CIS 500 Software Foundations Fall 2006

November 15

From last time...

A decision function for a relation $R \subseteq U$ is a total function p from U to $\{true, false\}$ such that p(u) = true iff $u \in R$.

A decision function for a relation $R \subseteq U$ is a total function p from U to $\{true, false\}$ such that p(u) = true iff $u \in R$.

Example:

$$U = \{1, 2, 3\}$$

 $R = \{(1, 2), (2, 3)\}$

Note that, for now, we are saying absolutely nothing about *computability*. We'll come back to this in a moment.

A decision function for a relation $R \subseteq U$ is a total function p from U to $\{true, false\}$ such that p(u) = true iff $u \in R$.

Example:

$$U = \{1, 2, 3\}$$

 $R = \{(1, 2), (2, 3)\}$

The function p whose graph is

```
 \{ \ ((1,2), \ true), \ ((2,3), \ true), \\ \ ((1,1), \ false), \ ((1,3), \ false), \\ \ ((2,1), \ false), \ ((2,2), \ false), \\ \ ((3,1), \ false), \ ((3,2), \ false), \ ((3,3), \ false) \}
```

is a decision function for R.

A decision function for a relation $R \subseteq U$ is a total function p from U to $\{true, false\}$ such that p(u) = true iff $u \in R$.

Example:

$$U = \{1, 2, 3\}$$

 $R = \{(1, 2), (2, 3)\}$

The function p' whose graph is

$$\{((1,2), true), ((2,3), true)\}$$

is *not* a decision function for R.

A decision function for a relation $R \subseteq U$ is a total function p from U to $\{true, false\}$ such that p(u) = true iff $u \in R$.

Example:

$$U = \{1, 2, 3\}$$

 $R = \{(1, 2), (2, 3)\}$

The function p'' whose graph is

$$\{((1,2), true), ((2,3), true), ((1,3), false)\}$$

is also *not* a decision function for R.

Of course, we want a decision procedure to be a procedure.

A decision procedure for a relation $R \subseteq U$ is a computable total function p from U to $\{true, false\}$ such that p(u) = true iff $u \in R$.

Example

$$U = \{1, 2, 3\}$$

 $R = \{(1, 2), (2, 3)\}$

Example

$$U = \{1, 2, 3\}$$

 $R = \{(1, 2), (2, 3)\}$

The function

$$p(x,y) = if x = 2$$
 and $y = 3$ then true else if $x = 1$ and $y = 2$ then true else false

whose graph is

```
\{ ((1,2), true), ((2,3), true), ((1,1), false), ((1,3), false), ((2,1), false), ((2,2), false), ((3,1), false), ((3,2), false), ((3,3), false) \}
```

is a decision procedure for R.

Example

$$U = \{1, 2, 3\}$$

 $R = \{(1, 2), (2, 3)\}$

The recursively defined partial function

$$p(x,y) = if x = 2$$
 and $y = 3$ then true
else if $x = 1$ and $y = 2$ then true
else if $x = 1$ and $y = 3$ then false
else $p(x,y)$

whose graph is

```
\{ ((1,2), true), ((2,3), true), ((1,3), false) \}
```

is *not* a decision procedure for R.

Subtyping Algorithm

This recursively defined *total* function is a decision procedure for the subtype relation:

```
\begin{aligned} & \textit{subtype}(S,T) &= \\ & \text{if } T = Top, \text{ then } \textit{true} \\ & \text{else if } S = S_1 {\rightarrow} S_2 \text{ and } T = T_1 {\rightarrow} T_2 \\ & \text{then } \textit{subtype}(T_1,S_1) \ \land \ \textit{subtype}(S_2,T_2) \\ & \text{else if } S = \{k_j {:} S_j {}^{j {\in} 1..m} \} \text{ and } T = \{1_j {:} T_i {}^{i {\in} 1..n} \} \\ & \text{then } \{1_i {}^{i {\in} 1..n} \} \subseteq \{k_j {}^{j {\in} 1..m} \} \\ & \land \text{ for all } i \in 1..n \text{ there is some } j \in 1..m \text{ with } k_j = 1_i \\ & \text{and } \textit{subtype}(S_j,T_i) \\ & \text{else } \textit{false}. \end{aligned}
```

To show this, we need to prove:

- 1. that it returns true whenever S <: T, and
- 2. that it returns either true or false on all inputs.

