Formal Methods

Lecture 11

(B. Pierce's slides for the book "Types and Programming Languages")

Developing an algorithmic subtyping relation

Subtype relation

$$\texttt{S} <: \texttt{S} \tag{S-Refl}$$

$$\frac{\texttt{S} <: \texttt{U} \qquad \texttt{U} <: \texttt{T}}{\texttt{S} <: \texttt{T}} \tag{S-Trans}$$

$$\{\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i\in 1..n+k}\} <: \{\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i\in 1..n}\} \tag{S-RcdWidth}$$

$$\frac{\text{for each } i \qquad \texttt{S}_i <: \texttt{T}_i}{\{\texttt{l}_i\texttt{:}\texttt{S}_i\ ^{i\in 1..n}\} <: \{\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i\in 1..n}\}} \tag{S-RcdDepth}$$

$$\frac{\{\texttt{k}_j\texttt{:}\texttt{S}_j\ ^{j\in 1..n}\} \text{ is a permutation of } \{\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i\in 1..n}\}}{\{\texttt{k}_j\texttt{:}\texttt{S}_j\ ^{j\in 1..n}\} <: \{\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i\in 1..n}\}} \tag{S-RcdPerm}$$

$$\frac{\texttt{T}_1 <: \texttt{S}_1 \qquad \texttt{S}_2 <: \texttt{T}_2}{\texttt{S}_1 \rightarrow \texttt{S}_2 <: \texttt{T}_1 \rightarrow \texttt{T}_2} \tag{S-Arrow}$$

$$\texttt{S} <: \texttt{Top} \tag{S-Top}$$

Issues

For a given subtyping statement, there are multiple rules that could be used last in a derivation.

1. The conclusions of S-RcdWidth, S-RcdDepth, and S-RcdPerm overlap with each other.
2. S-Refl and S-Trans overlap with every other rule.

Step 1: simplify record subtyping

Idea: combine all three record subtyping rules into one "macro rule" that captures all of their effects

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

Simpler subtype relation

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

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

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

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

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

Step 2: Get rid of reflexivity

Observation: S-Refl is unnecessary.

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

Even simpler subtype relation

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

$$\frac{\{\mathtt{l}_i\ ^{i \in 1..n}\} \subseteq \{\mathtt{k}_j\ ^{j \in 1..m}\} \qquad \mathtt{k}_j = \mathtt{l}_i \text{ implies } \mathtt{S}_j <: \mathtt{T}_i}{\{\mathtt{k}_j : \mathtt{S}_j\ ^{j \in 1..m}\} <: \{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\}} \qquad \text{(S-Rcd)}$$

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(S-Arrow)}$$

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

Step 3: Get rid of transitivity

Observation: S-Trans is unnecessary.

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

"Algorithmic" subtype relation

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

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

$$\frac{\{l_i\ ^{i \in 1..n}\} \subseteq \{k_j\ ^{j \in 1..m}\} \quad \text{for each } k_j = l_i,\ \vdash_{\blacktriangle} S_j <: T_i}{\vdash_{\blacktriangle} \{k_j : S_j\ ^{j \in 1..m}\} <: \{l_i : T_i\ ^{i \in 1..n}\}} \qquad \text{(SA-Rcd)}$$

Soundness and completeness

Theorem: $S <: T$ iff $\vdash_{\blacktriangle} S <: T$.

Proof: *(Homework)*

Terminology:

- The algorithmic presentation of subtyping is *sound* with respect to the original if $\vdash_{\blacktriangle} S <: T$ implies $S <: T$. (Everything validated by the algorithm is actually true.)
- The algorithmic presentation of subtyping is *complete* with respect to the original if $S <: T$ implies $\vdash_{\blacktriangle} S <: T$. (Everything true is validated by the algorithm.)

Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

$\mathit{subtype}(S, T)$ =

if $T = Top$, then *true*

else if $S = S_1 \to S_2$ and $T = T_1 \to T_2$

then $\mathit{subtype}(T_1, S_1) \wedge \mathit{subtype}(S_2, T_2)$

else if $S = \{k_j : S_j\ ^{j \in 1..m}\}$ and $T = \{l_i : T_i\ ^{i \in 1..n}\}$

then $\{l_i\ ^{i \in 1..n}\} \subseteq \{k_j\ ^{j \in 1..m}\}$

$\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $k_j = l_i$

and $\mathit{subtype}(S_j, T_i)$

else *false*.

Decision Procedures

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

Is our *subtype* function a decision procedure?

