CIS 500

Software Foundations

Fall 2002

6 November

`http://www.cis.upenn.edu/~sweirich/CIS500.ps`

In this class we'll turn the declarative version of subtyping into the algorithmic version.

The problem was that we don't have an algorithm to decide when $S <: T$ or $\Gamma \vdash t : T$. Both sets of rules are not syntax-directed.

Syntax-directed rules

When we say a set of rules is syntax-directed we mean two things:

1. There is exactly one rule in the set that applies to each syntactic form. (We can tell by the syntax of a term which rule to use.)
   - In order to derive a type for $t_1 t_2$, we must use T-App.
2. We don't have to "guess" an input (or output) for any rule.
   - To derive a type for $t_1 t_2$, we need to derive a type for t_1 and a type for t_2.

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \qquad \text{(T-App)}$$

16.1 Algorithmic Subtyping

How do we change the rules deriving $S <: T$ to be syntax-directed?

There are lots of ways to derive a given subtyping statement $S <: T$. The general idea is to change this system so that there is only one way to derive it.

Three rules for records

$$\{l_i : T_i^{i \in 1 \ldots n+k}\} <: \{l_i : T_i^{i \in 1 \ldots n}\} \qquad \text{(S-RcdWidth)}$$

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i^{i \in 1 \ldots n}\} <: \{l_i : T_i^{i \in 1 \ldots n}\}} \qquad \text{(S-RcdDepth)}$$

$$\frac{\{k_j : S_j^{i \in 1 \ldots n}\} \text{ is a permutation of } \{l_i : T_i^{i \in 1 \ldots n}\}}{\{k_j : S_j^{i \in 1 \ldots n}\} <: \{l_i : T_i^{i \in 1 \ldots n}\}} \qquad \text{(S-RcdPerm)}$$

Which rule do we use to decide if $\{k_j : S_j^{j \in 1 \ldots m}\} <: \{l_i : T_i^{i \in 1 \ldots n}\}$?

All-in-one

We can replace these three rules by a single rule that does permutation, width and depth subtyping all at once.

$$\frac{\begin{array}{c}\{l_i^{i \in 1 \ldots n}\} \subseteq \{k_j^{j \in 1 \ldots m}\} \\ k_j = l_i \text{ implies } S_j <: T_i\end{array}}{\{k_j : S_j^{j \in 1 \ldots m}\} <: \{l_i : T_i^{i \in 1 \ldots n}\}} \qquad \text{(S-Rcd)}$$

Lemma 1 *If $S <: T$ is provable with the 3 separate rules for width, depth and permutation, it is provable with just* S-Rcd.

Other rules

$$S <: S \qquad \text{(S-Refl)}$$

$$\frac{S <: U \quad U <: T}{S <: T} \qquad \text{(S-Trans)}$$

$$S <: Top \qquad \text{(S-Top)}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-Arrow)}$$

What other rules cause difficulty?

Non-syntax-directedness of subtyping

$$S <: S \qquad \text{(S-Refl)}$$

$$\frac{S <: U \quad U <: T}{S <: T} \qquad \text{(S-Trans)}$$

- Both reflexivity and transitivity apply to many inputs, regardless of their syntactic form.
- Furthermore, transitivity requires that we "guess" a value for U. This metavariable appears in the premises of the rule (in an input position) but not in the conclusion.

Just drop them

It turns out that the rest of the subtyping rules are reflexive and transitive, without explicitly declaring that they are.

Lemma 2 *1. $S <: S$ can be derived for every type S without using* S-Refl.

2. If $S <: T$ is derivable, then it can be derived without using S-Trans.

We don't need these rules in our system.

Algorithmic Subtyping Rules

$$\mapsto S <: Top \qquad \text{(SA-Top)}$$

$$\frac{\mapsto T_1 <: S_1 \qquad \mapsto S_2 <: T_2}{\mapsto S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(SA-Arrow)}$$

$$\frac{\{l_i^{i\in 1\ldots n}\} \subseteq \{k_j^{j\in 1\ldots m}\} \quad k_j = l_i \text{ implies } \mapsto S_j <: T_i}{\mapsto \{k_j : S_j^{j\in 1\ldots m}\} <: \{l_i : T_i^{i\in 1\ldots n}\}} \qquad \text{(SA-Rcd)}$$

Soundness and Completeness

The algorithmic rules are sound if every statement derivable from the algorithmic rules is derivable from the declarative rules.

The algorithmic rules are complete if every statement derivable from the declarative rules is derivable from the algorithmic rules.

Proposition 3 $S <: T$ *iff* $\mapsto S <: T$.

Algorithm

```
subtype(S, T) =
  if T = Top then true
  else if S = S1 → S2 and T = T1 → T2
       then subtype(T1, S1) ∧ subtype(S2, T2)
  else if S = { k_j : S_j  j∈1...m } and T = { l_i : T_i  i∈1...n }
       then { l_i  i∈1...n } ⊆ { k_j  j∈1...m }
            ∧ for all i there is some
                 j ∈ 1...m with k_j = l_i
            and subtype (S_j, T_i)
  else false
```

Does this pseudocode match the algorithmic rules? Is this function total?

