SNU 4541.574 Programming Language Theory

Ack: BCP's slides

Algorithmic Typing

Algorithmic typing

- ► How do we implement a type checker for the lambda-calculus with subtyping?
- ► Given a context \(\Gamma\) and a term \(\tau, \) how do we determine its type \(T, \) such that \(\Gamma \dagger t : T? \)

Issue

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

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$
 (T-Sub)

We observed above that this rule is sometimes *required* when typechecking applications:

E.g., the term

$$(\lambda r: \{x: Nat\}. r.x) \{x=0, y=1\}$$

is not typable without using subsumption.

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

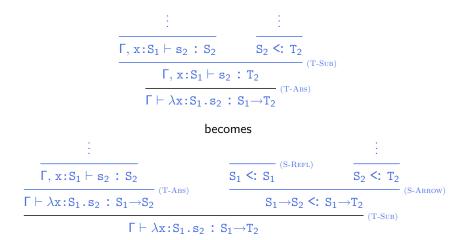
Plan

- Investigate how subsumption is used in typing derivations by looking at examples of how it can be "pushed through" other rules
- 2. Use the intuitions gained from this exercise to design a new, algorithmic typing relation that
 - omits subsumption
 - compensates for its absence by enriching the application rule
- 3. Show that the algorithmic typing relation is essentially equivalent to the original, declarative one

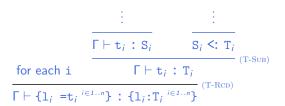
Example (T-Sub with T-Abs)

```
\frac{\vdots}{\frac{\Gamma, x: S_1 \vdash s_2: S_2}{\Gamma, x: S_1 \vdash s_2: T_2}} \frac{\vdots}{\frac{S_2 \lt: T_2}{\Gamma \vdash \lambda x: S_1 \cdot s_2: S_1 \rightarrow T_2}} (T-ABS)}
```

Example (T-Sub with T-Abs)



Example (T-Sub with T-Rcd)



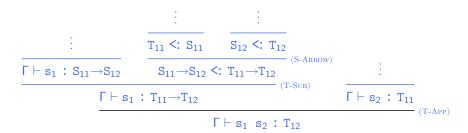
Intuitions

These examples show that we do not need T-Sub to "enable" T-Abs or T-Rcd: given any typing derivation, we can construct a derivation with the same conclusion in which T-Sub is never used immediately before T-Abs or T-Rcd.

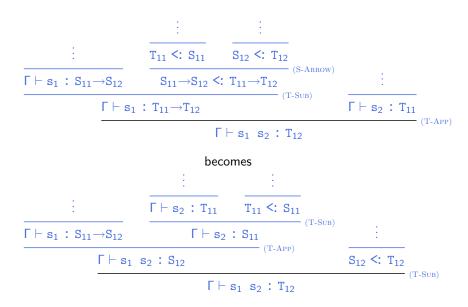
What about T-APP?

We've already observed that $T\textsc{-}\mathrm{SuB}$ is required for typechecking some applications. So we expect to find that we *cannot* play the same game with $T\textsc{-}\mathrm{APP}$ as we've done with $T\textsc{-}\mathrm{ABS}$ and $T\textsc{-}\mathrm{RcD}$. Let's see why.

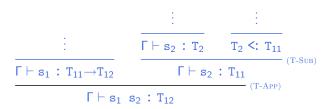
Example (T-Sub with T-App on the left)



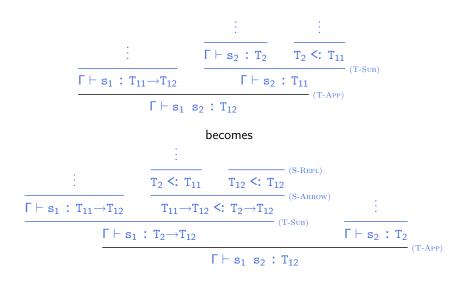
Example (T-Sub with T-App on the left)



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



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



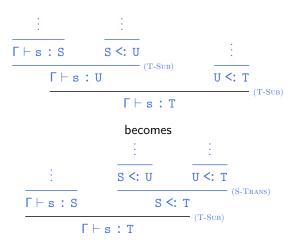
Intuitions

So we've seen that uses of subsumption can be "pushed" from one of immediately before $\operatorname{T-APP}$'s premises to the other, but cannot be completely eliminated.

Example (nested uses of T-Sub)

```
\frac{\vdots}{\Gamma \vdash s : S} \qquad \frac{\vdots}{S \lessdot : U} \qquad \qquad \vdots \\
\frac{\Gamma \vdash s : U}{\Gamma \vdash s : T} \qquad \qquad (T-SUB)
```

Example (nested uses of T-Sub)



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 or
 - 2. the root fo the derivation tree.
- ▶ In both cases, multiple uses of T-SUB can be collapsed into a single one.

