Try)}$$