Why declarative rules?

Why not use the algorithmic rules as the "official definition" of subtyping?

This does not save us much work as we still need to know that the rules are reflexive and transitive when we use them.

16.2 Algorithmic Typing

The subsumption rule is not syntax directed:

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

1. This rule could be used for any t.
2. We have to guess T in the subgoal $S <: T$.

Where is subsumption used?

We can't just get rid of it. A term like

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

is not typeable without subsumption.

$$\dfrac{\dfrac{\vdots}{\vdash \lambda r.\ r.x : \{x{:}N\} \to N} \qquad \dfrac{\dfrac{\vdots}{\vdash \{x = 0, y = 1\} : \{x{:}N, y{:}N\}} \quad \overline{\{x{:}N, y{:}N\} <: \{x{:}N\}}\ \text{(S-Sub)}}{\vdash \{x = 0, y = 1\} : \{x{:}N\}}}{\vdash (\lambda r{:}\{x{:}N\}.\ r.x)\{x = 0, y = 1\} : N}\ \text{(S-App)}$$

Do we need subsumption anywhere else?

Combine T-Sub with T-App

- Application is the only situation where subsumption is important.
- Why? It is the only rule where two types *must* match.
- Every other use of subsumption can be "postponed". If we use subsumption before any other rule, we can always rewrite the derivation so that subsumption is used after that rule.
- Therefore, we can incorporate subsumption with the application rule, and not lose any expressiveness.

Normalized derivation

Rewrite any derivation of $\Gamma \vdash t : T$ into a special form where T-Sub appears in only two places.

- Just before an application.
- At the very end of the derivation.

Pushing T-Sub around

If we have a (T-Sub) before a use of (T-Abs)

$$\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2} \quad \dfrac{\vdots}{S_2 <: T_2}\ \text{(T-Sub)}}{\dfrac{\Gamma, x{:}S_1 \vdash s_2 : T_2}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to T_2}\ \text{(T-Abs)}}$$

we can always switch the order of these rules with a little rearrangement.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to S_2}\ \text{(T-Abs)} \quad \dfrac{\dfrac{}{S_1 <: S_1} \quad \dfrac{\vdots}{S_2 <: T_2}}{S_1 \to S_2 <: S_1 \to T_2}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to T_2}\ \text{(T-Sub)}$$

Other rules

There are similar transformations for T-Sub followed by T-Rcd and for T-Sub followed by T-Proj.

T-Sub followed by T-Sub?

We can combine two uses of (T-Sub) together.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \quad \dfrac{\vdots}{S <: U}}{\Gamma \vdash s : U}\ \text{(T-Sub)} \quad \dfrac{\vdots}{U <: T}}{\Gamma \vdash s : T}\ \text{(T-Sub)}$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \quad \dfrac{\dfrac{\vdots}{S <: U} \quad \dfrac{\vdots}{U <: T}}{S <: T}\ \text{(S-Trans)}}{\Gamma \vdash s : T}\ \text{(T-Sub)}$$

(This is why $S <: T$ must be transitive.)

New Application Rule

$$\frac{\Gamma \mapsto t_1 : T_{11} \to T_{12} \quad \Gamma \mapsto t_2 : T_2 \quad \mapsto T_2 <: T_{11}}{\Gamma \mapsto t_1 t_2 : T_{12}} \text{ (TA-App)}$$

We can use subsumption for the type of the argument.

Replacing T-Sub with TA-App (RHS)

$$\frac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11} \to T_{12}} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash s_2 : T_2} \quad \dfrac{\vdots}{T_2 <: T_{11}}}{\Gamma \vdash s_2 : T_{11}} \text{ (T-Sub)}}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (T-App)}$$

$$\frac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11} \to T_{12}} \quad \dfrac{\vdots}{\Gamma \vdash s_2 : T_2} \quad \dfrac{\vdots}{T_2 <: T_{11}}}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (TA-App)}$$

T-Sub before T-App (LHS)

$$\frac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : S_{11} \to S_{12}} \quad \dfrac{\dfrac{\vdots}{T_{11} <: S_{11}} \quad \dfrac{\vdots}{S_{12} <: T_{12}}}{S_{11} \to S_{12} <: T_{11} \to T_{12}} \text{ (S-Arrow)}}{\Gamma \vdash s_1 : T_{11} \to T_{12}} \text{ (T-Sub)} \quad \dfrac{\vdots}{\Gamma \vdash s_2 : T_{11}}}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (T-App)}$$

$$\frac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : S_{11} \to S_{12}} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash s_2 : T_{11}} \quad \dfrac{\vdots}{T_{11} <: S_{11}}}{\Gamma \vdash s_2 : S_{11}} \text{ (T-Sub)}}{\Gamma \vdash s_1 s_2 : S_{12}} \text{ (T-App)} \quad \dfrac{\vdots}{S_{12} <: T_{12}}}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (T-Sub)}$$

