编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2026

Issues in Subtyping

Typing with Subsumption

Principle of safe substitution:

- *a value of one* can *always safely be used* where *a value of the other* is expected

1. a *subtyping relation* between types, written S <: T
2. a rule of *subsumption* stating that, if S <: T, then any value of type S can also be regarded as having type T, i.e.,

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

Issues in Subtyping

For a *given subtyping statement*, there are *multiple rules* that could be used 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*.

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

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

Syntax-directed rules

In the simply typed lambda-calculus (without subtyping), *each rule* can be "*read from bottom to top*" in a straightforward way.

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

If we are given some Γ and some t of the form $t_1\ t_2$, we can try to *find a type* for t by

1. finding (recursively) a type for t_1
2. checking that it has the form $T_{11} \longrightarrow T_{12}$
3. finding (recursively) a type for t_2
4. checking that it is the same as T_{11}

Syntax-directed rules

The first reason this works is that we can ***divide*** *the* "***positions***" of the typing relation ($\Gamma \vdash t : T$) into ***input positions*** (i.e., Γ and t) and ***output positions*** (T).

- For the input positions, all metavariables appearing in the *premises* also appear in the *conclusion* (so we can calculate inputs to the "*sub-goals*" from the sub-expressions of inputs to the main goal)
- For the output positions, all metavariables appearing in the *conclusions* also appear in the *premises* (so we can calculate outputs from the main goal from the outputs of the subgoals)

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

Syntax-directed sets of rules

The *second important point* about the simply typed lambda-calculus is that *the set of typing rules is syntax-directed*:

- For every "***input*** " Γ and t, *there is exactly one rule* that can be used to derive typing statements involving t, e.g.,

  if t is an *application*, then we must proceed by trying to use T-APP
- If we succeed, then we have found a type (indeed, the *unique type*) for t
- If it ***fails,*** then we know that t is ***not typable***

$\Longrightarrow$ no backtracking!

Non-syntax-directedness of typing

When the system is extended with *subtyping*, both aspects of syntax-directedness get broken.

1. The set of typing rules now includes *two rules* that can be used to give a type to terms of a given shape (*the old one* + T-SUB)

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

2. Worse yet, the new rule T-SUB *itself is not syntax directed*: the ***inputs*** to ***the left-hand sub-goal*** *are exactly the same* as the ***inputs*** to ***the main goal***

   Hence, if we translate the typing rules naively into a typechecking function, the case corresponding to T-SUB would cause *divergence*

Non-syntax-directedness of subtyping

Moreover, the *subtyping relation* is *not syntax directed* either

1. There are *lots of ways* to derive a given subtyping statement

   (∵ 8.2.4 /9.3.3 [uniqueness of types] ×)

2. The transitivity rule

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

is *badly non-syntax-directed*: the premises contain a *metavariable* (in an "*input position*") that does *not appear at all in the conclusion*.

To implement this rule naively, we have to *guess* a value for U!

What to do?

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*

Chap 16
Metatheory of Subtyping

Algorithmic Subtyping

Algorithmic Typing

Joins and Meets

Developing
an algorithmic
subtyping relation

Algorithmic Subtyping

What to do

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.

Step 1: simplify record subtyping

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

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

Lemma 16.1.1: If S <: T is derivable from the subtyping rules including S-RcdDepth, S-Rcd-Width, and S-Rcd-Perm (but not S-Rcd), then it can also be derived using S-Rcd (and not S-RcdDepth, S-Rcd-Width, or S-Rcd-Perm), and vice versa.

Simpler subtype relation

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

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

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

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

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

Step 2: Get rid of reflexivity

Observation: S-REFL is unnecessary.

Lemma 16.1.2: $S <: S$ can be derived for every type S without using S-REFL.

Even simpler subtype relation

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

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

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

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

Step 3: Get rid of transitivity

Observation: S-Trans is unnecessary.

Lemma 16.1.2: If $S <: T$ can be derived, then it can be derived without using S-Trans.

Even simpler subtype relation

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

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

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

"Algorithmic" subtype relation

Definition: The *algorithmic subtyping relation* is the least relation on types closed under the following 3 rules

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

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

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

Soundness and completeness

Theorem[16.1.5]: $S <: T$ iff $\mapsto S <: T$

Terminology:

- The *algorithmic presentation* of subtyping is *sound* with respect to the original, if $\mapsto S <: T$ 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 $\mapsto S <: T$

  (*Everything true* is *validated by the algorithm*)

Subtyping Algorithm

subtype(S, T) =

if $\mathrm{T} = \mathrm{Top}$, then *true*

else if $\mathrm{S} = \mathrm{S}_1 \longrightarrow \mathrm{S}_2$ and $\mathrm{T} = \mathrm{T}_1 \longrightarrow \mathrm{T}_2$

then $subtype(\mathrm{T}_1, \mathrm{S}_1) \wedge subtype(\mathrm{S}_2, \mathrm{T}_2)$

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

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

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

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?

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

1. if $subtype(\mathrm{S}, \mathrm{T}) = true$, then $\mapsto \mathrm{S} <: \mathrm{T}$

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

   hence, by completeness of the algorithmic rules, not $\mathrm{S} <: \mathrm{T}$

Q: What’s missing?

