CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2005

Lecture 14

Polymorphism Wrap-up; Exceptions; References;
Polymorphic References; C-Style Pointers

Where are we

Done investigating subtyping and type variables:

- *Both* forms of polymorphism permit code reuse and abstraction
  - Reuse: Same code for multiple types
  - Abstraction: Code has references to data it cannot use arbitrarily
  - Abstraction broken by downcasts or `instanceof`
- Different idioms best supported by subsumption or type variables.
- Recursive types also use α, for a different purpose:
  - But reused subsumption machinery for "equal to unrolling"
  - But reused type-variable machinery for "unrolling is type substitution"

Today: Exceptions and mutable references

Exceptions

Defining `raise` (a.k.a. `throw`) and `try` is straightforward if a bit cumbersome. Key ideas:

- Can define a raise as "bubbling up" to a matching catch.
- Hierarchical exceptions affect whether a catch matches (not in ML).
  - Fits in nicely with class-based OOP (coming soon).
- Types in ML typically do not mention possible exceptions; types in Java sort of do.

Operational Semantics

Let's model exceptions with just integers for now:

$$e ::= \ldots \mid \textbf{raise } e \mid \textbf{try } e \textbf{ catch } (c)\ e$$

Rules for new constructs:

$$\frac{e \longrightarrow e'}{\textbf{raise } e \longrightarrow \textbf{raise } e'} \qquad \frac{e_1 \longrightarrow e_1'}{\textbf{try } e_1 \textbf{ catch } (c)\ e_2 \longrightarrow \textbf{try } e_1' \textbf{ catch } (c)\ e_2}$$

$$\frac{}{\textbf{try } v \textbf{ catch } (c)\ e_2 \longrightarrow v} \qquad \frac{}{\textbf{try raise } c \textbf{ catch } (c)\ e_2 \longrightarrow e_2}$$

$$\frac{c \neq c'}{\textbf{try raise } c' \textbf{ catch } (c)\ e_2 \longrightarrow \textbf{raise } c'}$$

More Bubbles

And lots more bubble-up rules (can avoid this with a list of evaluation contexts). Examples:

$$\frac{}{(\textbf{raise } c)\ e \longrightarrow \textbf{raise } c} \qquad \frac{}{v\ (\textbf{raise } c) \longrightarrow \textbf{raise } c}$$

$$\frac{}{(\textbf{raise } c, e) \longrightarrow \textbf{raise } c} \qquad \frac{}{(v, \textbf{raise } c) \longrightarrow \textbf{raise } c}$$

Final result may be a v or an uncaught exception **raise** c

Efficiency note: Some compilers do raise in $O(1)$, but not a big deal to take $O(n)$ to pop off stack of size n since you built it in time $O(n)$.

Continuations note: letcc can restore an old stack, exceptions can't.

Typing Exceptions

ML-style typing of exceptions (though still just integers):

$$\frac{\Delta;\Gamma \vdash e : \textbf{int} \qquad \Delta \vdash \tau}{\Delta;\Gamma \vdash \textbf{raise}\ e : \tau} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau \qquad \Delta;\Gamma \vdash e_2 : \tau}{\Delta;\Gamma \vdash \textbf{try}\ e_1\ \textbf{catch}\ (c)\ e_2 : \tau}$$

Safety caveat: Any expression of type τ might evaluate to **raise** c.

Try-rule is like an if (strange-looking only if you're used to e_1 and e_2 being "statements" (i.e., of type unit)).

Carrying Values

If we change exceptions to carry a pair of an integer and another value, we're more like ML (or Java):

- The integer *tag* is the constructor (the class).
- The other value is the carried-value (the fields).

But the `exn` type is unlike other datatypes: Any file can add new constructors so no file can know them all!

Called an *extensible datatype*; makes exhaustiveness impossible.

Cannot reuse tags or could violate type-safety!

Implementation is clever: represent constructors not with integers *per se*, but with addresses — the linker guarantees uniqueness!

Mutable References

We had mutable variables in lecture 2; what's different now:

- Not all data is integers: mutable location can hold pairs, functions, other mutable locations, etc.
- We can create new references at run-time.
- We have types, including type variables and subtyping.
- We sometimes have termination.

References take *extreme care*.

They mess up many of our nice properties, but we are type-safe if we are careful.

