# CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 10— Intro to polmorphism; Subtyping

# Where are we

- Midterm next Tuesday: up through **fix** $e$, but not termination of ST$\boldsymbol{\lambda}$C. Open book and notes.
- We've used $\boldsymbol{\lambda}$ calculus to model functions and types to prevent stuck states.
- We've extended ST$\boldsymbol{\lambda}$C with primitives, pairs, sums, records, fix, etc.
- Haven't done recursive types (e.g., lists) or mutation or exceptions yet.
- But first let's be *less restrictive without affecting run-time behavior*

# Being Less Restrictive

"Will a $\boldsymbol{\lambda}$ term get stuck?" is Turing complete, 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 ST$\boldsymbol{\lambda}$C with explicit argument types ($\lambda x : \tau.\ e$) has no polymorphism:
If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$.

# Explicit Types and Non-Polymorphism

Fact: Previous fact holds for all our extensions (assuming all *binding occurrences* have explicit types)

Without explicit types, $\cdot \vdash \boldsymbol{\lambda x.\ x} : \textbf{int} \rightarrow \textbf{int}$ and $\cdot \vdash \boldsymbol{\lambda x.\ x} : (\textbf{int} \rightarrow \textbf{int}) \rightarrow (\textbf{int} \rightarrow \textbf{int})$.

But we still need "two copies" of $\boldsymbol{\lambda x.\ x}$ to use it at two types – type system is (still) preventing abstracting common parts.

From now on, assume the explicit-type version...

# 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

- Today: Subtyping, preferably without coercions
- Then: Parametric polymorphism ($\forall$) and maybe first-class ADTs ($\exists$) and recursive types ($\boldsymbol{\mu}$).
  (All use type variables ($\boldsymbol{\alpha}$).)
- 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:

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

$$\frac{e_i \rightarrow e_i'}{\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = e_n\} \rightarrow \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i', \ldots, l_n = e_n\}}$$

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

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

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

# 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*:

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

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

And our 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 $\{l_2{:}\mathbf{int}, l_1{:}\mathbf{int}\}$?

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

Example with width: Show

$\cdot \vdash \{l_1{=}7, l_2{=}8, l_3{=}9\} : \{l_2{:}\mathbf{int}, l_1{:}\mathbf{int}\}$.

There are multiple ways (subsumptions or transitivity), so it's unclear what an (efficient, sound, complete) algorithm should be. But they exist.

# 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...

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

Can use *dictionary-passing* and maybe *optimize away* (some) lookups.

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

# Depth Subtyping

With just records of ints, we miss another opportunity:

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

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

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

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

Note: With permutation subtyping could just allow depth on left-most field.

Note: Soundness of this rule depends *crucially* on fields being *immutable*. (We will get to this point.)

# 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 $\tau_4$ (including subtypes) whenever given something of type $\tau_3$ (including subtypes). A function assuming less than $\tau_3$ will do, but not one assuming more.

# Function subtyping, cont'd

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

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

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.

# 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.

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)

# A Matter of Opinion?

If subsumption makes well-typed terms get stuck, it is *wrong*.

We might allow less subsumption (for efficiency), but we shall not allow more than is sound.

But we have been discussing "subset semantics" in which $e : \tau$ and $\tau \leq \tau'$ means $e$ *is* a $\tau'$. (There are "fewer" values of type $\tau$ than of type $\tau'$, but not really.)

It is very tempting to go beyond this, but you must be very careful...

But first we need to emphasize a really nice property we had: *Types never affected run-time behavior.*

# Erasure

I.e., A program type-checks or does not. If it does, it evaluates just like in the untyped $\lambda$-calculus. More formally, we have:

- Our language with types (e.g., $\lambda x : \tau.\ e$, $\mathsf{inl}_{\tau_1+\tau_2}(e)$, etc.) and a semantics
- Our language without types (e.g., $\lambda x.\ e$, $\mathsf{inl}(e)$, etc.) and a different (but very similar) semantics
- An *erasure* metafunction from first language to second
- An equivalence theorem: Erasure commutes with evaluation.

This useful (for reasoning and efficiency) fact will be less obvious (but true) with parametric polymorphism.

# Coercion Semantics

Wouldn't it be great if...

- **int** $\leq$ **float**
- **int** $\leq$ $\{l_1$:**int**$\}$
- $\tau \leq$ **string**
- we could "overload the cast operator"

For these proposed $\tau \leq \tau'$ relationships, we need a run-time action to turn a $\tau$ into a $\tau'$. Called a coercion.

Programmers could use `float_of_int` and similar but they whine about it.

# Implementing Coercions

If coercion $C$ (e.g., `float_of_int`) "witnesses" $\tau \leq \tau'$ (e.g., **int** $\leq$ **float**), then we insert $C$ when using $\tau \leq \tau'$ with subsumption.

So our translation to the untyped semantics depends on where we use subsumption. So its really from *typing derivations* to programs.

And typing derivations aren't deterministic (uh-oh).

Example 1: Suppose **int** $\leq$ **float** and $\tau \leq$ **string**. Consider $\cdot \vdash \texttt{print\_string}(\mathbf{34})$ : **unit**.

Example 2: Suppose **int** $\leq$ $\{l_1$:**int**$\}$. Consider $\mathbf{34 == 34}$.

# Coherence

Coercions need to be *coherent*, meaning they don't have these problems. (More formally, programs are deterministic even though type checking is not—any typing derivation for $e$ translates to an equivalent program.)

You can also make (complicated) rules about where subsumption occurs and which subtyping rules take precedence.

It's a mess...

# C++

Semi-Example 3: Multiple inheritance a la C++.

```
class C2 {};
class C3 {};
class C1 : public C2, public C3 {};
class D {
  public:  int f(class C2) { return 0; }
           int f(class C3) { return 1; }
};
int main() { return D().f(C1()); }
```

Note: A compile-time error "ambiguous call"

Note: The first C++ I've written in a *long* time.

Note: Same in Java with interfaces ("reference is ambiguous")

# Where are we

- "Subset" subtyping allows "upcasts"
- "Coercive subtyping" allows casts with run-time effect
- What about "downcasts"?

  That is, should we have something like:

  `if_hastype`$(\tau, e_1)$ `then` $x.e_2$ `else` $e_3$

  (Roughly, if at run-time $e_1$ has type $\tau$ (or a subtype), then bind it to $x$ and evaluate $e_2$. Else evaluate $e_3$. Avoids having exceptions.)

# Downcasts

I can't deny downcasts exist, but here are some bad things about them:

- Types don't erase – you need to represent $\tau$ and $e_1$'s type at run-time. (Hidden data fields.)
- Breaks abstractions: Before, passing $\{l_1 = 3, l_2 = 4\}$ to a function taking $\{l_1 : \textbf{int}\}$ hid the $l_2$ field.
- Use ML-style datatypes – now programmer decides which data should have tags.
- Use parametric polymorphism – the right way to do container types (not downcasting results)