CIS 500 Software Foundations Fall 2006 November 15	From last time
Decision Procedures (take 1) 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 $u \in R$.	Decision Procedures (take 1) 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 <i>computability</i> . We'll come back to this in a moment.
Decision Procedures (take 1) 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 $\{ ((1, 2), true), ((2, 3), true), ((1, 1), false), ((1, 3), false), ((3, 2), false), ((3, 3), false)\}$ is a decision function for R .	Decision Procedures (take 1) 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 .

Decision Procedures (take 1) 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: $U = \{1, 2, 3\}$ $R = \{(1, 2), (2, 3)\}$	Decision Procedures (take 2) 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 $u \in R$.
The function p'' whose graph is	
$\{((1,2), true), ((2,3), true), ((1,3), false)\}$	
is also <i>not</i> a decision function for <i>R</i> .	
Example	Example
$U = \{1, 2, 3\}$ R = {(1, 2), (2, 3)}	$U = \{1, 2, 3\}$ $R = \{(1, 2), (2, 3)\}$
	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 $\left\{ \begin{array}{l} ((1, 2), true), ((2, 3), true), \\ ((1, 1), false), ((1, 3), false), \\ ((2, 1), false), ((2, 2), false), \\ ((3, 1), false), ((3, 2), false), ((3, 3), false) \right\}$ is a decision procedure for <i>R</i> .
Example	Subtyping Algorithm
$U = \{1, 2, 3\}$ $R = \{(1, 2), (2, 3)\}$ The recursively defined partial function	This recursively defined <i>total</i> function is a decision procedure for the subtype relation: subtype(S,T) = if $T = Top$, then <i>true</i> else if $S = S_1 \rightarrow S_2$ and $T = T_1 \rightarrow T_2$
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 $p(x,y)$	then subtype(T ₁ , S ₁) \land subtype(S ₂ , T ₂) else if S = {k _j :S _j ^{j∈1m} } and T = {1 _i :T _i ^{i∈1n} } then {1 _i ^{i∈1n} } \subseteq {k _j ^{j∈1m} } \land for all $i \in 1n$ there is some $j \in 1m$ with k _j = 1 _i and subtype(S _i , T _i)
whose graph is	else <i>false</i> .
{ ((1,2), true), ((2,3), true), ((1,3), false)}	To show this, we need to prove: 1. that it returns <i>true</i> whenever $S \leq T$, and
is <i>not</i> a decision procedure for R .	 2. that it returns either <i>true</i> or <i>false</i> on all inputs.

Subtyping Algorithm

But this recursively defined *partial* function is not:

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

Algorithmic Typing

lssue

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

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$
(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.

Example (T-SUB with T-ABS)

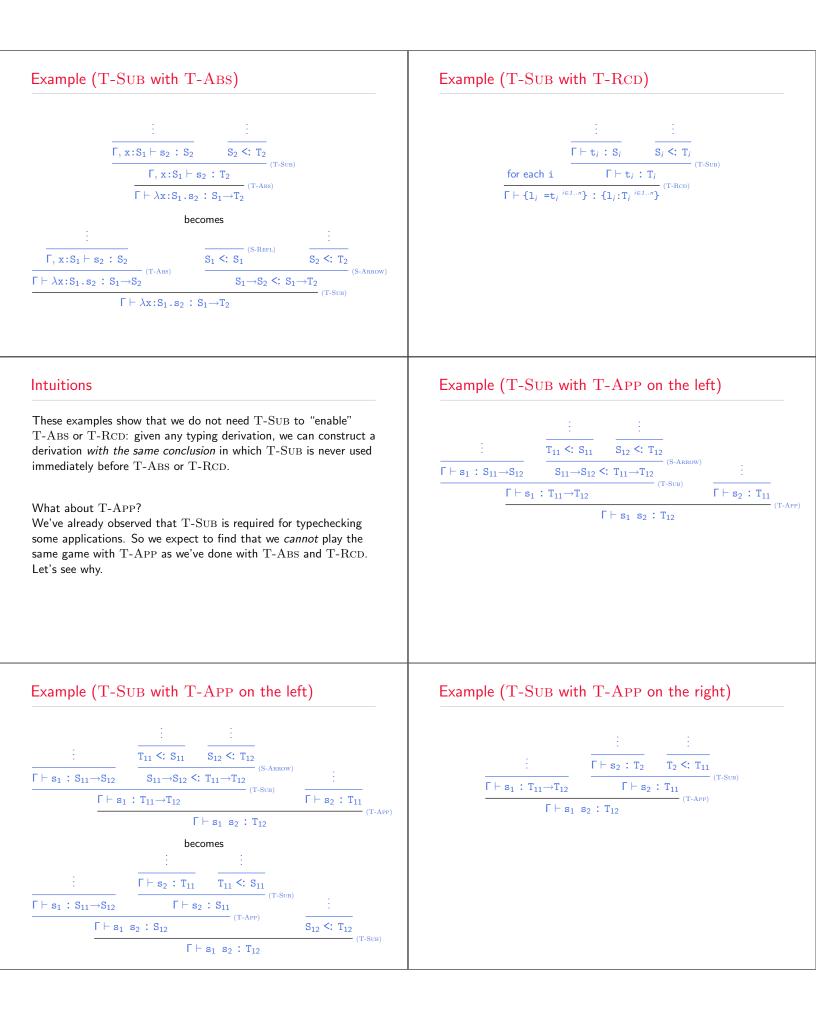
$$\frac{\overbrace{\Gamma, \mathbf{x}: S_1 \vdash \mathbf{s}_2 : S_2}^{\vdots}}{\frac{\Gamma, \mathbf{x}: S_1 \vdash \mathbf{s}_2 : \mathbf{z}_2}{\Gamma \vdash \lambda \mathbf{x}: S_1 \cdot \mathbf{s}_2 : \mathbf{z}_2}} \xrightarrow{(\text{T-SUB})}_{\text{(T-ABS)}}$$