Decision Procedures

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

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

1. if $subtype(S, T) = true$, then $\vdash_{\blacktriangle} S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)
2. if $subtype(S, T) = false$, then not $\vdash_{\blacktriangle} S <: T$
   (hence, by completeness of the algorithmic rules, not $S <: T$)

Decision Procedures

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

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

1. if $subtype(S, T) = true$, then $\vdash_{\blacktriangle} S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)
2. if $subtype(S, T) = false$, then not $\vdash_{\blacktriangle} S <: T$
   (hence, by completeness of the algorithmic rules, not $S <: T$)

Q: What's missing?

Decision Procedures

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

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

1. if $subtype(\mathrm{S}, \mathrm{T}) = true$, then $\vdash_{\blacktriangle} \mathrm{S} <: \mathrm{T}$
   (hence, by soundness of the algorithmic rules, $\mathrm{S} <: \mathrm{T}$)
2. if $subtype(\mathrm{S}, \mathrm{T}) = false$, then not $\vdash_{\blacktriangle} \mathrm{S} <: \mathrm{T}$
   (hence, by completeness of the algorithmic rules, not $\mathrm{S} <: \mathrm{T}$)

Q: What's missing?

A: How do we know that *subtype* is a *total* function?

Prove it!

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 *computability*. 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:

$$\begin{aligned} U &= \{1, 2, 3\} \\ R &= \{(1, 2),\ (2, 3)\} \end{aligned}$$

The function p whose graph is

$$\begin{aligned} \{ \ &((1, 2), true),\ ((2, 3), true), \\ &((1, 1), false),\ ((1, 3), false), \\ &((2, 1), false),\ ((2, 2), false), \\ &((3, 1), false),\ ((3, 2), false),\ ((3, 3), false)\} \end{aligned}$$

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:

$$\begin{aligned} U &= \{1, 2, 3\} \\ R &= \{(1, 2),\ (2, 3)\} \end{aligned}$$

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:

$$\begin{aligned} U &= \{1, 2, 3\} \\ R &= \{(1, 2),\ (2, 3)\} \end{aligned}$$

The function p'' whose graph is

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

is also *not* a decision function for R.

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 $u \in R$.

Example

$$
\begin{aligned}
U &= \{1, 2, 3\} \\
R &= \{(1, 2),\ (2, 3)\}
\end{aligned}
$$

Example

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

The function

$$p(x, y) = \begin{array}{l} \textit{if } x = 2 \textit{ and } y = 3 \textit{ then true} \\ \textit{else if } x = 1 \textit{ and } y = 2 \textit{ then true} \\ \textit{else false} \end{array}$$

whose graph is

$$\begin{array}{l} \{((1, 2), \textit{true}),\ ((2, 3), \textit{true}), \\ \quad ((1, 1), \textit{false}),\ ((1, 3), \textit{false}), \\ \quad ((2, 1), \textit{false}),\ ((2, 2), \textit{false}), \\ \quad ((3, 1), \textit{false}),\ ((3, 2), \textit{false}),\ ((3, 3), \textit{false})\} \end{array}$$

is a decision procedure for R.

Example

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

The recursively defined partial function

$$\begin{aligned} p(x, y) = \ & \mathit{if}\ \ x = 2\ \mathit{and}\ y = 3\ \mathit{then\ true} \\ & \mathit{else\ if}\ \ x = 1\ \mathit{and}\ y = 2\ \mathit{then\ true} \\ & \mathit{else\ if}\ \ x = 1\ \mathit{and}\ y = 3\ \mathit{then\ false} \\ & \mathit{else}\ p(x, y) \end{aligned}$$

whose graph is

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

is *not* a decision procedure for R.

Subtyping Algorithm

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

$subtype(S, T)$ =

if $T = Top$, then *true*

else if $S = S_1 \rightarrow S_2$ and $T = T_1 \rightarrow T_2$

then $subtype(T_1, S_1) \wedge subtype(S_2, T_2)$

else if $S = \{k_j : S_j\ ^{j \in 1..m}\}$ and $T = \{l_i : T_i\ ^{i \in 1..n}\}$

then $\{l_i\ ^{i \in 1..n}\} \subseteq \{k_j\ ^{j \in 1..m}\}$

$\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $k_j = l_i$ and $subtype(S_j, T_i)$

else *false*.

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) =

if $T = Top$, then *true*

else if $S = S_1 \rightarrow S_2$ and $T = T_1 \rightarrow T_2$

then *subtype*(T_1, S_1) $\wedge$ *subtype*(S_2, T_2)

else if $S = \{k_j : S_j \;^{j \in 1..m}\}$ and $T = \{l_i : T_i \;^{i \in 1..n}\}$

then $\{l_i \;^{i \in 1..n}\} \subseteq \{k_j \;^{j \in 1..m}\}$

$\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $k_j = l_i$

and *subtype*(S_j, T_i) else

subtype(T, S)

Algorithmic Typing

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 $\Gamma \vdash t : T$?