Decision Procedures

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 $\mapsto S <: T$

   (hence, by soundness of the algorithmic rules, $S <: T$)

1. if $subtype(S, T) = false$, then not $\mapsto S <: T$

   (hence, by completeness of the algorithmic rules, not $S <: T$)

Q: What's missing?

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

Decision Procedures

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 $\mapsto \mathrm{S} <: \mathrm{T}$

   (hence, by soundness of the algorithmic rules, $\mathrm{S} <: \mathrm{T}$)

1. if $subtype(\mathrm{S}, \mathrm{T}) = false$, then not $\mapsto \mathrm{S} <: \mathrm{T}$

   (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

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 $u \in R$.

Example:

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

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

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 $u \in R$.

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

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 $u \in R$.

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

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 $u \in R$.

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

Decision Procedures (take 2)

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

$$p(u) = true \text{ iff } u \in R.$$

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*

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

whose graph is

{ ((1, 2), *true*), ((2, 3), *true*), ((1, 3), *false*)}

is *not* a decision procedure for R.

Subtyping Algorithm

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

subtype(S, T) =

if $\mathrm{T} = \mathrm{Top}$ then *true*

else if $\mathrm{S} = \mathrm{S_1} \longrightarrow \mathrm{S_2}$ and $\mathrm{T} = \mathrm{T_1} \longrightarrow \mathrm{T_2}$

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

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

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

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

else *false*.

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*

[16.1.6 Termination Proposition]

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 rule*

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

Q: 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!

Uses of subsumption rule to help typecheck *applications* are the only interesting ones (where subsumption plays a crucial role in typing)

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 these examples 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)

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

becomes

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

Intuitions

These examples show that ***we do not need*** T-SUB ***to "enable"*** **T-ABS** :

- given any typing derivation, we **can construct a derivation** *with the same conclusion* **in which *T-SUB is never used immediately before* T-ABS.**

What about ***T-APP***?

We've already observed that ***T-SUB*** is required for typechecking some *applications*

Therefore we expect to find that we ***cannot*** play the same game with **T-APP** as we've done with **T-ABS**

Let's see why.

Example (T—Sub with T-APP **on the left**)

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

becomes

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

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

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

becomes

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

Observations

We've seen that ***uses of subsumption rule*** 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)

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

becomes

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

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 at the end of right-hand subderivations of T-App, or
2. the *root* of the derivation tree (the **very end of the whole derivation)**

– In both cases, ***multiple uses of*** T-Sub ***can be coalesced 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 rule 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 rule* 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

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

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

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

$$\frac{\text{for each } i \qquad \Gamma \mapsto t_i : T_i}{\Gamma \mapsto \{l_1{=}t_1 \ldots l_n{=}t_n\} : \{l_1{:}T_1 \ldots l_n{:}T_n\}} \quad \text{(TA-RCD)}$$

$$\frac{\Gamma \mapsto t_1 : R_1 \qquad R_1 = \{l_1{:}T_1 \ldots l_n{:}T_n\}}{\Gamma \mapsto t_1.l_i : T_i} \quad \text{(TA-PROJ)}$$

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*.

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

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

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{Bool} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T} \qquad \Gamma \vdash \texttt{t}_3 : \texttt{T}}{\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}} \qquad \text{(T-If)}$$

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

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

Q: 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.

How to prove it?

Calculating Joins

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

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. $Top \longrightarrow \{x: Bool\}$ and $Top \longrightarrow \{y: Bool\}$?
7. $\{x: Bool\} \longrightarrow Top$ and $\{y: Bool\} \longrightarrow Top$?

Meets

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

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., $\text{Bool} \longrightarrow \text{Bool}$ 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, we say that a type M is a meet of S and T, written $S \wedge T = M$ if

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*

Calculating Meets

$$
S \wedge T = \begin{cases}
S & \text{if } T = \mathtt{Top} \\
T & \text{if } S = \mathtt{Top} \\
\mathtt{Bool} & \text{if } S = T = \mathtt{Bool} \\
J_1 \to M_2 & \text{if } S = S_1 \to S_2 \quad T = T_1 \to T_2 \\
 & \quad S_1 \vee T_1 = J_1 \quad S_2 \wedge T_2 = M_2 \\
\{m_l : M_l \;^{l \in 1..q}\} & \text{if } S = \{k_j : S_j \;^{j \in 1..m}\} \\
 & \quad T = \{l_i : T_i \;^{i \in 1..n}\} \\
 & \quad \{m_l \;^{l \in 1..q}\} = \{k_j \;^{j \in 1..m}\} \cup \{l_i \;^{i \in 1..n}\} \\
 & \quad S_j \wedge T_i = M_l \qquad \text{for each } m_l = k_j = l_i \\
 & \quad M_l = S_j \qquad \text{if } m_l = k_j \text{ occurs only in } S \\
 & \quad M_l = T_i \qquad \text{if } m_l = l_i \text{ occurs only in } T \\
\mathit{fail} & \text{otherwise}
\end{cases}
$$

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. $Top \longrightarrow \{x: Bool\}$ and $Top \longrightarrow \{y: Bool\}$?
7. $\{x: Bool\} \longrightarrow Top$ and $\{y: Bool\} \longrightarrow Top$?

Homework☺

- Read and digest chapter 16 & 17

- HW:
  - 16.1.3;
  - 16.2.6;