CSE 505:
Concepts of Programming Languages

Dan Grossman

Fall 2009

Lecture 11— Lambda Interpreters; Polymorphism; Subtyping

Where are we

- Recall our use of *evaluation contexts* to make a more concise operational semantics
- Now use that basic concept to write an efficient interpreter that supports *continuations*
  - May hand-wave some details, but will post all the code
- Then "back to types"
  - Flavors of polymorphism
  - Start subtyping

Review

Evaluation contexts: expressions with 1 hole where "real work" occurs:

$$\begin{array}{lll} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\ & \mid & \mathsf{A}(E) \mid \mathsf{B}(E) \mid (\mathbf{match}\ E\ \mathbf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2) \end{array}$$

Define "filling the hole" $E[e]$ in the obvious way (see ML code).
Semantics is now just:

$$\frac{e \xrightarrow{\text{p}} e'}{E[e] \rightarrow E[e']}$$

$$\frac{}{(\lambda x.\ e)\ v \xrightarrow{\text{p}} e[v/x]} \qquad \frac{}{(v_1, v_2).1 \xrightarrow{\text{p}} v_1} \qquad \frac{}{(v_1, v_2).2 \xrightarrow{\text{p}} v_2}$$

$$\frac{}{\mathbf{match}\ \mathsf{A}(v)\ \mathbf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \xrightarrow{\text{p}} e_1[v/x]}$$

$$\frac{}{\mathbf{match}\ \mathsf{B}(v)\ \mathbf{with}\ \mathsf{A}y.\ e_1 \mid \mathsf{B}x.\ e_2 \xrightarrow{\text{p}} e_2[v/x]}$$

Theorem (Unique Decomposition): If $\cdot \vdash e : \tau$, then e is a value or there is exactly one decomposition of e.

Continuations

First-class continuations in one slide:

$$\begin{array}{rcl} e & ::= & \ldots \mid \textbf{letcc } x.\ e \mid \textbf{throw } e\ e \mid \textbf{cont } E \\ v & ::= & \ldots \mid \textbf{cont } E \\ E & ::= & \ldots \mid \textbf{throw } E\ e \mid \textbf{throw } v\ E \end{array}$$

$$\frac{}{E[\textbf{letcc } x.\ e] \rightarrow E[(\lambda x.\ e)(\textbf{cont } E)]}$$

$$\frac{}{E[\textbf{throw } (\textbf{cont } E')\ v] \rightarrow E'[v]}$$

Very powerful and general: For example, non-preemptive multithreading *in the language*.

This is the real power of evaluation contexts: *reifying them*.

Connection to interpreters

A "real" (efficient, natural) interpreter for lambda-calculus (or ML) would not be like our small-step semantics

- Would re-decompose the whole program for each step!

Instead, maintain the decomposition incrementally

- With a stack to remember "what to work on next"!

Also, don't use substitution; use environments (see your homework)

- At this point, need just one while-loop, pairs, and malloc

And if your stacks are heap-allocated and immutable, you can implement continuation operations (letcc and throw) in $O(1)$ time.

- A nice (and provably correct) sequence of more primitive interpreters

Where are we

- Have an operational model of functions, data structures, primitives, etc.
- Have a simple type system to ensure we use functions as functions, pairs as pairs, constants as constants, ...
- Digressed to:
  - compare types to logic
  - use evaluation contexts to define continuations
- Haven't done recursive types (e.g., lists) and exceptions.
  - Mutation on homework
- But first, *be less restrictive without affecting run-time behavior*

Being Less Restrictive

"Will a λ term get stuck?" is undecidable, so a sound, decidable type system can *always* be made less restrictive.

An "uninteresting" rule that is sound but not "admissable":

$$\frac{\Gamma \vdash e_1 : \tau}{\Gamma \vdash \textbf{if true then } e_1 \textbf{ else } e_2 : \tau}$$

We'll study ways to give one term many types ("polymorphism").

Fact: The version of STλC with explicit argument types ($\lambda x : \tau.\ e$) has no polymorphism:
If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$.

Fact: Even without explicit types, many "reuse patterns" do not type-check. Example: $(\lambda f.\ (f\ 0, f\ \textbf{true}))(\lambda x.\ (x, x))$ (evaluates to $((0, 0), (\textbf{true}, \textbf{true}))$).

My least favorite PL word

Polymorphism means many things...