Issue

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

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

Where is this rule really needed?

Issue

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

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

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else??

Issue

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

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

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else?? *Nowhere else!*

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

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 (T-Abs)

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

Example (T-Abs)

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

becomes

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma, x:S_1 \vdash s_2 : S_2}}{\Gamma \vdash \lambda x:S_1.s_2 : S_1 \to S_2}\;\text{(T-Abs)} \qquad \cfrac{\cfrac{}{S_1 <: S_1}\;\text{(S-Refl)} \qquad \cfrac{\vdots}{S_2 <: T_2}}{S_1 \to S_2 <: S_1 \to T_2}\;\text{(S-Arrow)}}{\Gamma \vdash \lambda x:S_1.s_2 : S_1 \to T_2}\;\text{(T-Sub)}$$

Example (T-Sub with T-Rcd)

$$
\dfrac{\text{for each } i \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash t_i : S_i} \qquad \dfrac{\vdots}{S_i <: T_i}}{\Gamma \vdash t_i : T_i}\ \text{(T-Sub)}}{\Gamma \vdash \{l_i = t_i\ ^{i \in 1..n}\} : \{l_i : T_i\ ^{i \in 1..n}\}}\ \text{(T-Rcd)}
$$

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-Sub is required for typechecking some applications. So we expect to find that we *cannot* play the same game with T-App as we've done with T-Abs and T-Rcd. Let's see why.

Example (T-App on the left)

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

Example (T-App on the left)

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

$$
\dfrac{\Gamma \vdash s_1 : T_{11}\to T_{12} \qquad \dfrac{\vdots}{\Gamma \vdash s_2 : T_{11}}}{\Gamma \vdash s_1\ s_2 : T_{12}}\ (\text{T-App})
$$

becomes

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

Example (T-App on the right)

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

Example ($\mathrm{T\text{-}App}$ on the right)

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

becomes

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11}\to T_{12}} \qquad \dfrac{\dfrac{\vdots}{T_2 <: T_{11}} \qquad \dfrac{}{T_{12} <: T_{12}}\ (\textsc{S-Refl})}{T_{11}\to T_{12} <: T_2\to T_{12}}\ (\textsc{S-Arrow})}{\Gamma \vdash s_1 : T_2\to T_{12}}\ (\textsc{T-Sub}) \qquad \dfrac{\vdots}{\Gamma \vdash s_2 : T_2}}{\Gamma \vdash s_1\ s_2 : T_{12}}\ (\textsc{T-App})$$

Intuitions

So we’ve seen that uses of subsumption can be “pushed” from one of immediately before $\mathrm{T\text{-}App}$’s premises to the other, but cannot be completely eliminated.

Example (nested uses of T-Sub)

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

Example (nested uses of T-Sub)

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

becomes

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

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 for 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 for 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.

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_2 \qquad \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

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!

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.

Minimal Types

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

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{x} : \mathtt{T}} \qquad \text{(TA-Var)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(TA-Abs)}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 = \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2 \qquad \mathrel{|\!\!\blacktriangleright} \mathtt{T}_2 <: \mathtt{T}_{11}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(TA-App)}$$

$$\frac{\text{for each } i \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_i : \mathtt{T}_i}{\Gamma \mathrel{|\!\!\blacktriangleright} \{\mathtt{l}_1\mathtt{=}\mathtt{t}_1 \ldots \mathtt{l}_n\mathtt{=}\mathtt{t}_n\} : \{\mathtt{l}_1:\mathtt{T}_1 \ldots \mathtt{l}_n:\mathtt{T}_n\}} \qquad \text{(TA-Rcd)}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1 : \mathtt{R}_1 \qquad \mathtt{R}_1 = \{\mathtt{l}_1:\mathtt{T}_1 \ldots \mathtt{l}_n:\mathtt{T}_n\}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1.\mathtt{l}_i : \mathtt{T}_i} \qquad \text{(TA-Proj)}$$

Soundness of the algorithmic rules

Theorem: If $\Gamma \vdash_{\blacktriangle} t : T$, then $\Gamma \vdash t : T$.

Completeness of the algorithmic rules

Theorem [Minimal Typing]: If $\Gamma \vdash t : T$, then $\Gamma \vdash_{\blacktriangle} t : S$ for some $S <: T$.