Subtyping Algorithm

But this recursively defined *partial* function is not:

```
subtype(S,T) = \\ \text{if } T = Top, \text{ then } true \\ \text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \\ \text{then } subtype(T_1,S_1) \land subtype(S_2,T_2) \\ \text{else if } S = \{k_j:S_j^{\ j \in 1..m}\} \text{ and } T = \{1_j:T_i^{\ i \in 1..n}\} \\ \text{then } \{1_i^{\ i \in 1..n}\} \subseteq \{k_j^{\ j \in 1..m}\} \\ \land \text{ for all } i \in 1..n \text{ there is some } j \in 1..m \text{ with } k_j = 1_i \\ \text{and } subtype(S_j,T_i) \\ \text{else } subtype(T,S) \\ \end{cases}
```

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)

```
 \begin{array}{c} \vdots \\ \hline \Gamma, \mathbf{x} \colon \! \mathbf{S}_1 \vdash \mathbf{s}_2 : \mathbf{S}_2 & \overline{\mathbf{S}}_2 \lessdot \mathbf{T}_2 \\ \hline \Gamma, \mathbf{x} \colon \! \mathbf{S}_1 \vdash \mathbf{s}_2 : \mathbf{T}_2 & \overline{\mathbf{T}}_{-\mathrm{Abs}} \\ \hline \hline \Gamma \vdash \lambda \mathbf{x} \colon \! \mathbf{S}_1 \colon \mathbf{s}_2 : \mathbf{S}_1 \! \to \! \mathbf{T}_2 & \overline{\mathbf{T}}_2 \\ \hline \end{array}
```

Example (T-Sub with T-Abs)

$$\begin{array}{c|c} \vdots & \vdots \\ \hline \Gamma, x \colon S_1 \vdash s_2 \colon S_2 & \overline{S_2} \lessdot T_2 \\ \hline \hline \Gamma, x \colon S_1 \vdash s_2 \colon T_2 & \text{\tiny (T-SUB)} \\ \hline \hline \Gamma \vdash \lambda x \colon S_1 \colon s_2 \colon S_1 \to T_2 & \text{\tiny (T-ABS)} \\ \hline \vdots & \vdots & \vdots \\ \hline \Gamma, x \colon S_1 \vdash s_2 \colon S_2 & \overline{S_1} & \overline{S_2} & \vdots \\ \hline \hline \Gamma \vdash \lambda x \colon S_1 \colon s_2 \colon S_1 \to S_2 & \overline{S_1} \to T_2 \\ \hline \hline \Gamma \vdash \lambda x \colon S_1 \colon s_2 \colon S_1 \to T_2 & \overline{S_1} \to T_2 \\ \hline \hline \end{array}$$

Example (T-Sub with T-Rcd)

$$\frac{\vdots}{\Gamma \vdash \mathsf{t}_i : S_i} \qquad \frac{\vdots}{S_i <: T_i} \\ \frac{\text{for each i}}{\Gamma \vdash \mathsf{t}_i : T_i} \\ \frac{\Gamma \vdash \mathsf{t}_i : T_i}{\Gamma \vdash \{1_i = \mathsf{t}_i \stackrel{i \in 1..n}{\longrightarrow}\} : \{1_i : T_i \stackrel{i \in 1..n}{\longrightarrow}\}}$$

Intuitions

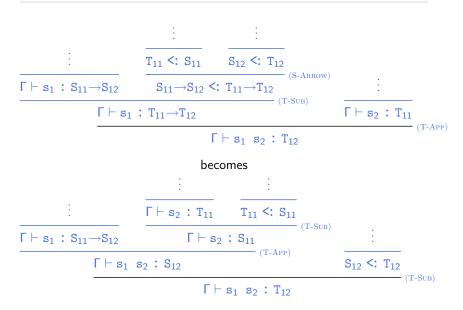
These examples show that we do not need T-SuB to "enable" T-ABS or T-RCD: given any typing derivation, we can construct a derivation with the same conclusion in which T-SuB is never used immediately before T-ABS or T-RCD.

