Programming Language Theory

Subtyping
Being less restrictive

Recall: “Will a $\lambda$ term get stuck?” is undecidable.

- A sound, decidable type system can *always* be made less restrictive.

An “uninteresting” rule that is sound but not “admissable” (implied by existing rules):

\[
\frac{\Gamma \vdash e_1 : \tau}{\Gamma \vdash \text{if true then } e_1 \text{ else } e_2 : \tau}
\]

Study “interesting” ways to give one term many types (“polymorphism”).

Fact: The STLC with explicit arg 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” are ill-typed:

- Ex: $(\lambda f. (f \ 0, f \ \text{true}))(\lambda x. (x, x))$ (evaluates to $((0, 0), (\text{true}, \text{true}))$).
“Polymorphism”: An Overloaded PL Word

“Polymorphism” means many forms (literally and figuratively) . . .

▶ Ad hoc polymorphism:
  ▶ $e_1 + e_2$ in SML, C, Java, C++, etc.

▶ Ad hoc polymorphism, cont’d:
  ▶ Choose the $+$ operation based on the run-time types of $e_1$ and $e_2$
    (which may be different)

▶ Parametric polymorphism:
  ▶ let fun dup x => (x, x) in (dup 0, dup true) end
    is legal SML because $\text{dup}$ has type $\mathit{\alpha} \rightarrow \mathit{\alpha} \times \mathit{\alpha}$

▶ Subtype polymorphism:
  ▶ $\text{new Vector().add(new C())}$
    is legal Java because $\text{new C()}$ has types Object and $\text{C}$

... and nothing.

(Better terms: “static overloading”, “dynamic dispatch”, “type abstraction”, and “subtyping”.)
Plan

Begin studying *subtyping*:

- A mechanism to let more expressions be well-typed, without (necessarily) adding any new operational behavior.
- Will consider *coercions* towards end of lecture.

Continue to use STLC (w/ Extensions) as core model.

Much later(?):

- Dynamic-dispatch, inheritance vs. subtyping, etc.
- (Concepts in OO programming.)

Motto: Subtyping is not a matter of opinion!
Records

We’ll use records to motivate subtyping:

\[
\begin{align*}
e & ::= \cdots | \{l_1 = e_1; \cdots ; l_n = e_n\} | e.l \\
v & ::= \cdots | \{l_1 = v_1; \cdots ; l_n = v_n\} \\
\tau & ::= \cdots | \{l_1: \tau_1; \cdots ; l_n: \tau_n\}
\end{align*}
\]

\[
\frac{e_i \rightarrow_{\text{cbv}} e'_i}{\{l_1 = v_1; \cdots ; l_{i-1} = v_{i-1}; l_i = e_i; \cdots ; l_n = e_n\} \rightarrow_{\text{cbv}} \{l_1 = v_1; \cdots ; l_{i-1} = v_{i-1}; l_i = e'_i; \cdots ; l_n = e_n\}}
\]

\[
\frac{e \rightarrow_{\text{cbv}} e'}{e.l \rightarrow_{\text{cbv}} e'.l}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \cdots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1; \cdots ; l_n = e_n\} : \{l_1: \tau_1; \cdots ; l_n: \tau_n\}}
\]

Fields in a record or record type should be distinct. Fields do not \(\alpha\)-convert.
An example

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

\[
(\lambda x : \{ l_1 : \text{int}; l_2 : \text{int} \}. \ x.l_1 + x.l_2) \ \{ l_1 = 3; l_2 = 4; l_3 = 5 \}
\]
An example

Does this program type check? No. Does this program get stuck? No.

\[(\lambda x : \{l_1:int; l_2:int\}. x.l_1 + x.l_2) \{l_1=3; l_2=4; l_3=5\}\]
An example

Does this program type check? No. Does this program get stuck? No.

