# Formal Methods

## Lecture 9

(B. Pierce's slides for the book "Types and Programming Languages")

**This Saturday, 17 November 2018, room 335 (FSEGA), we will recover the following activities:**

- **1 Programming Paradigms course, 7.30-9.30**
- **1 Formal Methods seminar, 9.30-11.30**

# References

# Mutability

- In most programming languages, *variables* are mutable — i.e., a variable provides both
  - a name that refers to a previously calculated value, and
  - the possibility of *overwriting* this value with another (which will be referred to by the same name)
- In some languages (e.g., OCaml), these features are separate:
  - variables are only for naming — the binding between a variable and its value is immutable
  - introduce a new class of *mutable values* (called *reference cells* or *references*)
  - at any given moment, a reference holds a value (and can be *dereferenced* to obtain this value)
  - a new value may be *assigned* to a reference

We choose OCaml's style, which is easier to work with formally.

So a variable of type T in most languages (except OCaml) will correspond to a Ref T (actually, a Ref(Option T)) here.

# Basic Examples

```
r  = ref 5

!r

r  := 7

(r:=succ(!r); !r)

(r:=succ(!r); r:=succ(!r); r:=succ(!r);
 r:=succ(!r); !r)
```

# Basic Examples

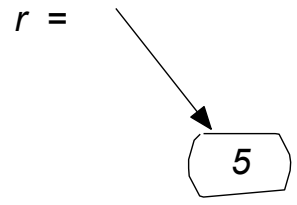
```
r  = ref 5

!r

r  := 7

(r:=succ(!r); !r)

(r:=succ(!r); r:=succ(!r); r:=succ(!r);
 r:=succ(!r); !r)
```

i.e.,

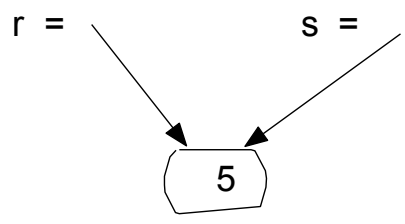
```
((((r:=succ(!r); r:=succ(!r)); r:=succ(!r));
  r:=succ(!r)); !r)
```

# Aliasing

A value of type Ref T is a *pointer* to a cell holding a value of type T.




If this value is "copied" by assigning it to another variable, the cell pointed to is not copied.




So we can change r by assigning to s:

```
(s:=6; !r)
```

# Aliasing all around us

Reference cells are not the only language feature that introduces the possibility of aliasing.

- arrays
- communication channels
- I/O devices (disks, etc.)

# The difficulties of aliasing

The possibility of aliasing invalidates all sorts of useful forms of reasoning about programs, both by programmers...

> *The function*
>
> $$\lambda r{:}Ref\ Nat.\ \lambda s{:}Ref\ Nat.\ (r:=2;\ s:=3;\ !r)$$
>
> *always returns* 2 unless *r and s are aliases*.

...and by compilers:

> Code motion out of loops, common subexpression elimination, allocation of variables to registers, and detection of uninitialized variables all depend upon the compiler knowing which objects a load or a store operation could reference.

High-performance compilers spend significant energy on *alias analysis* to try to establish when different variables cannot possibly refer to the same storage.

# The benefits of aliasing

The problems of aliasing have led some language designers simply to disallow it (e.g., Haskell).
But there are good reasons why most languages do provide constructs involving aliasing:

- efficiency (e.g., arrays)
- "action at a distance" (e.g., symbol tables)
- shared resources (e.g., locks) in concurrent systems
- etc.

# Example

```
c = ref 0
incc = λx:Unit. (c := succ (!c); !c)
decc = λx:Unit. (c := pred (!c); !c)
incc unit
decc unit
o = {i = incc, d = decc}
```

```
let newcounter =
  λ_:Unit.
    let c = ref 0 in
    let incc = λx:Unit. (c := succ (!c); !c) in
    let decc = λx:Unit. (c := pred (!c); !c) in
    let o = {i = incc, d = decc} in
    o
```

# Syntax

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

... plus other familiar types, in examples.

# Typing Rules

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \mathrm{ref}\ t_1 : \mathrm{Ref}\ T_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{Ref}\ T_1}{\Gamma \vdash !t_1 : T_1} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{Ref}\ T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \mathrm{Unit}} \qquad \text{(T-Assign)}$$

# Final example

```
NatArray = Ref (Nat→Nat);

newarray = λ_:Unit. ref (λn:Nat.0);
         : Unit → NatArray

lookup = λa:NatArray. λn:Nat. (!a) n;
       : NatArray → Nat → Nat

update = λa:NatArray. λm:Nat. λv:Nat.
           let oldf = !a in
           a := (λn:Nat. if equal m n then v else oldf n);
       : NatArray → Nat → Nat → Unit
```

# Evaluation

What is the *value* of the expression `ref 0`?

