CSE505: Graduate Programming Languages

Lecture 14 — Subtyping

Dan Grossman
Winter 2012

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":

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

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

Fact: The version of STLC 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 $((0, 0), (\textbf{true}, \textbf{true}))$)

An overloaded 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.
(More precise terms: "static overloading," "dynamic dispatch," "type abstraction," and "subtyping")

Today

This lecture is about *subtyping*

- Let more terms type-check without adding any new operational behavior
  - But at end consider *coercions*
- Continue using STLC as our core model
- Complementary to type variables which we will do later
  - Parametric polymorphism (∀), a.k.a. generics
  - First-class ADTs (∃)
- Even later: OOP, dynamic dispatch, inheritance vs. subtyping

Motto: Subtyping is not a matter of opinion!

Records

We'll use records to motivate subtyping:

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

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

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

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

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

Should this typecheck?

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

Right now, it doesn't, but it won't get stuck

Suggests *width subtyping*:

$\boxed{\tau_1 \le \tau_2}$

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

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

$$\frac{\Gamma \vdash e : \tau' \quad \tau' \le \tau}{\Gamma \vdash e : \tau}\ \text{SUBSUMPTION}$$

Now it type-checks

$$\dfrac{\dfrac{\dfrac{\vdots}{\cdot, x : \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\} \vdash x.l_1 + x.l_2 : \mathsf{int}}}{\cdot \vdash \lambda x : \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}.\ x.l_1 + x.l_2 : \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\} \to \mathsf{int}} \quad \dfrac{\dfrac{\cdot \vdash 3 : \mathsf{int} \quad \cdot \vdash 4 : \mathsf{int} \quad \cdot \vdash 5 : \mathsf{int}}{\cdot \vdash \{l_1{=}3, l_2{=}4, l_3{=}5\} : \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}, l_3{:}\mathsf{int}\}} \quad \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}, l_3{:}\mathsf{int}\} \leq \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}}{\cdot \vdash \{l_1{=}3, l_2{=}4, l_3{=}5\} : \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}}}{\cdot \vdash (\lambda x : \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}.\ x.l_1 + x.l_2)\{l_1{=}3, l_2{=}4, l_3{=}5\} : \mathsf{int}}$$

Instantiation of Subsumption is highlighted (pardon formatting)

The derivation of the *subtyping fact* $\{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}, l_3{:}\mathsf{int}\} \leq \{l_1{:}\mathsf{int}, l_2{:}\mathsf{int}\}$ would continue, using rules for the $\tau_1 \leq \tau_2$ judgment

- But here we just use the one axiom we have so far

Clean division of responsibility:

- Where to use subsumption
- How to show two types are subtypes

Permutation

Does this program type-check? Does it get stuck?

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

Suggests *permutation subtyping*:

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

Example with width and permutation: 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 there is an (efficient, sound, complete) type-checking algorithm

- They sometimes exist and sometimes don't
- Here they do

Transitivity

Subtyping is always transitive, so add a rule for that:

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

Or just use the subsumption rule multiple times. Or both.

In any case, type-checking is no longer syntax-directed: There may be 0, 1, *or many* different derivations of $\Gamma \vdash e : \tau$

- And also potentially many ways to show $\tau_1 \leq \tau_2$

Hopefully we could define an algorithm and prove it "answers yes" if and only if 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

Digression continued

With width subtyping alone, the strategy is easy

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

With both, it's not easy...

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

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

So far

- A new *subtyping judgement* and a new typing rule *subsumption*
- Width, permutation, and transitivity

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

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

$$\dfrac{}{\{l_1{:}\tau_1, \ldots, l_{i-1}{:}\tau_{i-1}, l_i{:}\tau_i, \ldots, l_n{:}\tau_n\} \leq \{l_1{:}\tau_1, \ldots, l_i{:}\tau_i, l_{i-1}{:}\tau_{i-1}, \ldots, l_n{:}\tau_n\}} \qquad \dfrac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

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

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

Now: This is all much more useful if we extend subtyping so it can be used on "parts" of larger types:

- Example: Can't yet use subsumption on a record field's type
- Example: There are no supertypes yet of $\tau_1 \to \tau_2$

Depth

Does this program type-check? Does it get stuck?