\[(\lambda x : \{ l_1 : \text{int}; l_2 : \text{int} \}. \ x. l_1 + x. l_2) \ \{ l_1 = 3; l_2 = 4; l_3 = 5 \}\]

Suggests \textit{width subtyping}:

\[\tau' \leq \tau\]

\[\{ l_1 : \tau_1; \ldots; l_n : \tau_n; l : \tau \} \leq \{ l_1 : \tau_1; \ldots; l_n : \tau_n \}\]

And one new type-checking rule:

\[
\text{\textbf{Subsumption}}
\]

\[\begin{array}{c}
\Gamma \vdash e : \tau' \\
\tau' \leq \tau
\end{array}
\]

\[\Rightarrow \Gamma \vdash e : \tau\]
Now Program is Well-Typed

\[
\frac{D_1 \quad D_2}{\cdot \vdash (\lambda x : \{l_1 : \text{int}; l_2 : \text{int}\}. x.l_1 + x.l_2) \{l_1 = 3; l_2 = 4; l_3 = 5\} : \text{int}}
\]

\[
\vdash x : \{l_1 : \text{int}; l_2 : \text{int}\} \vdash x.l_1 + x.l_2 : \text{int}
\]

\[
\vdash \lambda x : \{l_1 : \text{int}; l_2 : \text{int}\}. x.l_1 + x.l_2 : \{l_1 : \text{int}; l_2 : \text{int}\} \rightarrow \text{int}
\]

\[
\frac{\cdot \vdash \{l_1 = 3; l_2 = 4; l_3 = 5\} : \{l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int}\} \quad \{l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int}\} \leq \{l_1 : \text{int}; l_2 : \text{int}\}}{D_2 = \cdot \vdash \{l_1 = 3; l_2 = 4; l_3 = 5\} : \{l_1 : \text{int}; l_2 : \text{int}\}}
\]

The derivation of the subtyping fact uses rules for the $\tau' \leq \tau$ judgement.

- $\{l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int}\} \leq \{l_1 : \text{int}; l_2 : \text{int}\}$ only requires the width subtyping axiom

Clean division of responsibility:

- Where to use subsumption.
- How to show two types are subtypes.
Another example

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

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

Does this program type check? No. Does this program get stuck? No.

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

Does this program type check? No. Does this program get stuck? No.

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

Suggests permutation subtyping:

$$\{l_1:\tau_1; \cdots ; l_i:\tau_i; l_{i+1}:\tau_{i+1}; \cdots ; l_n:\tau_n\} \leq \{l_1:\tau_1; \cdots ; l_{i+1}:\tau_{i+1}; l_i:\tau_i; \cdots ; l_n:\tau_n\}$$

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 that there is an (efficient, sound, complete) algorithm. They sometimes exist and sometimes don’t. Here they do.
Reflexivity and Transitivity

Subtyping is always reflexive. There’s a rule for that:

\[
\tau \leq \tau
\]

Subtyping is always transitive. There’s a rule for that:

\[
\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \quad \Rightarrow \quad \tau_1 \leq \tau_3
\]
Reflexivity and Transitivity

Subtyping is always reflexive. There’s a rule for that:

\[
\tau \leq \tau
\]

Subtyping is always transitive. There’s a rule for that:

\[
\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}
\]

Or just use the subsumption rule multiple times.
Reflexivity and Transitivity

Subtyping is always reflexive. There’s a rule for that:

\[ \tau \leq \tau \]

Subtyping is always transitive. There’s a rule for that:

\[ \tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \]
\[ \tau_1 \leq \tau_3 \]

Or just use the subsumption rule multiple times.
Or both.
Subtyping, so far

\[ \frac{\tau \leq \tau'}{\Gamma \vdash e : \tau'} \]

\[ \frac{\tau' \leq \tau}{\Gamma \vdash e : \tau} \]

Subsumption

\[ \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \]

\[ \frac{\{l_1: \tau_1; \ldots; l_n: \tau_n; l: \tau\} \leq \{l_1: \tau_1; \ldots; l_n: \tau_n\}}{\{l_1: \tau_1; \ldots; l_i: \tau_i; l_i+1: \tau_{i+1}; \ldots; l_n: \tau_n\} \leq \{l_1: \tau_1; \ldots; l_i+1: \tau_{i+1}; l_i: \tau_i; \ldots; l_n: \tau_n\}} \]

Type checking is no longer syntax-directed!

