To Discuss

项目组队情况

作业

课程的建议

Recap on References

Syntax

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

t ::=		*terms*
	unit	*unit constant*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	ref t	*reference creation*
	!t	*dereference*
	t:=t	*assignment*
	l	store location

Evaluation

Evaluation becomes *a* *four-place* relation: $t \mid \mu \longrightarrow t' \mid \mu'$

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ \ \texttt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \texttt{v}_1)} \qquad \text{(E-RefV)}$$

$$\frac{\mu(l) = \texttt{v}}{!l \mid \mu \longrightarrow \texttt{v} \mid \mu} \qquad \text{(E-DerefLoc)}$$

$$l := \texttt{v}_2 \mid \mu \longrightarrow \texttt{unit} \mid [l \mapsto \texttt{v}_2]\mu \qquad \text{(E-Assign)}$$

Typing

Typing becomes *a three-place relation*: $\Gamma \mid \Sigma \vdash \texttt{t} : \texttt{T}$

$$\frac{\Sigma(l) = \texttt{T}_1}{\Gamma \mid \Sigma \vdash l : \texttt{Ref T}_1} \qquad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash \texttt{t}_1 : \texttt{T}_1}{\Gamma \mid \Sigma \vdash \texttt{ref t}_1 : \texttt{Ref T}_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \texttt{t}_1 : \texttt{Ref T}_{11}}{\Gamma \mid \Sigma \vdash \texttt{!t}_1 : \texttt{T}_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \texttt{t}_1 : \texttt{Ref T}_{11} \qquad \Gamma \mid \Sigma \vdash \texttt{t}_2 : \texttt{T}_{11}}{\Gamma \mid \Sigma \vdash \texttt{t}_1\texttt{:=t}_2 : \texttt{Unit}} \qquad \text{(T-Assign)}$$

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, $\emptyset \mid \Sigma \vdash \mathrm{t}: \mathrm{T}$ 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 $\mathrm{t} \mid \mu \longrightarrow \mathrm{t}' \mid \mu'$.

Nontermination via references

There are well-typed terms in this system that are not strongly normalizing. For example:

$$
\begin{aligned}
t1 &= \lambda r: Ref\ (Unit \to Unit). \\
&\quad (r := (\lambda x: Unit.\ (!\,r)x); \\
&\quad (!\,r)\ unit); \\
t2 &= ref\ (\lambda x: Unit.\ x);
\end{aligned}
$$

Applying t1 to t2 yields a (well-typed) divergent term.

Nontermination via references

There are well-typed terms in this system that are not strongly normalizing. For example:

$$
\begin{aligned}
t1 = \lambda r:&\ \mathrm{Ref}\ (\mathrm{Unit} \to \mathrm{Unit}).\\
&(r := (\lambda x{:}\ \mathrm{Unit}.\ (!\,r)x);\\
&(!\,r)\ \mathrm{unit});\\
t2 = \mathrm{ref}&\ (\lambda x{:}\ \mathrm{Unit}.\ x);
\end{aligned}
$$

Applying t1 to t2 yields a (well-typed) divergent term.

Recursion via references

Indeed, we can define arbitrary recursive functions using references.

1. Allocate a ref cell and initialize it with a dummy function of the appropriate type:

   $$\text{fact}_{ref} \;=\; \text{ref}\;(\lambda \text{n}:\text{Nat}.\,0)$$

2. Define the body of the function we are interested in, using the contents of the reference cell for making recursive calls:

   $$\begin{array}{l}\text{fact}_{body} = \\ \quad \lambda \text{n}:\text{Nat}. \\ \qquad \text{if iszero n then 1 else times n}\;((!\,\text{fact}_{ref})(\text{pred n}))\end{array}$$

3. "Backpatch" by storing the real body into the reference cell:

   $$\text{fact}_{ref} := \text{fact}_{body}$$

4. Extract the contents of the reference cell and use it as desired:

   $$\begin{array}{l}\text{fact} \;=\; !\,\text{fact}_{ref} \\ \text{fact}\;5\end{array}$$

Chapter 14: Exceptions

Why exceptions

Raising exceptions

Handling exceptions

Exceptions carrying values

Exceptions

Why exceptions?

Real world programming is full of situations where a function needs to signal to it 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
- ……

Why exceptions?

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 Some(x). But now we need 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.

Why exceptions?

```
# 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);;
```

Type inference?

