A Matter of Opinion?

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

We might allow less subsumption (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.)

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

But first we need to emphasize a really nice property we had: Types never affected run-time behavior.
Erasure

I.e., A program type-checks or does not. If it does, it evaluates just like in the untyped $\lambda$-calculus.

More formally, we have:

- Our language with types (e.g., $\lambda x : \tau. e$, $A_{\tau_1 + \tau_2}(e)$, etc.) and a semantics

- Our language without types (e.g., $\lambda x. e$, $A(e)$, etc.) and a different (but very similar) semantics

- An erasure metafunction from first language to second

- 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...

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

Programmers could use \text{float_of_int} and similar but they 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 $C$ when using $\tau \leq \tau'$ with subsumption.

So our translation to the untyped semantics depends on where we use subsumption. So it is really from typing derivations to programs.

And typing derivations aren’t unique (uh-oh).

Example 1: Suppose int $\leq$ float and $\tau \leq$ string. Consider $\cdot \vdash$ print_string(34) : unit.

Example 2: Suppose int $\leq$ \{l_1:int\}. Consider 34 == 34.
(Where == is bit-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.)

You can also make (complicated) rules about where subsumption occurs and which subtyping rules take precedence.

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")
Where are we

- “Subset” subtyping allows “upcasts”
- “Coercive subtyping” allows casts with run-time effect
- What about “downcasts”?
  That is, should we have something like:
  
  \[
  \text{if\_hastype}(\tau, e_1) \ \text{then} \ x.e_2 \ \text{else} \ e_3
  \]
  (Roughly, if at run-time \(e_1\) has type \(\tau\) (or a subtype), then bind it to \(x\) and evaluate \(e_2\). Else evaluate \(e_3\). Avoids having exceptions.)
Downcasts

I 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.

• Use ML-style datatypes – now programmer decides which data should have tags.

• Use parametric polymorphism – the right way to do container types (not downcasting results)

Now onto universally quantified types...
The Goal

Understand what this interface means and why it matters:

type 'a mylist;
val mt_list : 'a mylist
val cons : 'a -> 'a mylist -> 'a mylist
val decons : 'a mylist -> (('a * 'a mylist) option)
val length : 'a mylist -> int
val map : ('a -> 'b) -> 'a mylist -> 'b mylist

From two perspectives:

1. Library: Implement code to this partial specification
2. Client: Use code written to this partial specification
What The Client Likes

1. Library is reusable. Can make:
   - Different lists with elements of different types
   - New reusable functions outside of library. Example:
     ```haskell
     val twocons : 'a -> 'a -> 'a mylist -> 'a mylist
     ```

2. Easier, faster, more reliable than subtyping (cf. Java 1.4 Vector)
   - No downcast to write, run, maybe-fail

3. Library must “behave the same” for all “type instantiations”!!
   - ’a and ’b held abstract from library functions
   - E.g., with built-in lists: If foo has type ’a list -> int, foo [1;2;3] and foo [(5,4);(7,2);(9,2)] are totally equivalent! (Never true with downcasts)
   - In theory, means less (re)-integration testing
   - Proof is beyond this course, but not much
What the Library Likes

1. Reusability — For same reasons client likes it

2. Abstraction of mylist from clients
   - Clients must “behave the same” *for all* equivalent implementations, even if “hidden definition” of ’a mylist changes
   - Clients typechecked knowing only *there exists* a *type constructor* mylist
   - Unlike Java, C++, R5RS Scheme, no way to downcast a t mylist to, e.g., a pair
Start simpler

Our interface has a lot going on:

1. Element types *held abstract* from library
2. List type (constructor) *held abstract* from client
3. Reuse of type variables “makes connections” among expressions of abstract types
4. Lists need some form of recursive type
   - ST\(\lambda\)C has no unbounded data structures (except functions)

Today just consider (1) and (3)

- First using a formal language with explicit type abstraction
- Then highlight differences with ML

Note: Much more interesting than “not getting stuck”
Syntax

\[ e ::= c | x | \lambda x: \tau.\ e | e\ e | \Lambda\alpha.\ e | e[\tau] \]

\[ \tau ::= \text{int} | \tau \to \tau | \alpha | \forall\alpha.\tau \]

\[ v ::= c | \lambda x: \tau.\ e | \Lambda\alpha.\ e \]

\[ \Gamma ::= \cdot | \Gamma, x:\tau \]

\[ \Delta ::= \cdot | \Delta, \alpha \]