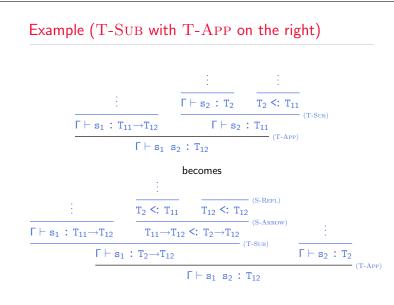
Algorithmic typing

- How do we implement a type checker for the lambda-calculus with subtyping?
- Given a context Γ and a term t, how do we determine its type T, such that $\Gamma \vdash t : T$?

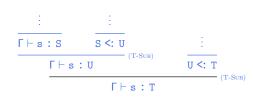
Plan

- 1. 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 (nested uses of $\mathrm{T}\text{-}\mathrm{SuB})$



Summary

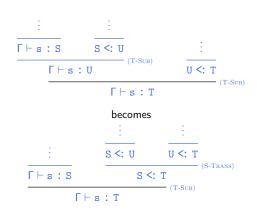
What we've learned:

- \blacktriangleright Uses of the $T\mathchar`-Sub}$ rule can be "pushed down" through typing derivations until they encounter either
 - 1. a use of $\operatorname{T-APP}$ or
 - 2. the root fo the derivation tree.
- ► In both cases, multiple uses of T-SUB can be collapsed into a single one.

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)



Summary

What we've learned:

- \blacktriangleright Uses of the $T\mathchar`SuB$ rule can be "pushed down" through typing derivations until they encounter either
 - 1. a use of $T\text{-}A\operatorname{PP}$ 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

- \blacktriangleright exactly one use of $T\text{-}S\mathrm{UB}$ before each use of $T\text{-}A\mathrm{PP}$
- \blacktriangleright one use of $T\mathchar`-Sub at the very end of the derivation$
- \blacktriangleright 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 $\rm T\text{-}APP$ rule to incorporate a subtyping premise.

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

Given any typing derivation, we can now

- 1. 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!

Final Algorithmic Typing Rules

$\frac{\mathbf{x}:T\in\Gamma}{\Gamma\models\mathbf{x}:T}$	(TA-VAR)
$\frac{\Gamma, \mathbf{x}: \mathbf{T}_1 \models \mathbf{t}_2 : \mathbf{T}_2}{\Gamma \models \lambda \mathbf{x}: \mathbf{T}_1 \cdot \mathbf{t}_2 : \mathbf{T}_1 \rightarrow \mathbf{T}_2}$	(TA-Abs)
$\label{eq:relation} {\textstyle \Gamma} \blacktriangleright {\tt t}_1 : {\tt T}_1 \qquad {\tt T}_1 = {\tt T}_{11} {\rightarrow} {\tt T}_{12} \qquad {\textstyle \Gamma} \trianglerighteq {\tt t}_2 : {\tt T}_2$	▶ T ₂ <: T ₁₁
$F \models t_1 \ t_2 : T_{12}$	(TA-App)
for each $i \Gamma \models t_i : T_i$ $\overline{\Gamma \models \{l_1 = t_1 \dots l_n = t_n\}} : \{l_1 : T_1 \dots l_n\}$	T_n (TA-RCD)
$\frac{\Gamma \models \mathbf{t}_1 : \mathbf{R}_1 \qquad \mathbf{R}_1 = \{\mathbf{l}_1 : \mathbf{T}_1 \dots \mathbf{l}_n : \mathbf{T}_n \\ \Gamma \models \mathbf{t}_1 , \mathbf{l}_i : \mathbf{T}_i \end{cases}$	-) (TA-Proj)

Completeness of the algorithmic rules

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

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.

Soundness of the algorithmic rules

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

Completeness of the algorithmic rules

Theorem [Minimal Typing]: If $\Gamma \vdash t : T$, then $\Gamma \models t : S$ 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.)



