Recap on Subtyping

Subsumption

Some types *are better* than others, in the sense that a value of one can *always safely be used* where a value of the other is expected.

Which can be formalized as by introducing:

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

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

Principle of safe substitution

Subtype Relation

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

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

$$\{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n+k}\} <: \{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\} \qquad \text{(S-RcdWidth)}$$

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

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

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

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

$\rightarrow$ <: Top — *Based on $\lambda_{\rightarrow}$ (9-1)*

Syntax

t ::=		*terms:*
	x	*variable*
	$\lambda x{:}T.t$	*abstraction*
	$t\ t$	*application*
v ::=		*values:*
	$\lambda x{:}T.t$	*abstraction value*
T ::=		*types:*
	Top	*maximum type*
	$T{\rightarrow}T$	*type of functions*
Γ ::=		*contexts:*
	$\varnothing$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

Subtyping $\boxed{S <: T}$

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

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

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

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

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1{\rightarrow}T_2} \qquad \text{(T-Abs)}$$

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

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

Records

$\rightarrow$ {} *Extends $\lambda_{\rightarrow}$ (9-1)*

New syntactic forms

$\mathtt{t} ::=$	...	*terms:*
	$\{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\}$	*record*
	$\mathtt{t.l}$	*projection*
$\mathtt{v} ::=$	...	*values:*
	$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}$	*record value*
$\mathtt{T} ::=$	...	*types:*
	$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}$	*type of records*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{.l} \longrightarrow \mathtt{t}_1'\mathtt{.l}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=t}_j\mathtt{,l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\\ \longrightarrow \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=t}_j'\mathtt{,l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Rcd)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1\mathtt{.l}_j : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

Records & Subtyping

$\rightarrow$ {} <: *Extends $\lambda_{<:}$ (15-1) and simple record rules (15-2)*

New subtyping rules $\boxed{S <: T}$

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

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

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

Properties of Subtyping

Safety

Do the Statements of progress and preservation theorems need change?

Statements of progress and preservation theorems are unchanged from $\lambda_{\rightarrow}$.

Safety

Statements of progress and preservation theorems are unchanged from $\lambda_{\rightarrow}$.

However, Proofs become a bit *more involved*, because the typing relation is no longer *syntax directed*.

Given a derivation, *we don't always know what rule was used* in the last step.

e.g., the rule T-SUB could appear anywhere.

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

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.

Preservation

Theorem: If $\Gamma \vdash \mathrm{t}: \mathrm{T}$ *and* $\mathrm{t} \longrightarrow \mathrm{t}'$, *then* $\Gamma \vdash \mathrm{t}' : \mathrm{T}$.

Proof: By induction on *typing derivations*.

Which cases are likely to be hard?

Subsumption case

Case T-Sub: $t : S \quad S <: T$

By the induction hypothesis, $\Gamma \vdash t' : S$.

By T-Sub , $\Gamma \vdash t' : T$.

Not hard!

Application case

Case T-App :

$$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$$

By the inversion lemma for evaluation, there are *three rules* by which $t \longrightarrow t'$ can be derived:

E-App1, E-App2, and E-AppAbs .

Proceed by cases.

Application case

Case T-App :

$$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$$

By the evaluation rules in Figure 15-1 and 15-2, there are *three rules* by which $t \longrightarrow t'$ can be derived:

E-App1, E-App2, and E-AppAbs.

Proceed by cases.

Subcase E-App1: $t_1 \longrightarrow t'_1 \qquad t' = t'_1\ t_2$

The result follows from the induction hypothesis and T-App.

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

Application case

Case T-App :

$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$

Subcase E-App2 : $t_1 = v_1 \quad t_2 \longrightarrow t'_2 \quad t' = v_1\ t'_2$

Similar.