New:

- Type variables
- Types, terms, and contexts to know “what type variables are in scope” (much like we did for term variables)
- Type-applications to instantiate polymorphic expressions
Informally speaking

1. \( \Lambda \alpha. \ e \): A value that when used runs \( e \) (with some type \( \tau \) for \( \alpha \))
   - To type-check \( e \), know \( \alpha \) is one type, but not *which* type

2. \( e[\tau] \): Evaluate \( e \) to some \( \Lambda \alpha. \ e' \) and then run \( e' \)
   - The choice of \( \tau \) is irrelevant at run-time
   - \( \tau \) used for type-checking and proof of Preservation

3. Types can use type variables \( \alpha \), \( \beta \), etc., but only if they’re *in scope* (just like term variables)
   - Type-checking will be \( \Delta; \Gamma \vdash e : \tau \) to know what type variables are in scope in \( e \)
   - In a type with \( \forall \alpha. \tau \), can also use \( \alpha \) in \( \tau \)
Semantics

Our evaluation judgment (e.g., small-step left-right $e \rightarrow e'$) still looks the same. Just two new rules (note $\Lambda \alpha. e$ a value):

Old:

$$
\begin{align*}
& e_1 \rightarrow e'_1 \\
& e_2 \rightarrow e'_2
\end{align*}
$$

$$
\begin{align*}
& e_1 e_2 \rightarrow e'_1 e_2 \\
& v e_2 \rightarrow v e'_2
\end{align*}
$$

$$
(\lambda x: \tau. e) v \rightarrow e[v/x]
$$

New:

$$
\begin{align*}
& e \rightarrow e' \\
& e[\tau] \rightarrow e'[\tau]
\end{align*}
$$

$$(\Lambda \alpha. e)[\tau] \rightarrow e[\tau/\alpha]$$

Plus now have 3 different kinds of substitution, all defined in straightforward capture-avoiding way:

- $e_1[e_2/x]$ (old)
- $e[\tau'/\alpha]$ (new)
- $\tau[\tau'/\alpha]$ (new)
Example

Example (using addition):

$$(\Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \rightarrow \beta. f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y : \text{int}. y + y)$$
Typing, part 1

Mostly we just get picky about “no free type variables”:

- Typing judgment has the form $\Delta; \Gamma \vdash e : \tau$
  (whole program $\cdot; \cdot \vdash e : \tau$)
  - Next slide

- Uses helper judgment $\Delta \vdash \tau$
  - “all free type variables in $\tau$ are in $\Delta$”

\[
\begin{array}{c}
\Delta \vdash \tau \\
\end{array}
\]

\[
\begin{array}{c}
\frac{\alpha \in \Delta}{\Delta \vdash \alpha} \\
\frac{}{\Delta \vdash \text{int}} \\
\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \rightarrow \tau_2} \\
\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall \alpha. \tau}
\end{array}
\]

Rules are boring, but trust me, allowing free type variables is a pernicious source of language/compiler bugs
Typing, part 2

Old (with one technical change to prevent free type variables):

\[
\begin{align*}
\Delta; \Gamma \vdash x : \Gamma(x) & \quad \quad \Delta; \Gamma \vdash c : \text{int} \\
\Delta; \Gamma, x: \tau_1 \vdash e : \tau_2 & \quad \quad \Delta \vdash \tau_1 \\
\Delta; \Gamma \vdash \lambda x: \tau_1. \ e : \tau_1 \rightarrow \tau_2 \\
\Delta; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 & \quad \quad \Delta; \Gamma \vdash e_2 : \tau_2 \\
\Delta; \Gamma \vdash e_1 \ e_2 : \tau_1
\end{align*}
\]