ML-style references

For now, let's take the ML approach:

- Variables are *not* mutable.
- *References* are (pointers to) mutable things; *dereference* is explicit.

$$e ::= \ldots \mid \textbf{ref}\ e \mid\ !e \mid e1 := e2$$

But our operational semantics needs a *heap* and *reference-values* (a.k.a. *addresses*):

$$\begin{array}{rcl} e & ::= & \ldots \mid r \\ v & ::= & \ldots \mid r \\ H & ::= & \cdot \mid H, r \mapsto v \end{array}$$

A program state is now $(H; e)$ so all our old rules change. Examples:

$$\frac{}{H; (\lambda x.\ e)\ v \longrightarrow (H; e[v/x])} \qquad \frac{H; e_1 \longrightarrow H'; e_1'}{H; e_1\ e_2 \longrightarrow (H'; e_1'\ e_2)}$$

Interesting Rules

The rules for our new non-value expressions affect the heap.

$$\frac{r \notin \mathbf{Dom}(H)}{(H; \textbf{ref } v) \longrightarrow (H, r \mapsto v; r)} \qquad \frac{}{(H; !r) \longrightarrow (H; H(r))}$$

$$\frac{}{(H; r := v) \longrightarrow (H, r \mapsto v; ())}$$

(4 new boring rules omitted)

Typing

A reference holding a τ is not a τ; the former supports (only) dereference and assignment; the latter does not.

$$\tau ::= \ldots \mid \tau \textbf{ ref}$$

Just 3 new typing rules for your type-checker:

$$\frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \textbf{ref } e : \tau \textbf{ ref}} \qquad \frac{\Delta;\Gamma \vdash e : \tau \textbf{ ref}}{\Delta;\Gamma \vdash !e : \tau}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \textbf{ ref} \qquad \Delta;\Gamma \vdash e_2 : \tau}{\Delta;\Gamma \vdash e_1 := e_2 : \textbf{unit}}$$

This is safe, but the Preservation proof won't go through!

Preservation and States

In practice you write your type-checker only for source programs.

But Preservation requires type-checking states (details not so crucial).

So *extend* the type system just for the proof to include reference values and heaps:

$$\Gamma ::= \ldots \mid r{:}\tau$$

$$\boxed{\Delta;\Gamma \vdash e : \tau} \quad \boxed{\Gamma_1 \vdash H : \Gamma_2} \quad \boxed{\vdash H; e}$$

$$\frac{}{\Delta;\Gamma \vdash r : (\Gamma(r))\ \textbf{ref}} \qquad \frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash H : \Gamma' \qquad \cdot;\Gamma \vdash v : \tau}{\Gamma \vdash H, r \mapsto v : \Gamma', r{:}\tau}$$

$$\frac{\Gamma \vdash H : \Gamma \qquad \cdot;\Gamma \vdash e : \tau}{\vdash H; e}$$

(Above has a technical trick to allow cycles in the heap.)

Preservation and States

Preservation: If $\cdot; \Gamma \vdash e : \tau$ and $\Gamma \vdash H : \Gamma$ and $H; e \longrightarrow H'; e'$, then there exists a Γ' such that $\cdot; \Gamma' \vdash e' : \tau$ and $\Gamma' \vdash H' : \Gamma'$ and Γ' *is an extension* of Γ (crucial for induction).

Note: Every step via a "boring rule" requires Weakening because the heap might get bigger.

In this math is a *key* idea: safety requires *type-invariance* of the heap...

As program runs, new heap locations can arrive and old locations can change value, but no location ever changes type! If it did, the program state might not type-check anymore.

All done?

That's about all there is to say about references, except how they interact (badly) with everything else:

- Subtyping
- Parametric polymorphism
- Termination in the absence of μ or fix
- Parametricity

Then: We can take a C/Java approach to formalizing mutable locations by distinguishing "left-evaluation" from "right-evaluation".

Subtyping

For subtyping, when should we allow τ **ref** $\leq \tau'$ **ref**?

Allow covariance ($\tau \leq \tau' \Rightarrow \tau$ **ref** $\leq \tau'$ **ref**)?

```
let x : {.l1:int, .l2:int} ref = ref {.l1=0, .l2=0} in
x := {.l1=0}; (* subsume left-side *)
(!x.l2)
```

Allow contravariance ($\tau' \leq \tau \Rightarrow \tau$ **ref** $\leq \tau'$ **ref**)?

```
let x : {.l1:int} ref = ref {.l1=0} in
let y : {.l1:int,.l2:int} ref = x in (* subsume x *)
(!y.l2)
```

Reference types are *invariant* (common mistake)!!!

$$\frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_1}{\tau_1 \text{ ref} \leq \tau_2 \text{ ref}}$$

Universal types

```
let x : forall 'a. (('a list) ref) = ref [] in
x [int] := 1::[];
match !(x [string]) with hd::tl -> hd ^ "gotcha!" | _ -> ()
```

We must reject something in this example!

