A Little More on References
Recap

Last time, we discussed how to formalize languages with mutable state...
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
    x
    $\lambda x:T.t$
    t t
    ref t
    !t
    t:=t
    /
Evaluation

Evaluation becomes a four-place relation: \( t \mid \mu \longrightarrow t' \mid \mu' \)

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

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

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

(Plus several congruence rules.)
Typing becomes a three-place relation: $\Gamma \mid \Sigma \vdash t : T$

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

\[
\frac{\Gamma \mid \Sigma \vdash t_1 : T_1
}{\Gamma \mid \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1}
\]

\[
\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11}
}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}}
\]

\[
\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11}, \Gamma \mid \Sigma \vdash t_2 : T_{11}
}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}}
\]

(T-Loc)

(T-Ref)

(T-Deref)

(T-Assign)
Preservation

**Theorem:** If

\[ \Gamma \mid \Sigma \vdash t : T \]
\[ \Gamma \mid \Sigma \vdash \mu \]
\[ t \mid \mu \rightarrow t' \mid \mu' \]

then, for some \( \Sigma' \supseteq \Sigma \),

\[ \Gamma \mid \Sigma' \vdash t' : T \]
\[ \Gamma \mid \Sigma' \vdash \mu'. \]
**Theorem:** Suppose \( t \) is a closed, well-typed term (that is, \( \emptyset \vdash \Sigma \vdash t : T \) for some \( T \) and \( \Sigma \)). Then either \( t \) is a value or else, for any store \( \mu \) such that \( \emptyset \vdash \Sigma \vdash \mu \), there is some term \( t' \) and store \( \mu' \) with \( t \mid \mu \rightarrow t' \mid \mu' \).
Nontermination via references

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

\[
\begin{align*}
t_1 &= \lambda r: \text{Ref} \ (\text{Unit} \rightarrow \text{Unit}). \\
&\quad \ (r := (\lambda x: \text{Unit}. (!r)x); \\
&\quad \quad (!r) \ \text{unit}); \\
t_2 &= \text{ref} \ (\lambda x: \text{Unit}. x);
\end{align*}
\]

Applying \(t_1\) to \(t_2\) 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:
   \[
   \text{fact}_{ref} = \text{ref} \left( \lambda n: \text{Nat}. 0 \right)
   \]

2. Define the body of the function we are interested in, using the contents of the reference cell for making recursive calls:
   \[
   \text{fact}_{body} = \\
   \lambda n: \text{Nat}.
   \text{if iszero } n \text{ then } 1 \text{ else times } n \left( (\text{!fact}_{ref})(\text{pred } n) \right)
   \]

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:
   \[
   \text{fact} = \text{!fact}_{ref} \\
   \text{fact} \ 5
   \]
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).

$$t ::= \ldots$$  terms

error  run-time error

Evaluation

$$\text{error } t_2 \rightarrow \text{error} \quad (E\text{-AppErr1})$$

$$v_1 \text{ error } \rightarrow \text{error} \quad (E\text{-AppErr2})$$

▶ What if we had booleans and numbers in the language?
Typing

\[ \Gamma \vdash \text{error} : T \quad (T-\text{ERROR}) \]
Typing errors

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

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

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

\[ \Gamma \vdash \text{error} : T \]  \hspace{1cm} (T-\text{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?

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

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)`
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?

\[ \Gamma \vdash (\text{error as } T) : T \quad (T\text{-Error}) \]

No, this doesn’t work!

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

\[
\begin{align*}
\text{succ} \ (\text{if (error as Bool) then 5 else 7}) \\
\rightarrow \text{succ (error as Bool)}
\end{align*}
\]

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!

\[ \Gamma \vdash \text{error} : \texttt{'a} \quad (\text{T-Error}) \]

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

Alternatively, in a system with subtyping (which we’ll discuss in the next lecture) and a minimal \texttt{Bot} type, we \textit{can} give \texttt{error} a unique type:

\[
\Gamma \vdash \texttt{error} : \texttt{Bot} \quad \text{(T-\texttt{Error})}
\]

(Of course, what we’ve really done is just pushed the complexity of the old \texttt{error} rule onto the \texttt{Bot} type! We’ll return to this point later.)
For now...