# Evaluation

What is the *value* of the expression `ref 0`?
Crucial observation: evaluating `ref 0` must *do* something.
Otherwise,

```
r = ref 0

s = ref 0
```

and

```
r = ref 0

s = r
```

would behave the same.

# Evaluation

What is the *value* of the expression `ref 0`?
Crucial observation: evaluating `ref 0` must *do* something.
Otherwise,

```
r = ref 0

s = ref 0
```

and

```
r = ref 0

s = r
```

would behave the same.

Specifically, evaluating `ref 0` should *allocate some storage* and yield a *reference* (or *pointer*) to that storage.
So what is a reference?

# The Store

A reference names a *location* in the *store* (also known as the *heap* or just the *memory*).

What is the store?

# The Store

A reference names a *location* in the *store* (also known as the *heap* or just the *memory*).

What is the store?

- *Concretely:* An array of 8-bit bytes, indexed by 32-bit integers.
- *More abstractly:* an array of *values*
- *Even more abstractly:* a partial function from *locations* to *values*.

# Locations

Syntax of values:

| | | |
|---|---|---|
| v ::= | | *values* |
| | unit | *unit constant* |
| | λx:T.t | *abstraction value* |
| | *l* | *store location* |

... and since all values are terms...

# Syntax of Terms

| | | |
|---|---|---|
| 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 |

# Aside

Does this mean we are going to allow programmers to write explicit locations in their programs??

No: This is just a modeling trick. We are enriching the "source language" to include some run-time structures, so that we can continue to formalize evaluation as a relation between source terms.

Aside: If we formalize evaluation in the big-step style, then we can add locations to the set of values (results of evaluation) without adding them to the set of terms.

# Evaluation

The result of evaluating a term now depends on the store in which it is evaluated. Moreover, the result of evaluating a term is not just a value — we must also keep track of the changes that get made to the store.

I.e., the evaluation relation should now map a term and a store to a reduced term and a new store.

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

We use the metavariable $\mu$ to range over stores.

# Evaluation

An assignment $t_1 := t_2$ first evaluates $t_1$ and $t_2$ until they become values...

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 := t_2 \mid \mu \longrightarrow t_1' := t_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1 := t_2 \mid \mu \longrightarrow v_1 := t_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

... and then returns $unit$ and updates the store:

$$l := v_2 \mid \mu \longrightarrow unit \mid [l \rightarrow v_2]\mu \qquad \text{(E-Assign)}$$

A term of the form $\mathtt{ref}\ t_1$ first evaluates inside $t_1$ until it becomes a value...

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{\mathtt{ref}\ t_1 \mid \mu \longrightarrow \mathtt{ref}\ t'_1 \mid \mu'} \qquad \text{(E-Ref)}$$

... and then chooses (allocates) a fresh location $l$, augments the store with a binding from $l$ to $v_1$, and returns $l$:

$$\frac{l \notin dom(\mu)}{\mathtt{ref}\ v_1 \mid \mu \longrightarrow l \mid (\mu, l \to v_1)} \qquad \text{(E-RefV)}$$

A term $!\,t_1$ first evaluates in $t_1$ until it becomes a value...

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{!\,t_1 \mid \mu \longrightarrow !\,t'_1 \mid \mu'} \qquad \text{(E-Deref)}$$

... and then looks up this value (which must be a location, if the original term was well typed) and returns its contents in the current store:

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

Evaluation rules for function abstraction and application are augmented with stores, but don't do anything with them directly.

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t'_1\ t_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t'_2 \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t'_2 \mid \mu'} \qquad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.\, t_{12})\ v_2 \mid \mu \longrightarrow [x \to v_2]t_{12} \mid \mu \qquad \text{(E-AppAbs)}$$

# Aside: garbage collection

Note that we are not modeling garbage collection — the store just grows without bound.

# Aside: pointer arithmetic

We can’t do any!

# Store Typings

# Typing Locations

Q: What is the *type* of a *location*?

# Typing Locations

Q: What is the *type* of a *location*?

A: It depends on the store!

E.g., in the store $(l_1 \to \mathsf{unit}, l_2 \to \mathsf{unit})$, the term $!\, l_2$ has type $\mathsf{Unit}$.

But in the store $(l_1 \to \mathsf{unit}, l_2 \to \lambda x{:}\mathsf{Unit}.x)$, the term $!\, l_2$ has type $\mathsf{Unit} \to \mathsf{Unit}$.