- May be 0, 1, or many ways to show \( \Gamma \vdash e : \tau \).
- May be 0, 1, or many ways to show \( \tau' \leq \tau \).

Hopefully, we could define an algorithm and prove it finds some derivation iff there exists a derivation.
Efficiency Digression

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.

If you do not know techniques for implementing high-level languages, then it may make restrictions seem odd.
Efficiency Digression (continued)

With width subtyping alone, the compilation strategy is easy.
▶ Can be used (and abused) in C.

With permutation subtyping alone, still easy.
▶ Have to “alphabetize” fields.
▶ Used in SML compilers.

With both, it’s not easy . . .

\[
f_1 : \{ l_1 : \text{int} \} \rightarrow \text{int} \quad f_2 : \{ l_2 : \text{int} \} \rightarrow \text{int}
\]
\[
x_1 = \{ l_1=0; l_2=0 \} \quad x_2 = \{ l_2=0; l_3=0 \}
\]
\[
f_1(x_1) \quad f_2(x_1) \quad f_2(x_2)
\]

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.
Subtyping, so far

- A new subtyping judgement, with width, permutation, reflexivity, and transitivity rules.
- A new typing rule, providing subsumption.

\[
\tau \leq \tau \\
\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3
\]

\[
\Gamma \vdash e : \tau' \quad \tau' \leq \tau
\]

\[
\Gamma \vdash e : \tau
\]

\[
\{l_1:\tau_1; \ldots ; l_n:\tau_n ; l:\tau\} \leq \{l_1:\tau_1; \ldots ; l_n:\tau_n\}
\]

If we extend subtyping so that it can be used on “parts” of larger types, then it will be much more useful:

- Example: Can’t yet use subsumption on a record-field’s types.
- Example: Aren’t yet any sub- or super-types of \(\tau_a \rightarrow \tau_r\).
Yet another example

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

\((\lambda y : \{ l_1 : \{ l_2 : \text{int}; l_3 : \text{int} \} \}). (\lambda x : \{ l_1 : \{ l_3 : \text{int} \} \}. x. l_1. l_3 + x. l_1. l_3) \ y) \ \{ l_1 = \{ l_3 = 3; l_4 = 9 \} \} \)
Yet another example

Does this program type check? No. Does this program get stuck? No.

\[(\lambda y : \{l_1:\{l_2:\text{int}; l_3:\text{int}\}\}. \ (\lambda x : \{l_1:\{l_3:\text{int}\}\}. x.l_1.l_3 + x.l_1.l_3) \ y) \ l_1=\{l_3 = 3; l_4 = 9\}\]
Yet another example

Does this program type check? No. Does this program get stuck? No.

\[
(\lambda y : \{ l_1 : \{ l_2 : \text{int}; l_3 : \text{int} \} \}. \\
(\lambda x : \{ l_1 : \{ l_3 : \text{int} \} \}. x. l_1. l_3 + x. l_1. l_3) \\ y) \{ l_1 = \{ l_3 = 3; l_4 = 9 \} \}
\]

Suggests depth subtyping

\[
\tau_i' \leq \tau_i \\
\{ l_1 : \tau_1; \ldots \}; l_i : \tau_i'; \ldots ; l_n : \tau_n \} \leq \{ l_1 : \tau_1; \ldots ; l_i : \tau_i; \ldots ; l_n : \tau_n \}
\]

Note: with permutation subtyping, could just allow depth on first 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).
- Next homework will explore mutation and subtyping in more detail.
Function subtyping

Given our rich subtyping on records, how do we extend it to other types, namely $\tau_1 \rightarrow \tau_2$?

