Where shall we add completeness?

if true 1 (2, 3) does not get stuck, but we can’t type it either.

Perhaps we should add this typing rule?

\[
\frac{e_1 \rightarrow \text{true}}{\Gamma \vdash \text{if } e_1 \; e_2 \; e_3 : \tau}
\]

Not if we want to keep decidability!

How about?

\[
\frac{\Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } \text{true} \; e_2 \; e_3 : \tau}
\]

Sound, adds completeness, but not terribly useful.

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.

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
  \[
  \Gamma \vdash \lambda x. \; x : \forall \alpha. \alpha ightarrow \alpha
  \]

- **subtype** - implementation assumes constrained types
  void makeSound(Dog d) {
  d.growl();
  }
  ... makeSound(new Husky());

Subtyping uses a value of type \( \tau \) as a different type \( \tau' \).
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 $B$.

**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

**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.

### 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

**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

\[
e ::= ··· | \{ l_1 = e_1, \ldots, l_n = e_n \} \mid e.l
\]

\[
\tau ::= ··· | \{ l_1 : \tau_1, \ldots, l_n : \tau_n \}
\]

\[
v ::= ··· | \{ l_1 = v_1, \ldots, l_n = v_n \}
\]

\[
\Gamma \vdash e : \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \quad 1 \leq i \leq n
\]

\[
\Gamma \vdash e.l_i : \tau_i
\]

Now it type-checks

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

Sure! It won’t get stuck.

Suggests \textit{width subtyping}:

\[
\tau_1 \leq \tau_2
\]

Add new typing rule to take advantage of subtyping: \textit{Subsumption}

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

Instantiation of Subsumption is highlighted (pardon formatting)

The derivation of the subtyping fact

\[
\{ l_1 : \text{int}, l_2 : \text{int}, l_3 : \text{int} \} \leq \{ l_1 : \text{int}, l_2 : \text{int} \}
\]

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

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

Suggests \textit{permutation subtyping}:

\[
\begin{align*}
\{ l_1 : \tau_1, \ldots, l_i-1 : \tau_i-1, l_i : \tau_i, \ldots, l_n : \tau_n \} & \leq \\
\{ l_1 : \tau_1, \ldots, l_i-1 : \tau_i, l_i-1 : \tau_i-1, \ldots, l_n : \tau_n \}
\end{align*}
\]

Example with width and permutation. Show:

\[
\vdash \{ l_1=7, l_2=8, l_3=9 \} : \{ l_2 : \text{int}, l_1 : \text{int} \}
\]

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 transitivity:

\[ \tau \leq \tau \]

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

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?

Getting more completeness.

Added new subtyping judgement:

- width, permutation, reflexive transitive closure

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

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

Added new typing rule, subsumption, to use subtyping:

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

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

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 \( \tau_1 \rightarrow \tau_2 \)
### Depth

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

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

Suggests depth subtyping

\[\tau_i \leq \tau'_i\]

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

(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 \(\text{int} \rightarrow \{l_1:\text{int},l_2:}\text{int}\} \leq \text{int} \rightarrow \{l_1:}\text{int}\) so we can pass a function of the subtype somewhere expecting a function of the supertype

\[\text{???}\]

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. A function guaranteeing more than \(\tau_4\) but not one guaranteeing less.

### Summary of subtyping rules

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

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

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

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).

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 $e \rightarrow e'$ 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 $\vdash \cdot v : \{l_1:\tau_1, \ldots, l_n:\tau_n\}$, then $v$ is a record with fields $l_1, \ldots, l_n$

▶ If $\vdash \cdot v : \tau_1 \rightarrow \tau_2$, 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 $e \rightarrow e'$, 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., $\lambda x : \tau$. $e$, $A_{\tau_1 + \tau_2}(e)$, etc.) and a semantics

2. Our language without types (e.g., $\lambda x$. $e$, $A(e)$, 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` ≤ `float`
- `int` ≤ `{l₁:int}`
- `τ` ≤ `string`
- we could “overload the cast operator”

For these proposed `τ` ≤ `τ'` 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” `τ` ≤ `τ'` (e.g., `int` ≤ `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` ≤ `float` and `τ` ≤ `string`
- Consider ▶ Suppose `int` ≤ `{l₁:int}`
- Consider `print.string(34) : unit`

Example 2:

- Suppose `int` ≤ `{l₁: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:

```plaintext
if_hastype(τ,e₁) then x. e₂ else e₃
```

Roughly, if at run-time `e₁` has type `τ` (or a subtype), then bind it to `x` and evaluate `e₂`. Else evaluate `e₃`. 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 $\tau$ 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 $\{l_1 : \text{int}\}$ 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)