- ML solution: Do not let `ref []` have a polymorphic type.
  - But a library can "hide" the ref type (`type 'a foo = 'a ref`), so make all function applications have non-polymorphic types (the "value restriction").
- Several other solutions exist. (Example: Only let (immutable) functions have polymorphic types, but then `[]` cannot have a polymorphic type.)

Termination

For Turing-completeness all you need is simply-typed λ calculus plus references.

Here's an infinite loop that type-checks:

```
let x : (int->int) ref = ref (fn y -> y)  in
let f : (int->int)     = (fn y -> (!x) y) in
x := f;
f 0
```

It's pretty easy to encode arbitrary recursion this way.

After all, it's just like back-patching in an assembler!

Parametricity

True: In System F with references, If `f` has type $\forall\boldsymbol{\alpha}.\mathbf{int} \rightarrow \boldsymbol{\alpha} \rightarrow \mathbf{int}$, then `f` $[\boldsymbol{\tau_1}]\ \boldsymbol{c}\ \boldsymbol{v_1}$ and `f` $[\boldsymbol{\tau_2}]\ \boldsymbol{c}\ \boldsymbol{v_2}$ always have the same behavior.

False: In System F with references, If `f` has type $\forall\boldsymbol{\alpha}.(\mathbf{int\ ref}) \rightarrow (\boldsymbol{\alpha}\ \mathbf{ref}) \rightarrow \mathbf{int}$, then `f` $[\boldsymbol{\tau_1}]\ \boldsymbol{r_1}\ \boldsymbol{r_2}$ and `f` $[\boldsymbol{\tau_2}]\ \boldsymbol{r_1}\ \boldsymbol{r_3}$ always have the same behavior.

Where are we

Investigated mutation in a language much more real than IMP.

Used the ML approach: distinguish references from immutable things:

- So variables still "map to values"
- And the type-checker just treats references as a library:

```
type 'a ref;
val ref : 'a -> 'a ref;
val ! : 'a ref -> 'a;
val := : 'a ref -> 'a -> unit
```

  Which does not quite work without the value restriction

In C, C++, Java, etc.:

- "Every variable is a reference"
- In certain positions "the dereference is implicit"

Formalizing C-style pointers

(Note: Most researchers never do it this way, but I find it useful.)

$$
\begin{array}{rcl}
\tau & ::= & \textbf{int} \mid \tau* \\
e & ::= & c \mid x \mid e = e \mid *e \mid \&e \mid e; e \\
v & ::= & c \mid \&x \\
H & ::= & \cdot \mid H, x \mapsto v
\end{array}
$$

For simplicity, *large-step* semantics:

$$\boxed{H; e \Downarrow_R H'; v} \qquad \boxed{H; e \Downarrow_L H'; x}$$

The judgments are inter-related
(i.e., the interpreter uses mutual recursion).

The Rules

$$\frac{}{H; c \Downarrow_R H; c} \qquad \frac{}{H; x \Downarrow_R H; H(x)}$$

$$\frac{H; e_1 \Downarrow_L H'; x \qquad H'; e_2 \Downarrow_R H''; v}{H; e_1 = e_2 \Downarrow_R H'', x \mapsto v; v} \qquad \frac{H; e \Downarrow_R H'; \&x}{H; *e \Downarrow_R H'; H'(x)}$$

$$\frac{H; e \Downarrow_L H'; x}{H; \&e \Downarrow_R H'; \&x} \qquad \frac{H; e_1 \Downarrow_R H'; v_1 \qquad H'; e_2 \Downarrow_R H''; v_2}{H; e_1; e_2 \Downarrow_R H''; v_2}$$

$$\frac{}{H; x \Downarrow_L H; x} \qquad \frac{H; e \Downarrow_R H'; \&x}{H; *e \Downarrow_L H'; x}$$

C Revealed

C expressions according to Dan:

- Right expressions evaluate to values.
- Left expressions evaluate to locations.
- Right expressions have implicit lookup; left expressions don't.
- & turns a left expression into a right expression.
- * (can) turn a right expression into a left expression.

Opinion:

- This slide (not the last one) is how to teach C (to sophomores).
- People have an irrational hatred of &.
- They should rationally hate it because it introduces dangling pointers and, more generally, creates aliases where it *looks like* there are not any.