



Recapitulation

Reference

Syntax



We added to λ_{\rightarrow} (with **Unit**) syntactic forms for *creating*, *dereferencing*, and *assigning* reference cells, plus a new type constructor **Ref**.

$t ::=$

`unit`

`x`

`$\lambda x:T.t$`

`t t`

`ref t`

`!t`

`t := t`

`/`

terms

unit constant

variable

abstraction

application

reference creation

dereference

assignment

store location

Evaluation



Evaluation becomes *a* relation with the states of store:

$$t \mid \boxed{\mu} \rightarrow t' \mid \boxed{\mu'}$$

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \quad (\text{E-REFV})$$

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \quad (\text{E-DEREFLOC})$$

$$l := v_2 \mid \mu \longrightarrow \text{unit} \mid [l \mapsto v_2]\mu \quad (\text{E-ASSIGN})$$

Typing



Typing becomes a *four-place* relation: $\Gamma \boxed{\Sigma} \vdash t : T$

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \text{Ref } T_1} \quad (\text{T-LOC})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1} \quad (\text{T-REF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \quad (\text{T-DEREF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

Preservation



Theorem: if

$$\Gamma \mid \Sigma \vdash t : T$$

$$\Gamma \mid \Sigma \vdash \mu$$

$$t \mid \mu \longrightarrow t' \mid \mu'$$

then, for **some** $\Sigma' \supseteq \Sigma$,

$$\Gamma \mid \Sigma' \vdash t' : T$$

$$\Gamma \mid \Sigma' \vdash \mu'.$$

Progress



Theorem: Suppose t is a *closed, well-typed* term (that is, $\emptyset \mid \Sigma \vdash t : T$ for some T and Σ). Then either t is a value or else, for any store μ such that $\emptyset \mid \Sigma \vdash \mu$, there is some term t' and store μ' with $t \mid \mu \longrightarrow t' \mid \mu'$.



Chapter 14: Exceptions

Why exceptions

Raising exceptions (aborting whole program)

Handling exceptions

Exceptions carrying values



Exceptions



Why 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 :

- Division by zero
- Arithmetic overflow
- Array index out of bound
- Lookup key missing
- File could not be opened
-

Most programming languages *provide some mechanism* for *interrupting the normal flow of control* in a program to *signal some exceptional condition* (& the transfer of control flow)



Why exceptions?

Note that it is always possible to program *without exceptions* :

- instead of raising an exception, return **None**
- instead of returning result x normally, return **Some(x)**

type ' α list = None | Some of ' α

let head l = match l with

 [] -> None
 | x::_ -> Some (x);;



Why exceptions?

```
# type 'α list = None | Some of 'α
```

```
# let head l = match l with
```

```
    []      -> None
  | x::_    -> Some (x);;
```

What is the result of type inference?

```
val head: 'α list -> 'α Option = <fun>
```

```
val head: 'α list -> 'α = <fun>
```

```
# let head l = match l with
```

```
    []      -> raise Not_found
  | x::_    -> x;;
```



Why exceptions?

If we want to wrap every function application in a **case** to find out *whether it returned a result or an exception*?

It is much more convenient to *build this mechanism into the language*, and *provide mechanism* for *interrupting the normal flow of control* in a program to *signal some exceptional condition* (& the transfer of control flow).



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
- `calcc` / continuations
- more esoteric variants (cf. many Scheme papers)

that allow programs to effect *non-local “jumps”* in the flow of control

Let’s begin with the simplest of these.

Raising exceptions

(aborting **whole** program)



An “abort” primitive in λ_{\rightarrow}

Raising exceptions (but not catching them), which cause the *abort of the whole program*

Syntactic forms

$t ::= \dots$
 error

terms

run-time error

Evaluation

$\text{error } t_2 \longrightarrow \text{error}$ (E-APPERR1)

$v_1 \text{ error} \longrightarrow \text{error}$ (E-APPERR2)

Typing



Typing

$\Gamma \vdash \text{error} : T$

(T-ERROR)

New syntactic forms

$t ::= \dots$

`error`

terms:

run-time error

New evaluation rules

$t \rightarrow t'$

`error` $t_2 \rightarrow \text{error}$

(E-APPERR1)

v_1 `error` $\rightarrow \text{error}$

(E-APPERR2)

New typing rules

$\Gamma \vdash t : T$

$\Gamma \vdash \text{error} : T$

(T-ERROR)



Typing errors

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

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

What if we had booleans and numbers in the language?