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

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-APP2)}$$

Application case

Case T-App:

$$t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$$

Subcase E-AppAbs :

$$t_1 = \lambda x{:}S_{11}.t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]\, t_{12}$$

by the *inversion lemma* for the typing relation ...

$$T_{11} <: S_{11} \text{ and } \Gamma, x{:}S_{11} \vdash t_{12} : T_{12} \ .$$

By using T-Sub, $\Gamma \vdash t_2 : S_{11}$.

by the *substitution lemma*, $\Gamma \vdash t' : T_{12}$.

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

$$(\lambda x{:}T_{11}.t_{12}) \ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

Inversion Lemma for Typing

Lemma(15.3.3): $\quad If \ \ \Gamma \vdash \lambda x{:}\,S_1.\,s_2{:}\ T_1 \longrightarrow T_2,$ then

$$T_1 <: S_1 \quad \text{and} \quad \Gamma, x{:}\,S_1 \vdash s_2{:}\,T_2.$$

Proof: *Induction on typing derivations*.

Case T–Sub: $\lambda x{:}S_1.s_2{:}U \qquad U{:}\ T_1{\longrightarrow}T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (since we do not know that U is an arrow type).

Need another lemma...

Lemma (15.3.2): If $U <: T_1 \longrightarrow T_2$, then U has the form of $U_1 \longrightarrow U_2$,

with $T_1 <: U_1$ and $U_2 <: T_2$.

(*Proof*: by *induction on subtyping derivations*.)

Inversion Lemma for Typing

By this lemma, we know

$$U = U_1 \longrightarrow U_2, \text{ with } T_1 <: U_1 \text{ and } U_2 <: T_2.$$

The IH now applies, yielding

$$U_1 <: S_1 \quad \text{and} \quad \Gamma, x{:}\, S_1 \vdash s_2 : U_2.$$

From $U_1 <: S_1$ and $T_1 <: U_1$, rule S-Trans gives

$$T_1 <: S_1.$$

From $\Gamma, x{:}\, S_1 \vdash s_2 : U_2$ and $U_2 <: T_2$, rule T-Sub gives

$$\Gamma, x{:}\, S_1 \vdash s_2 : T_2,$$

and we are done.

Progress

Theorem: *If* t *is a closed, well-typed term, then either* t *is a value or else there is some* t', *with and* t $\longrightarrow$ t'

Proof: By induction on *typing derivations*.

Which cases are likely to be hard?

case T-APP

case T-RCD

case T-PROJ

case T-SUB

Subtyping
with
Other Features

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T}}{\Gamma \vdash \mathtt{t_1} \text{ as } \mathtt{T} : \mathtt{T}} \qquad \text{(T-Ascribe)}$$

$$\mathtt{v_1} \text{ as } \mathtt{T} \longrightarrow \mathtt{v_1} \qquad \text{(E-Ascribe)}$$

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}}{\Gamma \vdash \mathtt{t}_1 \text{ as } \mathtt{T} : \mathtt{T}} \qquad \text{(T-ASCRIBE)}$$

$$\mathtt{v}_1 \text{ as } \mathtt{T} \longrightarrow \mathtt{v}_1 \qquad \text{(E-ASCRIBE)}$$

Casting (cf. Java):

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1 \text{ as } \mathtt{T} : \mathtt{T}} \qquad \text{(T-CAST)}$$

$$\frac{\vdash \mathtt{v}_1 : \mathtt{T}}{\mathtt{v}_1 \text{ as } \mathtt{T} \longrightarrow \mathtt{v}_1} \qquad \text{(E-CAST)}$$

Subtyping and Variants

$$\texttt{<l}_i\texttt{:T}_i\ ^{i\in \boxed{1..n}}\texttt{>} \quad <: \quad \texttt{<l}_i\texttt{:T}_i\ ^{i\in \boxed{1..n+k}}\texttt{>} \qquad \text{(S-VariantWidth)}$$

$$\frac{\text{for each } i \quad \texttt{S}_i <: \texttt{T}_i}{\texttt{<l}_i\texttt{:S}_i\ ^{i\in 1..n}\texttt{>} \quad <: \quad \texttt{<l}_i\texttt{:T}_i\ ^{i\in 1..n}\texttt{>}} \qquad \text{(S-VariantDepth)}$$