Completeness of the algorithmic rules

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

Proof: Induction on typing derivation.

(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.

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T} \qquad \text{(T-If)}$$

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

$$\text{if } t_1 \text{ then } t_2 \text{ else } t_3$$

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.

$$\frac{\Gamma \vdash_{\blacktriangle} t_1 : \text{Bool} \qquad \Gamma \vdash_{\blacktriangle} t_2 : T_2 \qquad \Gamma \vdash_{\blacktriangle} t_3 : T_3}{\Gamma \vdash_{\blacktriangle} \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \vee T_3} \quad \text{(T-If)}$$

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 $S <: K$ 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?

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?

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

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* <u>bounded</u> *meets*

Examples

What are the meets 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?

Calculating Joins

$$
\mathtt{S} \vee \mathtt{T} \;=\; \begin{cases}
\mathtt{Bool} & \text{if } \mathtt{S} = \mathtt{T} = \mathtt{Bool} \\
\mathtt{M}_1 \!\to\! \mathtt{J}_2 & \text{if } \mathtt{S} = \mathtt{S}_1 \!\to\! \mathtt{S}_2 \quad \mathtt{T} = \mathtt{T}_1 \!\to\! \mathtt{T}_2 \\
 & \quad \mathtt{S}_1 \wedge \mathtt{T}_1 = \mathtt{M}_1 \quad \mathtt{S}_2 \vee \mathtt{T}_2 = \mathtt{J}_2 \\
\{\mathtt{j}_l \mathtt{:} \mathtt{J}_l \;^{l \in 1..q}\} & \text{if } \mathtt{S} = \{\mathtt{k}_j \mathtt{:} \mathtt{S}_j \;^{j \in 1..m}\} \\
 & \quad \mathtt{T} = \{\mathtt{l}_i \mathtt{:} \mathtt{T}_i \;^{i \in 1..n}\} \\
 & \quad \{\mathtt{j}_l \;^{l \in 1..q}\} = \{\mathtt{k}_j \;^{j \in 1..m}\} \cap \{\mathtt{l}_i \;^{i \in 1..n}\} \\
 & \quad \mathtt{S}_j \vee \mathtt{T}_i = \mathtt{J}_l \quad \text{for each } \mathtt{j}_l = \mathtt{k}_j = \mathtt{l}_i \\
\mathtt{Top} & \text{otherwise}
\end{cases}
$$

Calculating Meets

$$\mathtt{S} \wedge \mathtt{T} \quad =$$

$$\begin{cases}
\mathtt{S} & \text{if } \mathtt{T} = \mathtt{Top} \\
\mathtt{T} & \text{if } \mathtt{S} = \mathtt{Top} \\
\mathtt{Bool} & \text{if } \mathtt{S} = \mathtt{T} = \mathtt{Bool} \\
\mathtt{J}_1 \rightarrow \mathtt{M}_2 & \text{if } \mathtt{S} = \mathtt{S}_1 \rightarrow \mathtt{S}_2 \quad \mathtt{T} = \mathtt{T}_1 \rightarrow \mathtt{T}_2 \\
 & \quad \mathtt{S}_1 \vee \mathtt{T}_1 = \mathtt{J}_1 \quad \mathtt{S}_2 \wedge \mathtt{T}_2 = \mathtt{M}_2 \\
\{\mathtt{m}_l : \mathtt{M}_l \ ^{l \in 1..q}\} & \text{if } \mathtt{S} = \{\mathtt{k}_j : \mathtt{S}_j \ ^{j \in 1..m}\} \\
 & \quad \mathtt{T} = \{\mathtt{l}_i : \mathtt{T}_i \ ^{i \in 1..n}\} \\
 & \quad \{\mathtt{m}_l \ ^{l \in 1..q}\} = \{\mathtt{k}_j \ ^{j \in 1..m}\} \cup \{\mathtt{l}_i \ ^{i \in 1..n}\} \\
 & \quad \mathtt{S}_j \wedge \mathtt{T}_i = \mathtt{M}_l \qquad \text{for each } \mathtt{m}_l = \mathtt{k}_j = \mathtt{l}_i \\
 & \quad \mathtt{M}_l = \mathtt{S}_j \qquad \text{if } \mathtt{m}_l = \mathtt{k}_j \text{ occurs only in } \mathtt{S} \\
 & \quad \mathtt{M}_l = \mathtt{T}_i \qquad \text{if } \mathtt{m}_l = \mathtt{l}_i \text{ occurs only in } \mathtt{T} \\
fail & \text{otherwise}
\end{cases}$$