CSE 505: Programming Languages

Lecture 17 — Subtyping

Zach Tatlock
Fall 2013

Tradeoffs

Desirable type system properties (*desiderata*):

- *soundness* - exclude all programs that get stuck
- *completeness* - include all programs that don't get stuck
- *decidability* - effectively determine if a program has a type

Our friend Turing says we can't have it all.

We choose soundness and decidability, aim for "reasonable" completeness, but still reject valid programs.

Any benefit to an *unsound*, complete, decidable type system?

Today: *subtype polymorphism* to start adding completeness.

Next Lecture: *parametric polymorphism* to get even more.

Where shall we add completeness?

$\textbf{if true } \mathbf{1}\ (\mathbf{2}, \mathbf{3})$ does not get stuck, but we can't type it either.

Perhaps we should add this typing rule?

$$\frac{e_1 \xrightarrow{*} \textbf{true} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{if } e_1\ e_2\ e_3 : \tau}$$

Not if we want to keep decidability!

How about?

$$\frac{\Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{if true } e_2\ e_3 : \tau}$$

Sound, adds completeness, but not terribly useful.

Where shall we add *useful* completeness?

Code reuse is crucial: write code once, use it in many contexts.

Polymorphism supports code reuse and comes in several flavors:

- *ad hoc* - implementation depends on type details

  $+$ in ML vs. C vs. C++

- *parametric* - implementation independent of type details

  $$\boldsymbol{\Gamma \vdash \lambda x.\ x : \forall \alpha . \alpha \to \alpha}$$

- *subtype* - implementation assumes constrained types

```
void makeSound(Dog d) {
    d.growl();
}
...
makeSound(new Husky());
```

Where shall we add *useful* completeness?

Code reuse is crucial: write code once, use it in many contexts.

Polymorphism supports code reuse and comes in several flavors:

- *ad hoc* - implementation depends on type details

  $+$ in ML vs. C vs. C++

- *parametric* - implementation independent of type details

  $$\Gamma \vdash \lambda x.\, x : \forall \alpha \to \alpha$$

- *subtype* - implementation assumes constrained types

```
void makeSound(Dog d) {
    d.growl();
}
...
makeSound(new Husky());
```

Subtyping uses a value of type A as a different type B.

Where shall we add *useful* completeness? Subtyping.

Wait... how many types can a STLC expression have?

At most one! Currently we have **no polymorphism** :(

If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$

Let's fix that:

- add completeness by extending STLC with subtyping
- consider implications for the compiler
- also touch on coercions and downcasts

Guiding principle:

If A is a subtype of B (written $A \leq B$), then we can safely use a value of type A anywhere a value of type B is expected.

Extending STLC with Subtyping

We know the extension recipe:

1. add new syntax
2. add new semantic rules
3. add new typing rules
4. update type safety proof

Extending STLC with Subtyping

We know the extension recipe: *already half done!*

1. ~~add new syntax~~
2. ~~add new semantic rules~~
3. add new typing rules
4. update type safety proof

Where to start adding new typing rules?

First, let's focus on *records*:

- review existing rules
- consider examples of incompleteness
- add new rules to handle examples and improve completeness

Records Review

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

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

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

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

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

Should this typecheck?

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

Sure! It won't get stuck.

Suggests *width subtyping*:

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

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

Add new typing rule to take advantage of subtyping: *Subsumption*

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

Now it type-checks

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

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 $\tau_1 \leq \tau_2$. So far we only have one subtyping axiom, just use that.

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*:

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

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

No longer obvious, efficient, sound, complete type-checking algo:

- sometimes such algorithms exist and sometimes they don't
- in this case, we have them

Reflexive Transitive Closure

The subtyping principle implies reflexivity and transtivity:

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

Could get transitivity w/ multiple subsumptions anyway.

Have we lost anything while gaining all these rules?

Type-checking no longer *syntax-directed*:

- may be 0, 1, *or many* distinct derivations of $\Gamma \vdash e : \tau$
- many potential ways to show $\tau_1 \leq \tau_2$

Still decidable? Need algorithm checking that labels always a subset of what's required, must prove it "answers yes" *iff* there exists a derivation.

Still efficient?

Implementation Efficiency

Given semantics, width and permutation subtyping totally reasonable.

How do they impact the lives of our dear friend, the compiler writer?

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.

In general:

> *If some language restriction seems odd, ask yourself: what useful invariant does limiting expressiveness provide the compiler?*

Implementation Efficiency

Changes to implement width subtyping alone? *None.*

Changes to implement permutation subtyping alone? *Sort fields.*

Changes to implement both? Not so easy...

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

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

Getting some sweet completeness.

Added new *subtyping judgement*:

- width, permutation, reflexive transitive closure

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

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

Added new typing rule, *subsumption*, to use subtyping:

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

Squeeze out more completeness:

- Extend subtyping to "parts" of larger types
- Example: Can't yet use subsumption on a record field's type
- Example: Don't yet have supertypes of $\tau_1 \to \tau_2$

Depth

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

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

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)

Function Subtyping

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

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

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

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. A function guaranteeing more than τ_4 but not one guaranteeing less.

Function Subtyping

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

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

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

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

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

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

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

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

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 $e \rightarrow e'$ 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 $\cdot \vdash v : \tau_1 \to \tau_2$, 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., $\lambda x : \tau.\ e$, $\mathsf{A}_{\tau_1+\tau_2}(e)$, etc.) and a semantics
2. Our language without types (e.g., $\lambda x.\ e$, $\mathsf{A}(e)$, 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...

- **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 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., **int** $\leq$ **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 **int** $\leq$ **float** and $\tau \leq$ **string**
- Consider $\cdot \vdash \texttt{print\_string}(34) : \textbf{unit}$

Example 2:

- Suppose **int** $\leq \{l_1{:}\textbf{int}\}$
- 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...

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 $\{l_1 : \mathbf{int}\}$ 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)