What about T-APP?

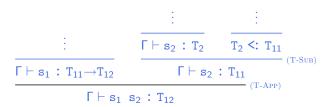
We've already observed that $T\textsc{-}\mathrm{SuB}$ is required for typechecking some applications. So we expect to find that we cannot play the same game with $T\textsc{-}\mathrm{APP}$ as we've done with $T\textsc{-}\mathrm{ABS}$ and $T\textsc{-}\mathrm{RcD}$. Let's see why.

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

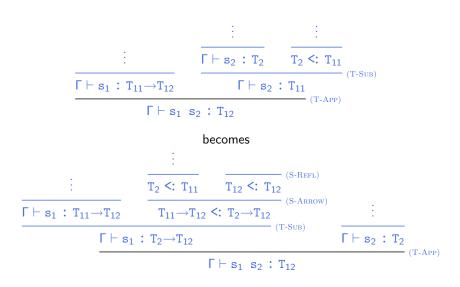
Example (T-Sub with T-App on the left)



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



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



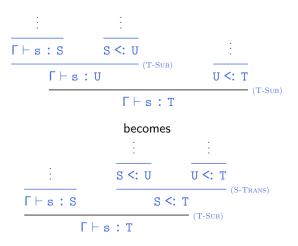
Intuitions

So we've seen that uses of subsumption can be "pushed" from one of immediately before 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 U \lessdot T
```

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-App rule to incorporate a subtyping premise.

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_{11} \rightarrow \mathsf{T}_{12} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_2 \qquad \vdash \mathsf{T}_2 \lessdot: \mathsf{T}_{11}}{\Gamma \vdash \mathsf{t}_1 \ \mathsf{t}_2 : \mathsf{T}_{12}}$$

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

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

$$\frac{\Gamma, x:T_1 \Vdash t_2:T_2}{\Gamma \Vdash \lambda x:T_1.t_2:T_1\to T_2} \qquad (TA-ABS)$$

$$\frac{\Gamma \Vdash t_1:T_1 \qquad T_1=T_{11}\to T_{12} \qquad \Gamma \Vdash t_2:T_2 \qquad \vdash T_2 <:T_{11}}{\Gamma \Vdash t_1:t_2:T_{12}} \qquad (TA-APP)$$

$$\frac{\Gamma \vdash t_1:T_1 \qquad \Gamma \vdash t_1:T_1}{\Gamma \vdash t_1:T_1\dots t_n:T_n} \qquad (TA-RCD)$$

$$\frac{\Gamma \vdash t_1:R_1 \qquad R_1=\{1_1:T_1\dots 1_n:T_n\}}{\Gamma \vdash t_1.1_i:T_i} \qquad (TA-PROJ)$$

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 $S \leq T$.

Completeness of the algorithmic rules

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

Proof: Induction on typing derivation. (*Details on this week's homework.*)

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

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 t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if} \ t_1 \ \text{then} \ t_2 \ \text{else} \ 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 \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} \qquad \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 .

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

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 <: K and T <: K, then J <: 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→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 <: 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 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

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

Calculating Meets

```
S \wedge T
```

```
 \begin{cases} S & \text{if } T = Top \\ T & \text{if } S = Top \\ 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 \lor T_1 = J_1 & S_2 \land T_2 = M_2 \\ \{m_I : M_I \ ^{I \in I...q} \} & \text{if } S = \{k_j : S_j \ ^{j \in I..m} \} \\ & T = \{1_j : T_i \ ^{i \in I...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
```