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.

$$\begin{array}{c|c} \Gamma \vdash \texttt{true} : \texttt{Bool} & (T-TRUE) \\ \Gamma \vdash \texttt{false} : \texttt{Bool} & (T-FALSE) \\ \hline \\ \hline \Gamma \vdash \texttt{t}_1 : \texttt{Bool} & \Gamma \vdash \texttt{t}_2 : T & \Gamma \vdash \texttt{t}_3 : T \\ \hline \\ \hline \\ \hline \\ \hline \\ \Gamma \vdash \texttt{if} \ \texttt{t}_1 \ \texttt{then} \ \texttt{t}_2 \ \texttt{else} \ \texttt{t}_3 : T \end{array}$$
 (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{\[f] \models t_1 : Bool \qquad \[f] \models t_2 : T_2 \qquad \[f] \models t_3 : T_3 \]}{\[f] \models if t_1 then t_2 else t_3 : T_2 \lor T_3} \quad (T-IF)$$

Existence of Joins

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

- 1. S <: J
- 2. T <: J
- 3. If K is a type such that $S \leq K$ and $T \leq K$, then $J \leq K$.
- I.e., J is the smallest type that is a supertype of both S and T.

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{\[f \models t_1 : Bool \qquad \[f \models t_2 : T_2 \qquad \[f \models t_3 : T_3 \]}{\[f \models if t_1 then t_2 else t_3 : T_2 \lor T_3 \]} \quad (T-IF)$$

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

Examples	Meets
<pre>What are the joins of the following pairs of types? 1. {x:Bool,y:Bool} and {y:Bool,z:Bool}? 2. {x:Bool} and {y:Bool}? 3. {x:{a:Bool,b:Bool}} and {x:{b:Bool,c:Bool}, y:Bool}? 4. {} and Bool? 5. {x:{}} and {x:Bool}? 6. Top→{x:Bool} and Top→{y:Bool}? 7. {x:Bool}→Top and {y:Bool}→Top?</pre>	To calculate joins of arrow types, we also need to be able to calculate <i>meets</i> (greatest lower bounds)! Unlike joins, meets do not necessarily exist. E.g., Bool→Bool and {} have <i>no</i> common subtypes, so they certainly don't have a greatest one! However
Existence of Meets	Examples
Theorem: For every pair of types S and T, if there is any type N such that $\mathbb{N} \leq \mathbb{S}$ and $\mathbb{N} \leq \mathbb{T}$, then there is a type M such that	What are the meets of the following pairs of types? 1. {x:Bool,y:Bool} and {y:Bool,z:Bool}?
 M <: S M <: T If 0 is a type such that 0 <: S and 0 <: T, then 0 <: M. I.e., M (when it exists) is the largest type that is a subtype of both S and T. 	 {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}?

Calculating Joins

$$S \lor T = \begin{cases} Bool & \text{if } S = T = Bool \\ M_1 \rightarrow J_2 & \text{if } S = S_1 \rightarrow S_2 \quad T = T_1 \rightarrow T_2 \\ & S_1 \land T_1 = M_1 \quad S_2 \lor T_2 = J_2 \\ \{j_l: J_l \ ^{l \in 1..q}\} & \text{if } S = \{k_j: S_j \ ^{j \in 1..m}\} \\ & T = \{l_i: T_i \ ^{i \in 1..n}\} \\ & \{j_l \ ^{l \in 1..q}\} = \{k_j \ ^{j \in 1..m}\} \cap \{1_i \ ^{i \in 1..n}\} \\ & S_j \lor T_i = J_l \text{ for each } j_l = k_j = l_i \\ & \text{Top} & \text{otherwise} \end{cases}$$

Calculating Meets

 $S \wedge T =$ S $\mathsf{if}\; T=\mathsf{Top}$ Т $\mathsf{if}\; S = \mathsf{Top}$ $\mathsf{if}\; \mathtt{S} = \mathtt{T} = \mathtt{Bool}$ Bool $\begin{array}{ll} \text{if } S = S_1 {\rightarrow} S_2 & T = T_1 {\rightarrow} T_2 \\ S_1 \lor T_1 = J_1 & S_2 \land T_2 = M_2 \end{array}$ $J_1 \rightarrow M_2$ $\{m_{l}: M_{l} \stackrel{l \in 1...q}{=} \{k_{j}: S_{j} \stackrel{j \in 1...m}{=} \}$ $\mathbf{T} = \{\mathbf{1}_{i}: \mathbf{T}_{i}^{i \in 1..n}\}$ $\{\mathbf{m}_{l} \mid l \in 1...q\} = \{\mathbf{k}_{j} \mid j \in 1...m\} \cup \{\mathbf{l}_{j} \mid i \in 1...n\}$ $S_j \wedge T_i = M_l$ for each $m_l = k_j = l_i$ if $m_l = k_j$ occurs only in S $M_I = S_j$ $M_I = T_i$ if $m_l = l_i$ occurs only in T f ail otherwise