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: this rule

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

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



Aside: Syntax-directed rules

When we say a set of rules is *syntax-directed* we mean two things:

1. There is *exactly one rule* in the set that applies to each syntactic form (in the sense that we can tell *by the syntax of a term* which rule to use)
 - e.g., to derive a type for $t_1 t_2$, we must use **T-App**
2. We *don't* have to “*guess*” an input (or output) for any rule
 - e.g., to derive a type for $t_1 t_2$, we need to *derive a type for t_1* and *a type for t_2*



Aside: Syntax-directedness

Note: this rule

$\Gamma \vdash \text{error} : T$ (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 about how the rule might be fixed ...



An alternative: Ascription

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

$$\Gamma \vdash (\text{error} \boxed{\text{as } T}) : T \quad (\text{T-ERROR})$$



An alternative : Ascription

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

$$\Gamma \vdash (\text{error} \boxed{\text{as } T}) : T \quad (\text{T-ERROR})$$

Unfortunately, this doesn't work!

e.g. assuming our language also has *numbers* and *booleans*:

$\text{succ (if (error as Bool) then 3 else 8)}$
 $\rightarrow \text{succ (error as Bool)}$

Another alternative: Variable type



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

$\Gamma \vdash \text{error} : ' \alpha$ (T-ERROR)

Another alternative: Variable type



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

$$\Gamma \vdash \text{error} : ' \alpha \quad (\text{T-ERROR})$$

In effect, we are replacing the *uniqueness of typing* property by a *weaker* (but still very 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 : *minimal* type



Alternatively, in a system with subtyping (which will be discussed in chapter 15) and a *minimal* **Bot** type, we *can* give **error** a unique type:

Yet another alternative : *minimal* type



Alternatively, in a system with subtyping (which will be discussed in chapter 15) and a *minimal* **Bot** type, we *can* give **error** a unique type:

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

Note :

What we've really done is *just pushed the complexity* of the old error rule *onto the Bot type*!

For now...



Let's stick with the original rule

$$\Gamma \vdash \text{error} : T$$

(T-ERROR)

and live with the resulting *non-determinism* of the typing relation

Type safety



Property of preservation?

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



Property of preservation?

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**.

Whereas,

Progress *requires a little 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

$$v_1 \text{ error} \longrightarrow \text{error} \quad (\text{E-APPERR2})$$

overlap with our *existing computation rule* for applications:

$$(\lambda x:T_{11}.t_{12}) \ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$$

e.g, the term

$(\lambda x:\text{Nat}.0) \text{ error}$

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 $t = \text{error}$.



Handling exceptions



Catching exceptions

$t ::= \dots$ *terms*
 $\text{try } t \text{ with } t$ *trap errors*

Evaluation

$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1$ (E-TRYV)

$\text{try error with } t_2 \longrightarrow t_2$ (E-TRYERROR)

$$\frac{t_1 \longrightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2}$$
 (E-TRY)

Typing

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T}$$
 (T-TRY)

Exceptions carrying values



Exceptions carrying values

When something unusual happened, it's useful to *send back some extra information* about *which unusual thing has happened* so that the handler can *take some actions* depending on this information.

$t ::= \dots$
 $\text{raise } t$

terms
raise exception



Exceptions carrying values

When something unusual happened, it's useful to *send back some extra information* about *which unusual thing has happened* so that the handler can *take some actions* depending on this information.

$t ::= \dots$

$\text{raise } t$

terms

raise exception

Atomic term **error** is replaced by a *term constructor*

$\text{raise } t$

where t is the *extra information* that we want to *pass to the exception handler*

Evaluation



$$(\text{raise } v_{11}) \ t_2 \longrightarrow \text{raise } v_{11} \quad (\text{E-APPRAISE1})$$

$$v_1 \ (\text{raise } v_{21}) \longrightarrow \text{raise } v_{21} \quad (\text{E-APPRAISE2})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{raise } t_1 \longrightarrow \text{raise } t'_1} \quad (\text{E-RAISE})$$

$$\boxed{\text{raise } (\text{raise } v_{11}) \longrightarrow \text{raise } v_{11} \quad (\text{E-RAISERAISE})}$$

$$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1 \quad (\text{E-TRYV})$$

$$\text{try raise } v_{11} \text{ with } t_2 \longrightarrow t_2 \ v_{11} \quad (\text{E-TRYRAISE})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2} \quad (\text{E-TRY})$$