Why exceptions?

```
# type 'α list = None | Some of 'α
# let head l = match l with
                 [] -> None
               | x::_ -> Some (x);;
```

Type inference?

```
# let head l = match l with
                 [] -> raise Not_found
               | x::_ -> x;;
```

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 in $\lambda_{\rightarrow}$

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

Syntactic forms

$$\begin{array}{lll} \texttt{t} ::= & \ldots & \textit{terms} \\ & \texttt{error} & \textit{run-time error} \end{array}$$

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

Typing

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

New syntactic forms

$\texttt{t} ::= \ldots$ — *terms:*

$\texttt{error}$ — *run-time error*

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

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

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

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

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

Typing errors

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

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

What if we had booleans and numbers in the language?

Typing errors

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

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

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 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 about how the rule might be fixed...

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. (We can tell by the syntax of a term which rule to use.)
   - In order 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.
   - To derive a type for t_1 t_2, we need to derive a type for t_1 and a type for t_2.

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 3 else 8)

$\longrightarrow$ succ (error as Bool)

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!

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

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!

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

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

Alternatively, in a system with subtyping (which we'll discuss next chapter) and a minimal Bot type, we *can* give error a unique type:

Yet another alternative

Alternatively, in a system with subtyping (which we'll discuss next chapter) 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!)

For now…

Let's stick with the original rule

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

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 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.

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

overlap with our existing computation rule for applications:

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

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* $t = error$.

Catching exceptions

$$\texttt{t} \ ::= \ \ldots \qquad\qquad \textit{terms}$$
$$\texttt{try t with t} \qquad\qquad \textit{trap errors}$$

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

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.

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	::=	...	*terms*
		raise t	*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	::=	...	*terms*
		raise t	*raise exception*

Atomic term error is replaced by *a term constructor*

raise t

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

Evaluation

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

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

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

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

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

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

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

Evaluation

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

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

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

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

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

$$\texttt{try raise v}_{11}\ \texttt{with t}_2 \longrightarrow \texttt{t}_2\ \texttt{v}_{11} \qquad \text{(E-TryRaise)}$$

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

Typing

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

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

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

What is T_{exn} ?

To complete the story, we need to decide what type to use as T_{exn}. There are several possibilities.

What is T_{exn} ?

To complete the story, we need to decide what type to use as T_{exn}. There are several possibilities.

1. Numeric error codes: T_{exn} = Nat (as in C)

What is T_{exn} ?

To complete the story, we need to decide what type to use as T_{exn}. There are several possibilities.

1. Numeric error codes: T_{exn} = Nat (as in C)
2. Error messages: T_{exn} = String

What is T_{exn} ?

To complete the story, we need to decide what type to use as T_{exn}. There are several possibilities.

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

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

What is T_{exn} ?

To complete the story, we need to decide what type to use as T_{exn}. There are several possibilities.

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

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

4. An *extensible* variant type (as in Ocaml)

What is T_{exn} ?

To complete the story, we need to decide what type to use as T_{exn}. There are several possibilities.

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

$$\mathrm{T}_{exn} = \texttt{<divideByZero: Unit, overflow: Unit, fileNotFound: String, fileNotReadable: String, ...>}$$

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

Recapitulation: Error handling

$\rightarrow$ *error* *try* — *Extends $\lambda_{\rightarrow}$ with errors (14-1)*

New syntactic forms

$$\begin{array}{lll} \texttt{t} ::= & \ldots & \textit{terms:} \\ & \texttt{try t with t} & \textit{trap errors} \end{array}$$

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

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

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

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

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

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

Recapitulation: Exceptions carrying values

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

New syntactic forms

t ::=	...	*terms:*
	raise t	*raise exception*
	try t with t	*handle exceptions*

New evaluation rules $t \longrightarrow t'$

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

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

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

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

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

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

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

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

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

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

Recapitulation

- Raising exception is more than an error mechanism: it's a programmable control structure
  - Sometimes a way to quickly escape from the computation

- 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.

Examples in OCaml

```
# let rec assoc key = function
    (k, v) :: l ->
        if k = key then v
        else assoc key l
    | [] -> raise Not_found;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>

# assoc 2 l;;
-   : string = "World“
# assoc 3 l;;
-   Exception: Not_found.
# "Hello" ^ assoc 2 l;;
- : string = "HelloWorld"
```

Examples in OCaml

```
let find_index p =
    let rec find n =
        function [] -> raise (Failure "not found")
            | x::L -> if p(x) then raise (Found n)
                else find (n+1) L
        in
        try find 1 L  with Found n ->   n;;
```