For example, we’d like \texttt{int} $\rightarrow \{l_1:\text{int}; l_2:\text{int}\}$ $\leq$ \texttt{int} $\rightarrow \{l_1:\text{int}\}$ so that we can pass a function of the subtype somewhere expecting a function of the supertype.
Function subtyping

Given our rich subtyping on records, how do we extend it to other types, namely \( \tau_1 \to \tau_2 \)?

For example, we’d like \( \text{int} \to \{ l_1: \text{int}; l_2: \text{int} \} \leq \text{int} \to \{ l_1: \text{int} \} \) so that we can pass a function of the subtype somewhere expecting a function of the supertype.

\[
\frac{???
}{\tau'_a \to \tau'_r \leq \tau_a \to \tau_r}
\]
Function subtyping (continued)

Example: \( \lambda x : \{ l_1 : \text{int}; l_2 : \text{int} \}. \{ l_1 = x \cdot l_2; l_2 = x \cdot l_1 \} \)

- “Naturally” has type \( \{ l_1 : \text{int}; l_2 : \text{int} \} \rightarrow \{ l_1 : \text{int}; l_2 : \text{int} \} \)
- Can have type \( \{ l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int} \} \rightarrow \{ l_1 : \text{int} \} \) (Why?)
- But can not have type \( \{ l_1 : \text{int} \} \rightarrow \{ l_1 : \text{int} \} \) (Why?)
- And can not have type \( \{ l_1 : \text{int}; l_2 : \text{int} \} \rightarrow \{ l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int} \} \) (Why?)
Function subtyping (continued)

Example: \( \lambda x : \{ l_1 : \text{int}; l_2 : \text{int} \}. \{ l_1 = x \cdot l_2; l_2 = x \cdot l_1 \} \)

- “Naturally” has type \( \{ l_1 : \text{int}; l_2 : \text{int} \} \rightarrow \{ l_1 : \text{int}; l_2 : \text{int} \} \)
- Can have type \( \{ l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int} \} \rightarrow \{ l_1 : \text{int} \} \) (Why?)
- But can not have type \( \{ l_1 : \text{int} \} \rightarrow \{ l_1 : \text{int} \} \) (Why?)
- And can not have type \( \{ l_1 : \text{int}; l_2 : \text{int} \} \rightarrow \{ l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int} \} \) (Why?)
Function subtyping (continued)

Example: \( \lambda x : \{ l_1 : \text{int}; l_2 : \text{int} \}. \{ l_1 = x.*l_2; l_2 = x.*l_1 \} \)

- “Naturally” has type \( \{ l_1 : \text{int}; l_2 : \text{int} \} \rightarrow \{ l_1 : \text{int}; l_2 : \text{int} \} \)
- Can have type \( \{ l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int} \} \rightarrow \{ l_1 : \text{int} \} \) (Why?)
- But can not have type \( \{ l_1 : \text{int} \} \rightarrow \{ l_1 : \text{int} \} \) (Why?)
- And can not have type \( \{ l_1 : \text{int}; l_2 : \text{int} \} \rightarrow \{ l_1 : \text{int}; l_2 : \text{int}; l_3 : \text{int} \} \) (Why?)

