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., $\Gamma \vdash \lambda x. x : \forall \alpha. \alpha \rightarrow \alpha$ or with explicit types: $\Gamma \vdash \lambda x : \alpha. x : \forall \alpha. \alpha \rightarrow \alpha$
  (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”)

Being Less Restrictive

“Will a $\lambda$ 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 “admissible”:

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

We’ll study ways to give one term many types (“polymorphism”)

Fact: The version of STLC with explicit argument types ($Ax : \tau. e$) 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: ($\lambda f. (f 0, f \text{ true}))((\lambda x. (x, x)) (0, 0), (\text{true, true}))$

Records

We’ll use records to motivate subtyping:

$$e ::= \ldots \mid \{l_1 = e_1, \ldots, l_n = e_n\} | e.e$$
$$\tau ::= \ldots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$
$$v ::= \ldots \mid \{l_1 = v_1, \ldots, l_n = v_n\}, l_i \rightarrow v_i$$

$$\frac{e_i \rightarrow e_i'}{e.e \rightarrow e.e'}$$

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

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

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

Should this typecheck?

$$\frac{(\lambda x : \{l_1 : \text{int}, l_2 : \text{int}\}. x.l_1 + x.l_2)\{l_1 = 3, l_2 = 4, l_3 = 5\}}{\text{Right now, it doesn’t, but it won’t get stuck}}$$

Suggests width subtyping:

$$\frac{\tau_1 \leq \tau_2}{\{l_1 : \tau_1, \ldots, l_n : \tau_n, l : \tau\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

And one one new type-checking rule: Subsumption

$$\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}$$
Now it type-checks

: x : (l_1:int, l_2:int) = x.l_1 + x.l_2 : int

· ⊢ 3 : int
· ⊢ 4 : int
· ⊢ 5 : int

: (l_1=3, l_2=4, l_3=5) : (l_1:int, l_2:int, l_3:int)

Γ ⊢ \lambda x : (l_1:int, l_2:int) . x.l_1 + x.l_2 : (l_1:int, l_2:int)

Instantiation of Subsumption is highlighted (pardon formatting)

The derivation of the subtyping fact
\{l_1:int, l_2:int\} \leq \{l_1:int, l_2: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

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

\[
\begin{align*}
\tau_1 & \leq \tau_2 \\
\tau_2 & \leq \tau_3 \\
\Rightarrow & \\
\tau_1 & \leq \tau_3
\end{align*}
\]

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 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…

\[
\begin{align*}
f_1 : \{l_1 : \text{int}\} \rightarrow \text{int} \\
f_2 : \{l_2 : \text{int}\} \rightarrow \text{int} \\
x_1 = \{l_1 = 0, l_2 = 0\} \\
x_2 = \{l_2 = 0, l_3 = 0\} \\
f_1(x_1) \\
f_2(x_1) \\
f_2(x_2)
\end{align*}
\]

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

Permuation

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

\[
(\lambda x : \{l_1:\text{int}, l_2:\text{int}\} . x.l_1 + x.l_2) \{l_2=3; l_1=4\}
\]

Suggests permutation subtyping:

\[
\{l_1:\tau_1, ..., l_{i-1}:\tau_{i-1}, l_i:\tau_i, ..., l_n:\tau_n\} \leq \{l_1:\tau_1, ..., l_{i-1}:\tau_{i-1}, ..., l_i:\tau_i\}
\]

Example with width and permutation: Show

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

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

So far

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

\[
\begin{align*}
\tau_1 & \leq \tau_2 \\
\{l_1:\tau_1, ..., l_n:\tau_n, l:\tau\} & \leq \{l_1:\tau_1, ..., l_n:\tau_n\}
\end{align*}
\]

\[
\begin{align*}
\{l_1:\tau_1, ..., l_{i-1}:\tau_{i-1}, l_i:\tau_i, ..., l_n:\tau_n\} & \leq \{l_1:\tau_1, ..., l_{i-1}:\tau_{i-1}, ..., l_i:\tau_i\} \\
\{\tau_1 \leq \tau_2, \tau_2 \leq \tau_3\} & \leq \{\tau_1 \leq \tau_2, \tau_2 \leq \tau_3\}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Gamma \vdash e : \tau' \\
\tau' \leq \tau
\end{align*}
\]