New:

\[
\begin{align*}
\Delta, \alpha; \Gamma \vdash e : \tau_1 & \quad \quad \Delta; \Gamma \vdash e : \forall \alpha. \tau_1 \quad \Delta \vdash \tau_2 \\
\Delta; \Gamma \vdash \Lambda \alpha. \ e : \forall \alpha. \tau_1 & \quad \quad \Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]
\end{align*}
\]
Example (using addition):

\[(Λα. Λβ. λx : α. λf:α → β. f x) \text{[int]} \text{[int]} 3 (λy : \text{int}. y + y)\]
The Whole Language (called System F)

\[
\begin{align*}
\text{e} & ::= \ c \mid x \mid \lambda x : \tau . \ e \mid e \ e \mid \Lambda \alpha . \ e \mid e[\tau] \\
\text{\tau} & ::= \ \text{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha . \tau \\
\text{v} & ::= \ c \mid \lambda x : \tau . \ e \mid \Lambda \alpha . \ e \\
\Gamma & ::= \ \cdot \mid \Gamma , x : \tau \\
\Delta & ::= \ \cdot \mid \Delta , \alpha \\
\end{align*}
\]

\[
\begin{align*}
\frac{e \to e'}{e \ e_2 \to e' \ e_2} & & \frac{e \to e'}{v \ e \to v \ e'} & & \frac{e \to e'}{e[\tau] \to e'[\tau]}
\end{align*}
\]

\[
\begin{align*}
(\lambda x : \tau . \ e) \ v & \to e[v/x] & (\Lambda \alpha . \ e)[\tau] & \to e[\tau/\alpha] \\
\frac{}{\Delta ; \Gamma \vdash x : \Gamma(x)} & & \frac{}{\Delta ; \Gamma \vdash c : \text{int}}
\end{align*}
\]

\[
\begin{align*}
\Delta ; \Gamma , x : \tau_1 \vdash e : \tau_2 & & \Delta ; \Gamma \vdash \tau_1 \\
\frac{}{\Delta ; \Gamma \vdash \lambda x : \tau_1 . \ e : \tau_1 \to \tau_2} & & \frac{}{\Delta , \alpha ; \Gamma \vdash e : \tau_1}
\end{align*}
\]

\[
\begin{align*}
\Delta ; \Gamma \vdash e_1 : \tau_2 \to \tau_1 & & \Delta ; \Gamma \vdash e_2 : \tau_2 \\
\frac{}{\Delta ; \Gamma \vdash e_1 \ e_2 : \tau_1} & & \frac{}{\Delta ; \Gamma \vdash e : \forall \alpha . \tau_1 \Delta \vdash \tau_2}
\end{align*}
\]

\[
\begin{align*}
\Delta ; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha] & & \Delta ; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]
\end{align*}
\]
Examples

An overly simple polymorphic function...

Let $id = \Lambda \alpha. \lambda x : \alpha. x$

- $id$ has type $\forall \alpha. \alpha \rightarrow \alpha$
- $id[\text{int}]$ has type $\text{int} \rightarrow \text{int}$
- $id[\text{int} \ast \text{int}]$ has type $(\text{int} \ast \text{int}) \rightarrow (\text{int} \ast \text{int})$
- $(id[\forall \alpha. \alpha \rightarrow \alpha]) id$ has type $\forall \alpha. \alpha \rightarrow \alpha$

In ML you can’t do the last one; in System F you can.
More Examples

Let applyOld = \( \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \rightarrow \beta. f \ x \)

- applyOld has type \( \forall \alpha. \forall \beta. \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \)
- \( \cdot; x:\text{int} \rightarrow \text{int} \vdash (\text{applyOld} [\text{int}][\text{int}] 3 \ x) : \text{int} \)

Let applyNew = \( \Lambda \alpha. \lambda x : \alpha. \Lambda \beta. \lambda f : \alpha \rightarrow \beta. f \ x \)

- applyNew has type \( \forall \alpha. \alpha \rightarrow (\forall \beta. (\alpha \rightarrow \beta) \rightarrow \beta) \)
  (impossible in ML)
- \( \cdot; x:\text{int} \rightarrow \text{string}, y:\text{int} \rightarrow \text{int} \vdash \)
  \( (\text{let} \ z = \text{applyNew} [\text{int}] \ \text{in} \ z \ (z \ 3 \ \text{[int]} \ y) \ \text{[string]} \ x) : \text{string} \)

Let twice = \( \Lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \rightarrow \alpha. f \ (f \ x) \).

- twice has type \( \forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \)
- Cannot be made more polymorphic