Therefore:

\[
\frac{\tau_a \leq \tau_a'}{\tau_a' \rightarrow \tau_r} \quad \frac{\tau_r' \leq \tau_r}{\tau_a \rightarrow \tau_r' \leq \tau_r}
\]

We say function types are

- contravariant in their argument
- covariant in their result
- (and don’t let anybody tell you otherwise)

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

\[ \Gamma \vdash e : \tau \]

\[ \Gamma \vdash e : \tau' \quad \tau' \leq \tau \]

\[ \Gamma \vdash e : \tau \]

\[ \tau' \leq \tau \]

\[ \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \quad \frac{\tau_a \leq \tau_a' \quad \tau_r' \leq \tau_r}{\tau_a' \rightarrow \tau_r' \leq \tau_a \rightarrow \tau_r} \]

\[ \{ l_1 : \tau_1; \ldots; l_n : \tau_n; l : \tau \} \leq \{ l_1 : \tau_1; \ldots; l_n : \tau_n \} \]

\[ \{ l_1 : \tau_1; \ldots; l_i : \tau_i; l_{i+1} : \tau_{i+1}; \ldots; l_n : \tau_n \} \]

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

\[ \tau_i' \leq \tau_i \]

\[ \{ l_1 : \tau_1; \ldots; l_i : \tau'_i; \ldots; l_n : \tau_n \} \leq \{ l_1 : \tau_1; \ldots; l_i : \tau_i; \ldots; l_n : \tau_n \} \]
Summary of subtyping additions

More rules for other types:

\[ \tau' \leq \tau \]

\[ \begin{align*}
\tau_1 \leq \tau'_1 & \quad \tau_2 \leq \tau'_2 \\
\tau_1 \ast \tau_2 \leq \tau'_1 \ast \tau'_2 & \quad \tau_1 \leq \tau'_1 & \quad \tau_2 \leq \tau'_2 \\
\tau_1 + \tau_2 \leq \tau'_1 + \tau'_2 & 
\end{align*} \]

Both pairs and sums are \textit{covariant} in their component types.
Maintaining soundness

Preservation and Progress still “work” in the presence of subsumption.

- In theory, any mistakes/bugs in the subtyping rules should be caught when trying to prove soundness!

Things seem too easy . . .
Lemma (Progress):
If $\cdot \vdash e : \tau$, then either $e$ is a value
or there exists an $e'$ such that $e \rightarrow_{cbv} e'$.

Proof:
By induction on (the derivation of) $\cdot \vdash e : \tau$.
One new case: The derivation of $\cdot \vdash e : \tau$ ends with subsumption.
Therefore, $\cdot \vdash e : \tau^\dagger$ and $\tau^\dagger \leq \tau$.
By IH applied to $\cdot \vdash e : \tau^\dagger$, we have $e$ is a value or takes a step.
Maintaining soundness: Preservation

Lemma (Preservation):
If $\cdot \vdash e : \tau$ and $e \rightarrow_{cbv} e'$, then $\cdot \vdash e' : \tau$.

Proof:
By induction on (the derivation of) $\cdot \vdash e : \tau$.
One new case: The derivation of $\cdot \vdash e : \tau$ ends with subsumption.
Therefore, $\cdot \vdash e : \tau^\dagger$ and $\tau^\dagger \leq \tau$.
By IH applied to $\cdot \vdash e : \tau^\dagger$ w/ $e \rightarrow_{cbv} e'$, we have $\cdot \vdash e' : \tau^\dagger$.
Use subsumption with $\cdot \vdash e' : \tau^\dagger$ and $\tau^\dagger \leq \tau$ to derive $\cdot \vdash e' : \tau$. 
Maintaining soundness: Canonical Forms

Things were easy with Progress and Preservation, 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 $\vdash v : \tau_a \rightarrow \tau_r$, then $v$ is a function.

Proof is now by induction on typing derivation

- (may end with many subsumptions)
and induction on the subtyping derivation

- (“going up the subtyping derivation” only adds fields)

Note: CF is typically trivial without subtyping; now it requires some work.
Subtyping: a matter of opinion?

If subtyping makes well-typed terms get stuck, then it is \textit{wrong}!

We might allow less subtyping (for efficiency), but we \textit{never} allow more subtyping than is sound.