- *Ad hoc polymorphism:* $e_1 + e_2$ in SML<C<Java<C++
- *Ad hoc, cont'd:* Maybe e_1 and e_2 can have different *run-time* types and we choose the $+$ based on them
- *Parametric polymorphism:* e.g., $\Gamma \vdash \lambda x.\ x : \forall\alpha.\alpha \to \alpha$ or with explicit types: $\Gamma \vdash \Lambda\alpha.\ \lambda x : \alpha.\ x : \forall\alpha.\alpha \to \alpha$ (which "compiles" i.e. "erases" to $\lambda x.\ x$)
- *Subtype polymorphism:* `new Vector().add(new C())` is legal Java because `new C()` has types `Object` and `C`

...and nothing.

(I prefer "static overloading" "dynamic dispatch" "type abstraction" and "subtyping")

Our plan

- Starting today: Subtyping, preferably without coercions
- Then: Parametric polymorphism (∀) and maybe first-class ADTs (∃) and recursive types (μ).
  (All use type variables (α).)
- Even later: Dynamic-dispatch, inheritance vs. subtyping, etc.
  (Concepts in OO programming)

Today's Motto: Subtyping is not a matter of opinion!

Record types

We'll use records to motivate subtyping:

$$\begin{array}{rcl} e & ::= & \ldots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\ \tau & ::= & \ldots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\ v & ::= & \ldots \mid \{l_1 = v_1, \ldots, l_n = v_n\} \end{array}$$

$$\frac{e_i \to e_i'}{\begin{array}{c}\{l_1{=}v_1, \ldots, l_{i-1}{=}v_{i-1}, l_i{=}e_i, \ldots, l_n{=}e_n\} \\ \to \{l_1{=}v_1, \ldots, l_{i-1}{=}v_{i-1}, l_i{=}e_i', \ldots, l_n{=}e_n\}\end{array}} \qquad \frac{e \to e'}{e.l \to e.l}$$

$$\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n \qquad \text{labels distinct}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \qquad 1 \le i \le n}{\Gamma \vdash e.l_i : \tau_i}$$

Should this typecheck?

$(\lambda x : \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}.\ x.l_1 + x.l_2)\{l_1{=}3, l_2{=}4, l_3{=}5\}$

Right now, it doesn't.

Our operational semantics won't get stuck.

Suggests *width subtyping*:

$\boxed{\tau_1 \leq \tau_2}$

$$\frac{}{\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n, l{:}\tau\} \leq \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}}$$

And one one new type-checking rule: *Subsumption*

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau}$$

Permutation

Our semantics for projection doesn't care about position...

So why not let $\{l_1{=}3, l_2{=}4\}$ have type $\{l_2{:}\mathsf{int}, l_1{:}\mathsf{int}\}$?

$$\frac{}{\{l_1{:}\tau_1, \ldots, l_{i-1}{:}\tau_{i-1}, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, l_{i-1}{:}\tau_{i-1}, \ldots, l_n{:}\tau_n\}}$$

Example with width: Show
$\cdot \vdash \{l_1{=}7, l_2{=}8, l_3{=}9\} : \{l_2{:}\mathsf{int}, l_1{:}\mathsf{int}\}$.

It's no longer clear what an (efficient, sound, complete) algorithm should be. They sometimes exist and sometimes don't. Here they do.

Transitivity

Subtyping is always transitive. We can add a rule for that:

$$\frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

Or just use the subsumption rule multiple times.

Or both.

In any case, type-checking is no longer syntax-directed: Given $\Gamma \vdash e : \tau_1$, there may be 0, 1, or many ways to show $\Gamma \vdash e : \tau_2$.

So we could (hopefully) define an algorithm and prove it succeeds exactly when there exists a derivation.

Digression: Efficiency

With our semantics, width and permutation subtyping make perfect sense.

But it would be nice to compile $e.l$ down to:

1. evaluate e to a record stored at an address a
2. load a into a register r_1
3. load field l *from a fixed offset (e.g., 4) into* r_2

Many type systems are engineered to make this easy for compiler writers.

Makes restrictions seem odd if you do not know techniques for implementing high-level languages. (CSE501)

Digression continued

With width subtyping, the strategy is easy. (No problem.)

With permutation subtyping, it's easy but have to "alphabetize".

With both, it's not easy...

$f_1 : \{l_1 : \mathsf{int}\} \rightarrow \mathsf{int} \quad f_2 : \{l_2 : \mathsf{int}\} \rightarrow \mathsf{int}$
$x_1 = \{l_1 = 0, l_2 = 0\} \quad x_2 = \{l_2 = 0, l_3 = 0\}$
$f_1(x_1) \quad f_2(x_1) \quad f_2(x_2)$

Can use *dictionary-passing* (look up offset at run-time) and maybe *optimize away* (some) lookups.