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 or
 - 2. the root fo the derivation tree.
- ▶ In both cases, multiple uses of T-SUB can be collapsed into a single one.

This suggests a notion of "normal form" for typing derivations, in which there is

- ▶ exactly one use of T-Sub before each use of T-App
- ▶ one use of T-Sub at the very end of the derivation
- ▶ no uses of T-SUB anywhere else.

Algorithmic Typing

The next step is to "build in" the use of subsumption in application rules, by changing the $T\text{-}\mathrm{APP}$ rule to incorporate a subtyping premise.

$$\frac{\Gamma \vdash \mathtt{t}_1 \,:\, \mathtt{T}_{11} \!\rightarrow\! \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 \,:\, \mathtt{T}_2 \qquad \vdash \mathtt{T}_2 \mathrel{<:} \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1 \ \mathtt{t}_2 \,:\, \mathtt{T}_{12}}$$

Given any typing derivation, we can now

- normalize it, to move all uses of subsumption to either just before applications (in the right-hand premise) or at the very end
- 2. replace uses of T-APP with T-SUB in the right-hand premise by uses of the extended rule above

This yields a derivation in which there is just *one* use of subsumption, at the very end!

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!

It is just used to give *more* types to terms that have already been shown to have a type.

In other words, if we dropped subsumption completely (after refining the application rule), we would still be able to give types to exactly the same set of terms — we just would not be able to give as many types to some of them.

If we drop subsumption, then the remaining rules will assign a *unique*, *minimal* type to each typable term.

For purposes of building a typechecking algorithm, this is enough.

Final Algorithmic Typing Rules

$$\frac{\mathbf{x}: \mathsf{T} \in \mathsf{\Gamma}}{\mathsf{\Gamma} \models \mathbf{x}: \mathsf{T}} \qquad (\mathsf{TA-VAR})$$

$$\frac{\mathsf{\Gamma}, \mathbf{x}: \mathsf{T}_1 \models \mathsf{t}_2: \mathsf{T}_2}{\mathsf{\Gamma} \models \lambda \mathsf{x}: \mathsf{T}_1. \mathsf{t}_2: \mathsf{T}_1 \to \mathsf{T}_2} \qquad (\mathsf{TA-ABS})$$

$$\frac{\mathsf{\Gamma} \models \mathsf{t}_1: \mathsf{T}_1}{\mathsf{T} \models \mathsf{t}_1: \mathsf{T}_1} \qquad \mathsf{T} \models \mathsf{t}_2: \mathsf{T}_2 \qquad \models \mathsf{T}_2 <: \mathsf{T}_{11}}{\mathsf{\Gamma} \models \mathsf{t}_1: \mathsf{t}_2: \mathsf{T}_{12}} \qquad (\mathsf{TA-APP})$$

$$\frac{\mathsf{for each } i \qquad \mathsf{\Gamma} \models \mathsf{t}_i: \mathsf{T}_i}{\mathsf{\Gamma} \models \mathsf{t}_1: \mathsf{L}_1 = \mathsf{t}_n } : \{\mathsf{l}_1: \mathsf{T}_1 \dots \mathsf{l}_n: \mathsf{T}_n \}} \qquad (\mathsf{TA-RcD})$$

$$\frac{\mathsf{\Gamma} \models \mathsf{t}_1: \mathsf{R}_1 \qquad \mathsf{R}_1 = \{\mathsf{l}_1: \mathsf{T}_1 \dots \mathsf{l}_n: \mathsf{T}_n \}}{\mathsf{\Gamma} \models \mathsf{t}_1. \mathsf{l}_i: \mathsf{T}_i} \qquad (\mathsf{TA-PROJ})$$

Soundness of the algorithmic rules

Theorem: If $\Gamma \triangleright t : T$, then $\Gamma \vdash t : T$.

Completeness of the algorithmic rules

Theorem [Minimal Typing]: If $\Gamma \vdash t : T$, then $\Gamma \blacktriangleright t : S$ for some $S \leq T$.

Meets and Joins

Adding Booleans

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

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

```
\begin{array}{c} \Gamma \vdash \text{true} : \text{Bool} & \text{(T-True)} \\ \Gamma \vdash \text{false} : \text{Bool} & \text{(T-False)} \\ \hline \frac{\Gamma \vdash \text{t}_1 : \text{Bool} \quad \Gamma \vdash \text{t}_2 : T \quad \Gamma \vdash \text{t}_3 : T}{\Gamma \vdash \text{if} \ \text{t}_1 \ \text{then} \ \text{t}_2 \ \text{else} \ \text{t}_3 : T} \end{array} \tag{T-IF}
```

A Problem with Conditional Expressions

For the *algorithmic* presentation of the system, however, we encounter a little difficulty.

