CIS 500
Software Foundations
Fall 2006

October 25

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`	*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: $t \mid \mu \longrightarrow t' \mid \mu'$

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

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

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

(Plus several congruence rules.)

Typing

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

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

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

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

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

Preservation

Theorem: If

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

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

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

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\emptyset \mid \Sigma \vdash \texttt{t} : \texttt{T}$ 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 $\texttt{t}'$ and store μ' with $\texttt{t} \mid \mu \longrightarrow \texttt{t}' \mid \mu'$.

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:

   ```
   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:

   ```
   fact_body =
    λn:Nat.
      if iszero n then 1 else times n ((!fact_ref)(pred n))
   ```

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

   ```
   fact_ref := fact_body
   ```

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

   ```
   fact = !fact_ref
   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` ::= ...	*terms*
`error`	*run-time error*

Evaluation

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

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

- What if we had booleans and numbers in the language?

Typing

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 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

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

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.

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

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

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

t ::=	...	*terms*
	try t with t	*trap errors*

Evaluation

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

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

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

Typing

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

Exceptions carrying values

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

Evaluation

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

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

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

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

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

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

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

Typing

To typecheck raise expressions, we need to choose a type — let's call it $\texttt{T}_{exn}$ — for the values that are carried along with exceptions.

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

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

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: $T_{exn} = \texttt{Nat}$ (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: $T_{exn} = \texttt{Nat}$ (as in C)
2. Error messages: $T_{exn} = \texttt{String}$

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

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

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

```
T_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)