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 $\tau_1 \rightarrow \tau_2$
Depth

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

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

Suggests depth subtyping

\[ \tau_1 \leq \tau'_1 \leq \cdots \leq \tau_i \leq \cdots \leq \tau_n \]

(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 will explore mutation and subtyping

Function subtyping, cont’d

\[ \tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4 \quad \text{Also want:} \quad \tau \leq \tau \]

Example: \( \lambda x : \{l_1: \text{int}, l_2: \text{int}\}. \{l_1 = x.l_2, l_2 = x.l_1\} \) can have type \( \{l_1: \text{int}, l_2: \text{int}, l_3: \text{int}\} \implies \{l_1: \text{int}\} \) but not \( \{l_1: \text{int}\} \implies \{l_1: \text{int}\} \)

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). Remember (?) that a PL professor JUMPED UP AND DOWN about this.

Summary of subtyping rules

\[ \tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \quad \text{Also want:} \quad \tau \leq \tau \]

\[ \{l_1: \tau_1, \ldots, l_n: \tau_n, l_i: \tau\} \leq \{l_1: \tau_1, \ldots, l_n: \tau_n\} \]

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

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 \( \vdash e : \tau \) ends with subsumption. Then \( \vdash e : \tau' \) via a shorter derivation, so by induction a value or takes a step.
- Preservation: One new case if typing derivation \( \vdash e : \tau \) ends with subsumption. Then \( \vdash e : \tau' \) via a shorter derivation, so by induction if \( e \to e' \) then \( \vdash e' : \tau' \). So use subsumption to derive \( \vdash e' : \tau \).

Hmm....

Ah, Canonical Forms

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

- If \( \vdash v : \{l_1: \tau_1, \ldots, l_n: \tau_n\} \), then \( v \) is a record with fields \( l_1, \ldots, l_n \)
- If \( \vdash v : \tau_1 \to \tau_2 \), 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 \( \tau' \)

There are "fewer" values of type \( \tau \) than of type \( \tau' \), 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 \( \lambda \)-calculus. More formally, we have:

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

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

Coercion Semantics

Wouldn’t it be great if...

\( \text{int} \leq \text{float} \)
\( \text{int} \leq \{ l_1 : \text{int} \} \)
\( \tau \leq \text{string} \)
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

Could use \text{float} \_o\text{f} \_i\text{nt} and similar but programmers whine about it

Implementing Coercions

If coercion \( C \) (e.g., \text{float} \_o\text{f} \_i\text{nt}) “witnesses” \( \tau \leq \tau' \) (e.g., \( \text{int} \leq \text{float} \)), then we insert \( C \) where \( \tau \) is subsumed to \( \tau' \)

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:

\( \text{Suppose } \text{int} \leq \text{float} \) and \( \tau \leq \text{string} \)

Consider \( \cdot \vdash \text{print} \cdot _{\text{string}}(34) : \text{unit} \)

Example 2:

\( \text{Suppose } \text{int} \leq \{ l_1 : \text{int} \} \)

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++

\[
\begin{array}{l}
\text{class C2 \{} \\
\text{class C3 \{} \\
\text{class C1 : public C2, public C3 \{} \\
\text{class D \{} \\
\text{public: int f(class C2) \{ return 0; \}} \\
\text{int f(class C3) \{ return 1; \}} \\
\}\; \\
\text{int main() \{ return D().f(C1()); \}} \\
\end{array}
\]

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:

```c
if hassubtype(τ, e1) then x. e2 else e3
```

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

I 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 \{\( l_1 : \text{int} \}\) 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)