$$\frac{x : T \in \Gamma}{\Gamma \mapsto x : T} \text{ (TA-Var)}$$

$$\frac{\Gamma, x{:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda x{:}T_1.t_2 : T_1 \to T_2} \text{ (TA-Abs)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \quad \Gamma \mapsto t_2 : T_2 \quad T_1 = T_{11} \to T_{12} \quad \mapsto T_2 <: T_{11}}{\Gamma \mapsto t_1 t_2 : T_{12}} \text{ (TA-App)}$$

$$\frac{\text{for each } i \quad \Gamma \mapsto t_i : T_i}{\Gamma \mapsto \{l_i = t_i \ldots l_n = t_n\} : \{l_i : T_1 \ldots l_n : T_n\}} \text{ (TA-Rcd)}$$

$$\frac{\Gamma : t_i : R \quad R = \{l_1 : T_1 \ldots l_n : T_n\}}{\Gamma \mapsto t_1.l_i : T_i} \text{ (TA-Proj)}$$

Soundness and Completeness

As before, we need to argue that the algorithmic rules are sound and complete with respect to the declarative rules.

Lemma 4 (Soundness) *If* $\Gamma \mapsto\!\blacktriangleright t : T$ *then* $\Gamma \vdash t : T$.

We can't prove the straightforward converse of the above lemma because while the declarative rules could assign many types to t, the algorithmic rules assign only one. For completeness, we can prove that the algorithmic rules give us the smallest or (minimal) possible type.

Lemma 5 (Completeness) *If* $\Gamma \vdash t : T$ *then* $\Gamma \mapsto\!\blacktriangleright t : S$ *for some* $S <: T$.

16.3 Joins and Meets

If we have conditionals or case expressions, we need additional machinery to support algorithmic subtyping.

$$\frac{\Gamma \vdash t_1 : \mathit{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathit{if}\ \ t_1\ \ \mathit{then}\ \ t_2\ \ \mathit{else}\ \ t_3 : T}$$

With subsumption, there may be many ways of giving the two branches the same type.

Algorithmic rule

Because the two branches must be the same type, we need to incorporate subsumption in an algorithmic version of this rule. How?

$$\frac{\Gamma \vdash t_1 : \mathit{Bool} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_2 <: T \quad T_3 <: T}{\Gamma \vdash \mathit{if}\ \ t_1\ \ \mathit{then}\ \ t_2\ \ \mathit{else}\ \ t_3 : T}$$

However, with this rule, we may not produce the minimal type for the expression. We can choose any T that is greater than T_2 and T_3.

We need to restrict this rule so that it produces the least such T. This T is called the join of T_2 and T_3.

Joins

Definition 6 *A type J is the join of two types S and T (and is written $J = S \vee T$) if*

1. $S <: J$
2. $T <: J$
3. *For all U, if $S <: U$ and $T <: U$ then $J <: U$.*

A join is a generalization of a *least upper bound*.

Meets

Definition 7 *A type M is the meet of two types S and T (and is written $J = S \wedge T$) if*

1. $M <: S$
2. $M <: J$
3. *For all U, if $U <: S$ and $U <: T$ then $U <: M$.*

A meet is a generalization of a *greatest lower bound*.

Note: do not confuse joins and meets with the intersection types and union types that we saw on Monday.

Existence of Joins and Meets

Given a subtype relation, it may or may not be the case that joins and meets exist for every pair of types.

- In fact, the subtype relation does not *have meets*. For example, there is no meet for the types $\{\}$ and $Top \to Top$.

Proposition 8 (Joins Exist) *For every pair of types S and T, there is some type J such that $S \vee T = J$.*

Proposition 9 (Bounded Meets Exist) *For every pair of types S and T with a common subtype, there is some type M such that $S \wedge T = M$.*

Algorithmic rule

Using join, we can give an algorithmic rules for *if*.

$$\frac{\Gamma \mapsto t_1 : Bool \quad \Gamma \mapsto t_2 : T_2 \quad \Gamma \mapsto t_3 : T_3 \quad T_2 \vee T_3 = T}{\Gamma \mapsto \mathit{if}\ \ t_1\ \ \mathit{then}\ \ t_2\ \ \mathit{else}\ \ t_3 : T}$$

16.4 Algorithmic typing and Bottom

$$\mapsto Bot <: T \qquad \text{(SA-Bot)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \quad T_1 = Bot \quad \Gamma \vdash t_2 : T_2}{\Gamma \mapsto t_1 t_2 : Bot} \qquad \text{(TA-AppBot)}$$

$$\frac{\Gamma \mapsto t_1 : R \quad R = Bot}{\Gamma \mapsto t_1.l_i : Bot} \qquad \text{(TA-ProjBot)}$$

In a declarative system, we can apply something of type Bot to an argument of absolutely any type (by using subsumption to promote the Bot to whatever function type we like), and assume that the result has any other type.