# Typing Locations — first try

Roughly:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \mathsf{Ref}\ T_1}$$

# Typing Locations — first try

Roughly:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \mathsf{Ref}\ T_1}$$

More precisely:

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : \mathsf{Ref}\ T_1}$$

I.e., typing is now a *four*-place relation (between contexts, *stores*, terms, and types).

# Problem

However, this rule is not completely satisfactory. For one thing, it can make typing derivations very large!

E.g., if

$$
\begin{aligned}
(\mu = & \; l_1 \to \lambda x{:}\mathsf{Nat}.\; 999, \\
& \; l_2 \to \lambda x{:}\mathsf{Nat}.\; !\, l_1 \;(!\, l_1 \;\; x), \\
& \; l_3 \to \lambda x{:}\mathsf{Nat}.\; !\, l_2 \;(!\, l_2 \;\; x), \\
& \; l_4 \to \lambda x{:}\mathsf{Nat}.\; !\, l_3 \;(!\, l_3 \;\; x), \\
& \; l_5 \to \lambda x{:}\mathsf{Nat}.\; !\, l_4 \;(!\, l_4 \;\; x)),
\end{aligned}
$$

then how big is the typing derivation for $!\, l_5$?

# Problem!

But wait... it gets worse. Suppose

$$(\mu = l_1 \to \lambda x{:}\mathsf{Nat}.\ !\, l_2\ x,$$
$$l_2 \to \lambda x{:}\mathsf{Nat}.\ !\, l_1\ x),$$

Now how big is the typing derivation for $!\, l_2$?

# Store Typings

Observation: The typing rules we have chosen for references guarantee that a given location in the store is *always* used to hold values of the *same* type.

These intended types can be collected into a *store typing* — a partial function from locations to types.

E.g., for

$$\begin{aligned}\mu = (&l_1 \to \lambda x{:}\mathsf{Nat}.\ 999,\\ &l_2 \to \lambda x{:}\mathsf{Nat}.\ !l_1\ (!l_1\ x),\\ &l_3 \to \lambda x{:}\mathsf{Nat}.\ !l_2\ (!l_2\ x),\\ &l_4 \to \lambda x{:}\mathsf{Nat}.\ !l_3\ (!l_3\ x),\\ &l_5 \to \lambda x{:}\mathsf{Nat}.\ !l_4\ (!l_4\ x)),\end{aligned}$$

A reasonable store typing would be

$$\begin{aligned}\Sigma = (&l_1 \to \mathsf{Nat}\to\mathsf{Nat},\\ &l_2 \to \mathsf{Nat}\to\mathsf{Nat},\\ &l_3 \to \mathsf{Nat}\to\mathsf{Nat},\\ &l_4 \to \mathsf{Nat}\to\mathsf{Nat},\\ &l_5 \to \mathsf{Nat}\to\mathsf{Nat})\end{aligned}$$

Now, suppose we are given a store typing $\Sigma$ describing the store $\mu$ in which we intend to evaluate some term $t$. Then we can use $\Sigma$ to look up the types of locations in $t$ instead of calculating them from the values in $\mu$.

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

I.e., typing is now a four-place relation between between contexts, *store typings*, terms, and types.

# Final typing rules

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

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

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

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

Q: Where do these store typings come from?

A: When we first typecheck a program, there will be no explicit locations, so we can use an empty store typing.

So, when a new location is created during evaluation,

$$\frac{l \notin dom(\mu)}{\mathtt{ref}\ v_1 \mid \mu \longrightarrow l \mid (\mu, l \rightarrow v_1)} \qquad \text{(E-RefV)}$$

we can observe the type of $v_1$ and extend the "current store typing" appropriately.

# Safety

# Preservation

First attempt: just add stores and store typings in the appropriate places.

> *Theorem (?):* If $\Gamma \mid \Sigma \vdash t : T$ and $t \mid \mu \longrightarrow t' \mid \mu'$, then $\Gamma \mid \Sigma \vdash t' : T$.

# Preservation

First attempt: just add stores and store typings in the appropriate places.

> *Theorem (?):* If $\Gamma \mid \Sigma \vdash t : T$ and $t \mid \mu \longrightarrow t' \mid \mu'$, then $\Gamma \mid \Sigma \vdash t' : T$. Wrong!

Why is this wrong?

# Preservation

First attempt: just add stores and store typings in the appropriate places.

> *Theorem (?):* If $\Gamma \mid \Sigma \vdash t : T$ and $t \mid \mu \longrightarrow t' \mid \mu'$, then $\Gamma \mid \Sigma \vdash t' : T$. Wrong!

