SNU 4541.574
Programming Language Theory

ack: BCP's slides

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*
	λ`x: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: $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$

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

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

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

(Plus several congruence rules.)

Typing

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

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

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

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

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

Preservation

Theorem: If

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

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

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

Progress

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

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:

   $\texttt{fact}_{ref}$ `= ref (`λ`n:Nat.0)`

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

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

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

   $\texttt{fact}_{ref}$ `:=` $\texttt{fact}_{body}$

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

   `fact = !`$\texttt{fact}_{ref}$
   `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 $\lambda_{\to}$

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

$$\Gamma \vdash \texttt{error} : \texttt{T} \qquad \text{(T-E}\textsc{rror}\text{)}$$

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.

An 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.

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.

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

overlap with our existing computation rule for applications:

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

e.g., the term

```
(λx: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 $t = \mathtt{error}$.*

Catching exceptions

$$\begin{array}{lll} \mathtt{t} & ::= & \ldots \\ & & \mathtt{try\ t\ with\ t} \end{array}$$

terms

trap errors

Evaluation

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

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

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

Typing

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

Exceptions carrying values

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

Evaluation

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

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

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

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

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

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

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

Typing

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

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

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

What is $\mathtt{T}_{exn}$?

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

1. Numeric error codes: $\mathtt{T}_{exn} = \mathtt{Nat}$ (as in C)

What is $\mathtt{T}_{exn}$?

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

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

What is $\mathtt{T}_{exn}$?

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

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

$$\begin{array}{lll} \mathtt{T}_{exn} & = & \mathtt{<divideByZero:} \quad \mathtt{Unit}, \\ & & \ \mathtt{overflow:} \quad \mathtt{Unit}, \\ & & \ \mathtt{fileNotFound:} \quad \mathtt{String}, \\ & & \ \mathtt{fileNotReadable:} \quad \mathtt{String}, \\ & & \ \mathtt{...\ >} \end{array}$$

What is $\mathtt{T}_{exn}$?

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

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

$$\begin{array}{lcll} \mathtt{T}_{exn} & = & \mathtt{<divideByZero:} & \mathtt{Unit}, \\ & & \mathtt{\ overflow:} & \mathtt{Unit}, \\ & & \mathtt{\ fileNotFound:} & \mathtt{String}, \\ & & \mathtt{\ fileNotReadable:} & \mathtt{String}, \\ & & \mathtt{\ ...\ >} & \end{array}$$

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} = `Nat` (as in C)
2. Error messages: T_{exn} = `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)