Named types can avoid this, but make code less flexible.

Depth Subtyping

With just records of ints, we miss another opportunity:

$(\lambda x : \{l_1{:}\{l_3{:}\mathsf{int}\}, l_2{:}\mathsf{int}\}.\ x.l_1.l_3 + x.l_2)$

$\{l_1{=}\{l_3 = 3, l_4 = 9\}, l_2{=}4\}$

Again, does not type-check but does not get stuck.

$$\frac{\tau_i \leq \tau_i'}{\{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i', \ldots, l_n{:}\tau_n\}}$$

(With permutation subtyping could just allow depth on left-most field)

Soundness of this rule depends *crucially* on fields being *immutable*. (Depth subtyping is *unsound* in the presence of mutation.)

- Trade-off between power (mutation) and sound expressiveness (depth subtyping)

Function subtyping

Given our rich subtyping on records, how do we extend it to other types, namely $\tau_1 \rightarrow \tau_2$. For example, with width subtyping we'd like $\mathbf{int} \rightarrow \{l_1{:}\mathbf{int}, l_2{:}\mathbf{int}\} \leq \mathbf{int} \rightarrow \{l_1{:}\mathbf{int}\}$.

$$\frac{???}{\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4}$$

For a function to have type $\tau_3 \rightarrow \tau_4$ it must return something of type τ_4 (including subtypes) whenever given something of type τ_3 (including subtypes). A function assuming less than τ_3 will do, but not one assuming more.

Function subtyping, cont'd

$$\frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4}{\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4} \qquad \text{Also want: } \frac{}{\tau \leq \tau}$$

Example: $\lambda x : \{l_1\text{:int}, l_2\text{:int}\}.\ \{l_1 = x.l_2, l_2 = x.l_1\}$ can have type $\{l_1\text{:int}, l_2\text{:int}, l_3\text{:int}\} \rightarrow \{l_1\text{:int}\}$
but *not* $\{l_1\text{:int}\} \rightarrow \{l_1\text{:int}\}$.
We say function types are *contravariant* in their argument and *covariant* in their result.

We say function types are contravariant in their argument *with our eyes closed*, **on one foot**, IN OUR SLEEP, and we never let anybody tell us otherwise. Ever.

(Depth subtyping means immutable records are covariant in their fields.)

Summary of subtyping additions

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau}$$

$\boxed{\tau_1 \leq \tau_2}$

$$\frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \qquad \frac{}{\tau \leq \tau}$$

$$\frac{}{\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n, l{:}\tau\} \leq \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}}$$

$$\frac{}{\begin{array}{c}\{l_1{:}\tau_1, \ldots, l_{i-1}{:}\tau_{i-1}, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \\ \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, l_{i-1}{:}\tau_{i-1}, \ldots, l_n{:}\tau_n\}\end{array}}$$

$$\frac{\tau_i \leq \tau_i'}{\{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i', \ldots, l_n{:}\tau_n\}}$$

$$\frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4}{\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4}$$

(For other types, e.g., sums or pairs, would have more rules.)

Maintaining soundness

Our Preservation and Progress Lemmas still work in the presence of subsumption. (So in theory, any subtyping mistakes would be caught when trying to prove soundness!)

In fact, it seems too easy: induction on typing derivations makes the subsumption case easy:

- Progress: One new case if typing derivation $\cdot \vdash e : \tau$ ends with subsumption. Then $\cdot \vdash e : \tau'$ via a shorter derivation, so by induction a value or takes a step.

- Preservation: One new case if typing derivation $\cdot \vdash e : \tau$ ends with subsumption. Then $\cdot \vdash e : \tau'$ via a shorter derivation, so by induction if $e \rightarrow e'$ then $\cdot \vdash e' : \tau'$. So use subsumption to derive $\cdot \vdash e : \tau$.

Hmm...

Ah, Canonical Forms

That's because Canonical Forms is where the action is:

- If $\cdot \vdash v : \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}$, then v is a record with fields $l_1, \ldots, l_n$.
- If $\cdot \vdash v : \tau_1 \rightarrow \tau_2$, then v is a function.

Have to use induction on the typing derivation (may end with many subsumptions) and induction on the subtyping derivation (e.g., "going up the derivation" only adds fields)

- Canonical Forms is typically trivial without subtyping; now it requires some work.