Why is this wrong?

Because $\Sigma$ and $\mu$ here are not constrained to have anything to do with each other!

# Preservation

A store $\mu$ is said to be *well typed* with respect to a typing context $\Gamma$ and a store typing $\Sigma$, written $\Gamma \mid \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$.

# Preservation

A store $\mu$ is said to be *well typed* with respect to a typing context $\Gamma$ and a store typing $\Sigma$, written $\Gamma \mid \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$.

Next attempt:

*Theorem (?):* If

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

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

# Preservation

A store $\mu$ is said to be *well typed* with respect to a typing context $\Gamma$ and a store typing $\Sigma$, written $\Gamma \mid \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$.

Next attempt:

*Theorem (?):* If

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

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

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

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

Still wrong!

T. What's wrong now?

# Preservation

A store $\mu$ is said to be *well typed* with respect to a typing context $\Gamma$ and a store typing $\Sigma$, written $\Gamma \mid \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$.

Next attempt:

*Theorem (?):* If

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

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

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

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

Still wrong!

Creation of a new reference cell...

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

... breaks the correspondence between the store typing and the store.

# Preservation (correct version)

*Theorem*: If

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

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

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

# Preservation (correct version)

*Theorem*: If

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

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

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

*Proof*: Easy extension of the preservation proof for $\lambda_{\rightarrow}$.

# Progress

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

# Nontermination via references

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

```
t1 = λr:Ref (Unit→Unit).
       (r := (λx:Unit. (!r)x);
        (!r) unit);
t2 = ref (λx:Unit. x);
```

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:

   $\mathtt{fact}_{ref}$ = ref (λn:Nat.0)

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

   $\mathtt{fact}_{body}$ =
   λn:Nat.
   if iszero n then 1 else times n ((!$\mathtt{fact}_{ref}$)(pred n))

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

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

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

   fact = !$\mathtt{fact}_{ref}$ 5

# Exceptions

# 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 `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.

# 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).

| | | |
|---|---|---|
| $t$ | ::= ... | *terms* |
| | $\mathtt{error}$ | *run-time error* |

*Evaluation*

$$\mathtt{error}\ t_2 \longrightarrow \mathtt{error} \qquad \text{(E-AppErr1)}$$

$$v_1\ \mathtt{error} \longrightarrow \mathtt{error} \qquad \text{(E-AppErr2)}$$

# Typing

*Typing*

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

# Typing errors

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

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

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

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

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)
```

# 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!

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

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 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

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

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.

$$v_1 \ \texttt{error} \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

overlap with our existing computation rule for applications:

$$(\lambda x{:}T_{11}.\, t_{12})\ v_2 \longrightarrow [x \to v_2]t_{12} \text{(E-AppAbs)}$$

e.g., the term

$$(\lambda x{:}\mathsf{Nat}.\, 0)\ \texttt{error}$$

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

$t ::= \ldots$ — *terms*

$\quad$ `try t with t` — *trap errors*

*Evaluation*

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

$$\texttt{try error with } t_2 \longrightarrow t_2 \qquad \text{(E-TryError)}$$

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

*Typing*

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

# Exceptions carrying values

| | | |
|---|---|---|
| `t` | ::= … | *terms* |
| | `raise t` | *raise exception* |

# Evaluation

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

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

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

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

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

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

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

# Typing

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.

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

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

# What is $T_{exn}$?

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

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

# 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: $T_{exn} = \mathsf{Nat}$ (as in C)
2. Error messages: $T_{exn} = \mathsf{String}$
3. A predefined variant type:

$$
\begin{array}{lll}
T_{exn} = & \langle \mathsf{divideByZero}: & \mathsf{Unit}, \\
& \mathsf{overflow}: & \mathsf{Unit}, \\
& \mathsf{fileNotFound}: & \mathsf{String}, \\
& \mathsf{fileNotReadable}: & \mathsf{String}, \\
& \ldots \rangle &
\end{array}
$$

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: $T_{exn} = \mathtt{Nat}$ (as in C)
2. Error messages: $T_{exn} = \mathtt{String}$
3. A predefined variant type:

$$
\begin{array}{lll}
T_{exn} & = & \langle \mathtt{divideByZero}: \quad \mathtt{Unit}, \\
 & & \quad \mathtt{overflow}: \quad \mathtt{Unit}, \\
 & & \quad \mathtt{fileNotFound}: \quad \mathtt{String}, \\
 & & \quad \mathtt{fileNotReadable}: \quad \mathtt{String}, \\
 & & \quad \ldots \ \rangle
\end{array}
$$

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