$(\lambda x{:}\{l_1{:}\{l_3{:}\textbf{int}\}, l_2{:}\textbf{int}\}.\ x.l_1.l_3 + x.l_2)\{l_1{=}\{l_3{=}3, l_4{=}9\}, l_2{=}4\}$

Suggests *depth subtyping*

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

(With permutation subtyping, can just have 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)
- Homework 4 explores mutation and subtyping

Function subtyping

Given our rich subtyping on records (and/or other primitives), how do we extend it to other types, notably $\tau_1 \to \tau_2$?

For example, we'd like $\textbf{int} \to \{l_1{:}\textbf{int}, l_2{:}\textbf{int}\} \leq \textbf{int} \to \{l_1{:}\textbf{int}\}$ so we can pass a function of the subtype somewhere expecting a function of the supertype

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

For a function to have type $\tau_3 \to \tau_4$ 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. A function returning more than τ_4 but not one returning less.

Function subtyping, cont'd

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

Example: $\lambda x : \{l_1{:}\textbf{int}, l_2{:}\textbf{int}\}.\ \{l_1 = x.l_2, l_2 = x.l_1\}$
can have type $\{l_1{:}\textbf{int}, l_2{:}\textbf{int}, l_3{:}\textbf{int}\} \to \{l_1{:}\textbf{int}\}$
but *not* $\{l_1{:}\textbf{int}\} \to \{l_1{:}\textbf{int}\}$

Jargon: Function types are *contravariant* in their argument and *covariant* in their result

- Depth subtyping means immutable records are covariant in their fields

This is unintuitive enough that you, a friend, or a manager, will some day be convinced that functions can be covariant in their arguments. THIS IS ALWAYS WRONG (UNSOUND).

Summary of subtyping rules

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

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

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

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

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

Notes:

- As always, elegantly handles arbitrarily large syntax (types)
- For other types, e.g., sums or pairs, would have more rules, deciding carefully about co/contravariance of each position

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 $e \to e'$ 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 $\cdot \vdash v : \tau_1 \to \tau_2$, then v is a function

We need these for the "interesting" cases of Progress

Now 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

Note: Without subtyping, Preservation is a little "cleaner" via induction on $e \to e'$, but with subtyping it's *much* cleaner via induction on the typing derivation

- That's why we did it that way

A matter of opinion?

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

We might allow less subsumption (e.g., 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 τ'

- There are "fewer" values of type τ than of type τ', but not really

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

But first we need to emphasize a really nice property of our current setup: *Types never affect run-time behavior*

Erasure

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

1. Our language with types (e.g., $\lambda x : \tau.\ e$, $\mathbf{A}_{\tau_1+\tau_2}(e)$, etc.) and a semantics
2. Our language without types (e.g., $\lambda x.\ e$, $\mathbf{A}(e)$, etc.) and a different (but very similar) semantics
3. An *erasure* metafunction from first language to second
4. 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...

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

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

- Called a coercion

Could use `float_of_int` and similar but programmers whine about it

Implementing Coercions

If coercion C (e.g., `float_of_int`) "witnesses" $\tau \leq \tau'$ (e.g., $\mathbf{int} \leq \mathbf{float}$), then we insert C where τ is subsumed to τ'

So translation to the untyped language depends on where subsumption is used. So it's from *typing derivations* to programs.

But typing derivations aren't unique: uh-oh

Example 1:

- Suppose $\mathbf{int} \leq \mathbf{float}$ and $\tau \leq \mathbf{string}$
- Consider $\cdot \vdash \mathtt{print\_string}(34) : \mathbf{unit}$

Example 2:

- Suppose $\mathbf{int} \leq \{l_1{:}\mathbf{int}\}$
- Consider $34 == 34$, where $==$ is equality on ints or pointers

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

Alternately, can make (complicated) rules about where subsumption occurs and which subtyping rules take precedence

- Hard to understand, remember, implement correctly

It's a mess...

C++

Semi-Example: 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: Same in Java with interfaces ("reference is ambiguous")

Upcasts and Downcasts

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

That is, should we have something like:

`if_hastype(`τ`,`e_1`) then` x. e_2 `else` e_3

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

- Not hard to formalize

Downcasts

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

- Types don't erase – you need to represent τ 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, so you know it doesn't change or affect the callee

Some better alternatives:

- Use ML-style datatypes — the programmer decides which data should have tags
- Use parametric polymorphism — the right way to do container types (not downcasting results)