Evaluation



$$(\text{raise } v_{11}) \ t_2 \longrightarrow \text{raise } v_{11} \quad (\text{E-APPRaise1})$$

$$v_1 \ (\text{raise } v_{21}) \longrightarrow \text{raise } v_{21} \quad (\text{E-APPRaise2})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{raise } t_1 \longrightarrow \text{raise } t'_1} \quad (\text{E-RAISE})$$

$$\text{raise } (\text{raise } v_{11}) \longrightarrow \text{raise } v_{11} \quad (\text{E-RAISERaise})$$

$$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1 \quad (\text{E-TRYV})$$

$$\text{try raise } v_{11} \text{ with } t_2 \longrightarrow t_2 \ v_{11} \quad (\text{E-TRYRAISE})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2} \quad (\text{E-TRY})$$

Typing



To typecheck **raise** expressions, we need to *choose a type* for *the values* that are *carried along* with exceptions, let's call it T_{exn}

$$\frac{\Gamma \vdash t_1 : T_{\text{exn}}}{\Gamma \vdash \text{raise } t_1 : T} \quad (\text{T-RAISE})$$

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



What is T_{exn} ?

Further, we need to decide *what type* to use as T_{exn}

There are *several possibilities*.

1. Numeric error codes: $T_{\text{exn}} = \text{Nat}$ (as in Unix)
2. Error messages: $T_{\text{exn}} = \text{String}$
3. A *predefined* variant type:

```
 $T_{\text{exn}}$  = <divideByZero:    Unit,  
              overflow:    Unit,  
              fileNotFound: String,  
              fileNotReadable: String,  
              ... >
```

4. An *extensible* variant type (as in Ocaml)
5. A *class* of “*throwable objects*” (as in Java)

Recapitulation: Error handling

→ error try

Extends λ_{\rightarrow} with errors (14-1)

New syntactic forms

$t ::= \dots$

try t with t

terms:

trap errors

New evaluation rules

try v_1 with $t_2 \rightarrow v_1$

try error with t_2
 $\rightarrow t_2$

$t \rightarrow t'$

(E-TRYV)

(E-TRYERROR)

$$\frac{t_1 \rightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \rightarrow \text{try } t'_1 \text{ with } t_2}$$

(E-TRY)

New typing rules

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T}$$

(T-TRY)



Recapitulation: Exceptions carrying values

→ *exceptions*

Extends λ_{\perp} (9-1)

New syntactic forms

$t ::= \dots$

$\text{raise } t$

$\text{try } t \text{ with } t$

terms:

raise exception
handle exceptions

New evaluation rules

$(\text{raise } v_{11}) t_2 \rightarrow \text{raise } v_{11}$ (E-APPRAISE1)

$v_1 (\text{raise } v_{21}) \rightarrow \text{raise } v_{21}$ (E-APPRAISE2)

$$\frac{t_1 \rightarrow t'_1}{\text{raise } t_1 \rightarrow \text{raise } t'_1}$$
 (E-RAISE)

$$\text{raise } (\text{raise } v_{11}) \rightarrow \text{raise } v_{11}$$
 (E-RAISERAISE)

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

$$\text{try raise } v_{11} \text{ with } t_2 \rightarrow t_2 v_{11}$$
 (E-TRYRAISE)

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

New typing rules

$\Gamma \vdash t : T$

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

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

Recapitulation



Raising exception is *more than an error mechanism*: it's a *programmable control structure*

- Sometimes a way to quickly *escape from the computation*.
- And allow programs to effect *non-local “jumps”* in the flow of control by setting a *handler* during evaluation of an expression that may be invoked by raising an exception.
- Exceptions are *value-carrying* in the sense that one may pass a value to *the exception handler* when the exception is raised.
- Exception values have a single type, T_{exn} , which is *shared by all exception handler*.

Recapitulation



E.g., Exceptions are used in **OCaml** as a *control mechanism*, **either** to signal errors, **or** to control the flow of execution.

- When an exception is raised, the current execution is aborted, and control is thrown to the most recently entered active exception handler, which may choose to handle the exception, or pass it through to the next exception handler.
- T_{exn} is defined to be an extensible data type, in the sense that new constructors may be introduced using exception declaration, with no restriction on the types of value that may be associated with the constructor.

HW for chap14



- 14.3.1