Let’s stick with the original rule

$$\Gamma \vdash \text{error} : T$$ \hfill (T-\text{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.
First, note that we do not want to extend the set of values to include \texttt{error}, since this would make our new rule for propagating errors through applications.

\[ v_1 \text{ error} \longrightarrow \text{ error} \quad (E\text{-AppErr2}) \]

overlap with our existing computation rule for applications:

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

e.g., the term

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

could evaluate to either \texttt{0} (which would be wrong) or \texttt{error} (which is what we intend).
Instead, we keep \texttt{error} as a non-value normal form, and refine the statement of progress to explicitly mention the possibility that terms may evaluate to \texttt{error} instead of to a value.

\textbf{Theorem [Progress]:} Suppose \( t \) is a closed, well-typed normal form. Then either \( t \) is a value or \( t = \texttt{error} \).
Catching exceptions

t ::= ... terms

\[ \text{try } t \text{ with } t \text{ trap errors} \]

Evaluation

\[ \text{try } v_1 \text{ with } t_2 \longrightarrow v_1 \]  \hspace{1cm} (E-TRYV)

\[ \text{try error with } t_2 \longrightarrow t_2 \]  \hspace{1cm} (E-TRYERROR)

\[ \begin{array}{c}
    t_1 \longrightarrow t'_1 \\
    \hline
    \text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2
\end{array} \]  \hspace{1cm} (E-TRY)

Typing

\[ \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \]

\[ \Gamma \vdash \text{try } t_1 \text{ with } t_2 : T \]  \hspace{1cm} (T-TRY)
Exceptions carrying values

\[ t ::= \ldots \]  
\[ \text{raise } t \]
Evaluation

\[(\text{raise } v_{11}) \ t_2 \longrightarrow \text{raise } v_{11} \quad (E\text{-AppRaise1})\]

\[v_1 \ (\text{raise } v_{21}) \longrightarrow \text{raise } v_{21} \quad (E\text{-AppRaise2})\]

\[t_1 \longrightarrow t'_1 \quad (E\text{-Raise})\]

\[\frac{\text{raise } t_1 \longrightarrow \text{raise } t'_1}{\text{raise } \text{raise } v_{11} \longrightarrow \text{raise } v_{11}} \quad (E\text{-RaiseRaise})\]

\[\text{try } v_1 \text{ with } t_2 \longrightarrow v_1 \quad (E\text{-TryV})\]

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

\[\frac{t_1 \longrightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2} \quad (E\text{-Try})\]
Typing

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

$$\Gamma \vdash t_1 : T_{\text{exn}}$$

\[ \Gamma \vdash \text{raise } t_1 : T \]  \hspace{1cm} (T-EXN)

$$\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_{\text{exn}} \rightarrow T$$

\[ \Gamma \vdash \text{try } t_1 \text{ with } t_2 : T \]  \hspace{1cm} (T-TRY)
What is $T_{\text{exn}}$?

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

1. Numeric error codes: $T_{\text{exn}} = \text{Nat}$ (as in C)
What is $T_{\text{exn}}$?

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

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

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

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

   $$T_{\text{exn}} = \langle \text{divideByZero: Unit,} \ \
   \text{overflow: Unit,} \ \
   \text{fileNotFound: String,} \ \
   \text{fileNotReadable: String,} \ \
   ... \rangle$$
What is $T_{\text{exn}}$?

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

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

   $$T_{\text{exn}} = \langle \text{divideByZero: Unit,}$$
   $$\text{overflow: Unit,}$$
   $$\text{fileNotFound: String,}$$
   $$\text{fileNotReadable: String,}$$
   $$\ldots \rangle$$

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

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

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