What is the minimal type of

```
if true then {x=true,y=false} else {x=true,z=true}
```

The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

if
$$t_1$$
 then t_2 else t_3

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

$$\frac{\Gamma \Vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \Vdash \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \Vdash \mathtt{t}_3 : \mathtt{T}_3}{\Gamma \Vdash \mathtt{if} \ \mathtt{t}_1 \ \mathtt{then} \ \mathtt{t}_2 \ \mathtt{else} \ \mathtt{t}_3 : \mathtt{T}_2 \vee \mathtt{T}_3} \quad \text{(T-IF)}$$

The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

if
$$t_1$$
 then t_2 else t_3

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

$$\frac{\Gamma \Vdash \mathsf{t}_1 : \mathsf{Bool} \qquad \Gamma \Vdash \mathsf{t}_2 : T_2 \qquad \Gamma \Vdash \mathsf{t}_3 : T_3}{\Gamma \Vdash \mathsf{if} \ \mathsf{t}_1 \ \mathsf{then} \ \mathsf{t}_2 \ \mathsf{else} \ \mathsf{t}_3 : T_2 \vee T_3} \quad \text{(T-IF)}$$

Does such a type exist for every T_2 and T_3 ??

Existence of Joins

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

- 1. S <: J
- 2. T <: J
- 3. If K is a type such that $S \le K$ and $T \le K$, then $J \le K$.

I.e., J is the smallest type that is a supertype of both S and T.

Examples

What are the joins of the following pairs of types?

```
    {x:Bool,y:Bool} and {y:Bool,z:Bool}?
    {x:Bool} and {y:Bool}?
    {x:{a:Bool,b:Bool}} and {x:{b:Bool,c:Bool}, y:Bool}?
    {} and Bool?
    {x:{}} and {x:Bool}?
    Top→{x:Bool} and Top→{y:Bool}?
    {x:Bool}→Top and {y:Bool}→Top?
```

Meets

To calculate joins of arrow types, we also need to be able to calculate *meets* (greatest lower bounds)!

Unlike joins, meets do not necessarily exist.

E.g., $Bool \rightarrow Bool$ and $\{\}$ have *no* common subtypes, so they certainly don't have a greatest one!

However...

Existence of Meets

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

- 1. M <: S
- 2. M <: T
- 3. If 0 is a type such that $0 \le S$ and $0 \le T$, then $0 \le M$.

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 subtype relation *has joins*
- ► The subtype relation *has* bounded *meets*

Examples

What are the meets of the following pairs of types?

```
    {x:Bool,y:Bool} and {y:Bool,z:Bool}?
    {x:Bool} and {y:Bool}?
    {x:{a:Bool,b:Bool}} and {x:{b:Bool,c:Bool}}, y:Bool}?
    {} and Bool?
    {x:{}} and {x:Bool}?
    Top→{x:Bool} and Top→{y:Bool}?
    {x:Bool}→Top and {y:Bool}→Top?
```

Calculating Joins

```
 S \vee T \ = \ \begin{cases} \ Bool & \text{if } S = T = Bool \\ M_1 \rightarrow J_2 & \text{if } S = S_1 \rightarrow S_2 & T = T_1 \rightarrow T_2 \\ S_1 \wedge T_1 = M_1 & S_2 \vee T_2 = J_2 \\ \{j_I \colon J_I \overset{I \in I \dots q}{} \} & \text{if } S = \{k_j \colon S_j \overset{j \in I \dots m}{} \} \\ T = \{l_i \colon T_i \overset{i \in I \dots m}{} \} & \{j_I \overset{I \in I \dots m}{} \} = \{k_j \overset{j \in I \dots m}{} \} \cap \{l_i \overset{i \in I \dots m}{} \} \\ S_j \vee T_i = J_I & \text{for each } j_I = k_j = l_i \end{cases}  Top otherwise
```

Calculating Meets

 $S \wedge T =$

```
\begin{cases} & \text{S} & \text{if } T = Top \\ & T & \text{if } S = Top \\ & \text{Bool} & \text{if } S = T = Bool \\ & J_1 {\rightarrow} M_2 & \text{if } S = S_1 {\rightarrow} S_2 & T = T_1 {\rightarrow} T_2 \\ & S_1 \vee T_1 = J_1 & S_2 \wedge T_2 = M_2 \\ & \{m_i : M_i \mid i \in 1...q\} & \text{if } S = \{k_j : S_j \mid i \in 1...m\} \\ & T = \{1_i : T_i \mid i \in 1...n\} \end{cases}
                                                       \{\mathbf{m}_{i}^{l \in 1..q}\} = \{\mathbf{k}_{i}^{j \in 1..m}\} \cup \{\mathbf{1}_{i}^{i \in 1..n}\}
                                                      S_i \wedge T_i = M_I for each m_I = k_i = 1_i
                                              M_I = S_i if m_I = k_i occurs only in S
                                 M_I = T_i if m_I = 1_i occurs only in T
                                            otherwise
```