We have been discussing “subset semantics”:

- $\cdot \vdash e : \tau'$ and $\tau' \leq \tau$ means $e$ “is” a $\tau$.
- There are “fewer” values of type $\tau'$ than of type $\tau$.
- The set of values of type $\tau'$ is a \textit{subset} of the set of values of type $\tau$.

Very tempting to go beyond this interpretation (and some languages do so), but one must be very careful . . .

One nice property of our current setup:

- \textit{Types never affected run-time behavior}. 
Erasure

Types never affected run-time behavior.

▶ A program is well-typed or it is not.
▶ A well-typed program evaluates just like in the untyped LC.

More formally, we have:

▶ Our language with types (e.g., $\lambda x : \tau. \ e$, $L_{\tau_1 + \tau_2}(e)$, etc.) and an operational semantics
▶ Our language without types (e.g., $\lambda x. \ e$, $L(e)$, etc.) and a different (but very similar) operational semantics
▶ An erasure metafunction from first language to second: $E[\cdot]$
▶ An equivalence theorem: Erasure commutes with evaluation.

$$e \rightarrow_{\text{typed}} e' \quad \text{iff} \quad E[e] \rightarrow_{\text{untyped}} E[e']$$

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

But, wouldn’t it be great if . . .

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

For each of 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 \text{intToFloat} and \text{toString} and similar (but they whine about it).
Implementing Coercions

If coercion \( C \) (e.g., \texttt{intToFloat}) "witnesses" \( \tau' \leq \tau \) (e.g., \texttt{int} \leq \texttt{float}), then we insert \( C \) when using \( \tau' \leq \tau \) with subsumption.

Translation to the untyped lang. depends on where subsumption is used. So, translation is really from \textit{typing derivations} to programs.

And typing derivations aren’t unique (problem?!?).
Implementing Coercions

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

Translation to the untyped lang. depends on where subsumption is used. So, translation is really from *typing derivations* to programs.

And typing derivations aren’t unique (problem?!?).

Example 1

▶ Suppose `int \leq float` and $\tau \leq \text{string}$.
▶ Consider $\cdot \vdash \text{print_string}(34) : \text{unit}$.

Example 2

▶ Suppose `int \leq \{l_1: \text{int}\}`.
▶ Consider $34 == 34$ (where `==` is bit-equality on integers 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.)

- Hard to verify, if coercions are arbitrary code.

Alternatively, try to eliminate incoherence with (complicated) rules about where subsumption occurs and which subtyping rules take precedence.

- Hard to understand and predict.

It’s a mess . . .

- Which probably means its wrong . . .
Incoherence in C++, Java

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")
Subtyping: Upcasts and Downcasts

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

Roughly, if at run-time `e` has type `τ` (or a subtype), then bind it to `x` and evaluate `e_τ`. Else evaluate `e_f`.

(Avoids having exceptions.)

How would you write the type system rule?

**Hint:** Why do we need `x` bound in `e_τ`?
Subtyping: 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:

  ```
  typecase e of τ(x) => e_t | otherwise => e_f
  ```

  Roughly, if at run-time `e` has type `τ` (or a subtype), then bind it to `x` and evaluate `e_t`. Else evaluate `e_f`. (Avoids having exceptions.)

  How would you write the type system rule?
  - Hint: Why do we need `x` bound in `e_t`?

  How would you write the operational semantics rule?
Subtyping: Downcasts

Hard to deny that downcasts exist.

But (like coercive subtyping), some bad things:

- Types don’t erase:
  - Need to represent $\tau$ and $e$’s type at run-time.
  - (Hidden data fields.)

- Breaks abstractions:
  - Without downcasts, passing $\{l_1 = 3, l_2 = 4\}$ to a function taking $\{l_1 : \text{int}\}$ hid the $l_2$ field.
  - No way for function to access the $l_2$ field.

Some better alternatives:

- Use ML-style datatypes:
  - Programmer decides which data should have tags.

- Use parametric polymorphism:
  - The right way to do container types (not downcasting results).