subsumption:

But we conjectured that applications were the only critical uses of subsumption.

We observed last time that this rule is sometimes needed when typechecking applications:

\[
\frac{\text{subsumption}}{\frac{\tau \vdash \eta}{\frac{\eta \vdash \xi}{\frac{\xi \vdash \eta}{\frac{\xi \vdash \tau}{\xi \vdash \tau}}}}}
\]

For the typing relation, we have just one problematic rule to deal with:

\[
\frac{\text{T-Sem}}{\frac{\tau \vdash \eta}{\frac{\eta \vdash \xi}{\frac{\xi \vdash \tau}{\xi \vdash \eta}}}}
\]
I. Investigate how subsumption is used in typing derivations by looking at examples of how it can be pushed through "other rules."

II. Use the intuition gained from this exercise to design a new, algorithmic typing relation that omits subsumption. Compensate for its absence by enriching the application rule.

III. Show that the algorithmic typing relation is essentially equivalent to the original, declarative one.
Example (T-Sub with T-App on the right)

Example (T-Sub with T-App on the left)

Intuitions
Example (nested uses of T-Sub)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Sub</td>
<td>$T_1 \Rightarrow S$</td>
<td>$T_2 \Rightarrow S$</td>
</tr>
<tr>
<td>S-Trans</td>
<td>$T_1 \Rightarrow S$</td>
<td>$T_2 \Rightarrow S$</td>
</tr>
</tbody>
</table>

Intuitions

Intuitively, because T-App, S-Refl, and S-Arrow are immediately before T-App, S-Refl, and S-Arrow, respectively, the order cannot be completely reversed.

We see that uses of substitution can be "pushed" from one side of the premises to the other.
Summary

What we've learned:

Uses of the T-Sub rule can be pushed down "through" typing derivations until they encounter either

1. A use of T-App
2. The root of the derivation tree.

In both cases, multiple uses of T-Sub can be collapsed into a single one.

In both cases, multiple uses of T-Sub can be collapsed into a single one.

Minimal Types

But... if subsumption is only used at the very end of derivations, then it is actually not needed in order to show that any term is typable.

If we dropped subsumption completely (after renaming the type symbols), then it is actually not needed in order to show that any term is typable.

But... if subsumption is only used at the very end of derivations, then it is actually not needed in order to show that any term is typable.

In other words, if we dropped subsumption completely (after renaming the type symbols), then it is actually not needed in order to show that any term is typable.

Algorithmic Typing

The next step is to "build in" the use of subsumption in application rules.

The next step is to "build in" the use of subsumption in application rules.

This yields a derivation in which there is just one use of subsumption at the extended rule above.

and

The next step is to "build in" the use of subsumption in application rules.

The next step is to "build in" the use of subsumption in application rules.
Soundness of the algorithmic rules

Theorem (Soundness): If \( I \vdash t : T \), then \( \exists S \geq T \).

Proof: Homework.

Completeness of the algorithmic rules

Theorem (Completeness): If \( t : T \), then \( \exists S < T \).

Proof: Homework. (N.b.: All the messing around with transforming derivations was just to build intuitions and decide which algorithmic rules to write down and which property to prove. The proof itself is a straightforward induction on typing derivations.)
Adding Booleans

Suppose we want to add booleans and conditionals to the language we have.

For the declarative presentation of the system, we just add in the appropriate syntax, forms, evaluation rules, and typing rules.

A Problem with Conditional Expressions

For the algorithmic presentation, however, we encounter a difficulty.

What is the minimal type of

\[
\text{if true then } \{x=true, y=false\} \text{ else } \{x=true, z=true\}
\]

The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

The Algorithmic Conditional Rule

\[
\text{if } t_1 \text{ then } t_2 \text{ else } t_3
\]

of the minimal type of \(t_2\) and the minimal type of \(t_3\).

So the minimal type of the conditional is the least common supertype (or join) any type that is a possible type of both \(t_2\) and \(t_3\).

If \(t_1\) is true then \(t_2\) else \(t_3\):

\[
\text{Join } t_2 \text{ and } t_3
\]

where generally, we can use subsumption to give an expression

of the minimal type of \(t_2\) and the minimal type of \(t_3\).

Any type that is a possible type of both \(t_2\) and \(t_3\).

The minimal type of the conditional is the least common supertype (or join) any type that is a possible type of both \(t_2\) and \(t_3\).

So the minimal type of the conditional is the least common supertype (or join) any type that is a possible type of both \(t_2\) and \(t_3\).

The minimal type of the conditional is the least common supertype (or join) any type that is a possible type of both \(t_2\) and \(t_3\).

The minimal type of the conditional is the least common supertype (or join) any type that is a possible type of both \(t_2\) and \(t_3\).
The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression if $t_1$ then $t_2$ else $t_3$.

Theorem: For every pair of types $S$ and $T$, there is a type $J$ such that

Moreover...

However...

If $S$, $T$ have no common subtypes, so they certainly don't have a greatest one, and $\bot$.

Under joins, meets do not necessarily exist.

(Least upper bounds)!

To calculate joins of arrow types, we also need to be able to calculate meets.

Examples

Whatare the joins of the following pairsof types?

1. $\{x: \text{Bool}, y: \text{Bool}\}$ and $\{y: \text{Bool}, z: \text{Bool}\}$
2. $\{x: \text{Bool}\}$ and $\{y: \text{Bool}\}$
3. $\{x: \{a: \text{Bool}, b: \text{Bool}\}\}$ and $\{x: \{b: \text{Bool}, c: \text{Bool}\}, y: \text{Bool}\}$
4. $\{\}\$ and $\{\}$
5. $\{\}$ and $\text{Bool}$
6. $\{\text{Boot}: x: \text{Boot}\}$ and $\{\text{Boot}: y: \text{Boot}\}$
7. $\{\text{Boot}: x: \text{Boot}\}$ and $\{\text{Boot}: y: \text{Boot}, c: \text{Boot}: q: x: \text{Boot}\}$
8. $\{\text{Boot}: y: \text{Boot}\}$ and $\{\text{Boot}: x: \text{Boot}\}$

What are the joins of the following pairs of types?

Existence of Joins

Theorem: For every $T_1$ and $T_2$,

$\exists$ such a type $J$.

Does such a type exist for every $T_1$ and $T_2$?

$\forall x, y : T_1 \times T_2 \forall T_3 : T_3$.

So the minimal type of the conditional is the least common suptype (or join) of the minimal type of $T_2$ and $T_3$.

Any type that is a possible type of both $T_2$ and $T_3$.

If $T_1$ then $T_2$ else $T_3$.
**Existence of Meets**

Theorem: For every pair of types $S$ and $T$, if there is any type $N$ such that $N <: S$ and $N <: T$, then there is a type $M$ such that

1. $M <: S$
2. $M <: T$
3. If $O$ is a type such that $O <: S$ and $O <: T$, then $O <: M$.

I.e., $M$ (when it exists) is the largest type that is a subtype of both $S$ and $T$.

**Jargon:** In the simply typed lambda calculus with subtyping, records, and booleans...

The subtyping relation has **bounded meets**.