$$\frac{\texttt{<k}_j\texttt{:S}_j\ ^{j\in 1..n}\texttt{>} \text{ is a permutation of } \texttt{<l}_i\texttt{:T}_i\ ^{i\in 1..n}\texttt{>}}{\texttt{<k}_j\texttt{:S}_j\ ^{j\in 1..n}\texttt{>} \quad <: \quad \texttt{<l}_i\texttt{:T}_i\ ^{i\in 1..n}\texttt{>}} \qquad \text{(S-VariantPerm)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1}{\Gamma \vdash \texttt{<l}_1\texttt{=t}_1\texttt{>} : \texttt{<l}_1\texttt{:T}_1\texttt{>}} \qquad \text{(T-Variant)}$$

Subtyping and Lists

$$\frac{S_1 <: T_1}{\texttt{List}\ S_1 <: \texttt{List}\ T_1} \qquad \text{(S-List)}$$

i.e., List is a covariant type constructor.

Subtyping and References

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \qquad \text{(S-Ref)}$$

i.e., Ref is not a *covariant* (nor a *contravariant*) type constructor.

Subtyping and References

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \qquad \text{(S-REF)}$$

i.e., Ref is not a *covariant* (nor a *contravariant*) type constructor.

Why?

– When a reference is *read*, the context expects a T_1, so if $S_1 <: T_1$ then an S_1 is ok.

Subtyping and References

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\mathtt{Ref}\ S_1 <: \mathtt{Ref}\ T_1} \qquad \text{(S-REF)}$$

i.e., Ref is not a *covariant* (nor a *contravariant*) type constructor.

Why?

- When a reference is *read*, the context expects a T_1, so if $S_1 <: T_1$ then an S_1 is ok.
- When a reference is *written*, the context provides a T_1 and if the actual type of the reference is $\mathrm{Ref}\ S_1$, someone else may use the T_1 as an S_1. So we need $T_1 <: S_1$.

References again

Observation: a value of type Ref T can be used in two different ways: as a *source* for values of type T and as a *sink* for values of type T.

Idea： Split Ref T into three parts:

- Source T: reference cell with "read capability"
- Sink T: reference cell with "write capability"
- Ref T: cell with both capabilities

Subtyping and Arrays

Similarly...

$$\frac{\mathtt{S_1} <: \mathtt{T_1} \qquad \mathtt{T_1} <: \mathtt{S_1}}{\mathtt{Array}\ \mathtt{S_1} <: \mathtt{Array}\ \mathtt{T_1}} \qquad (\textsc{S-Array})$$

$$\frac{\mathtt{S_1} <: \mathtt{T_1}}{\mathtt{Array}\ \mathtt{S_1} <: \mathtt{Array}\ \mathtt{T_1}} \qquad (\textsc{S-ArrayJava})$$

This is regarded (even by the Java designers) as a mistake in the design.

References again

Observation: a value of type $Ref\ \ T$ can be used in two different ways:

- as a *source* for values of type T , and
- as a *sink* for values of type T.

References again

Observation: a value of type $Ref\ \ T$ can be used in two different ways:

- as a *source* for values of type T , and
- as a *sink* for values of type T.

Idea: Split Ref T into three parts:

- Source T: reference cell with "read capability"
- Sink T: reference cell with "write capability"
- Ref T: cell with both capabilities

Modified Typing Rules

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Source}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{!t}_1 : \mathtt{T}_{11}} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Sink}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1 \mathtt{:=} \mathtt{t}_2 : \mathtt{Unit}} \quad \text{(T-Assign)}$$

Subtyping rules

$$\frac{\mathtt{S_1} <: \mathtt{T_1}}{\mathtt{Source}\ \mathtt{S_1} <: \mathtt{Source}\ \mathtt{T_1}} \qquad \text{(S-Source)}$$

$$\frac{\mathtt{T_1} <: \mathtt{S_1}}{\mathtt{Sink}\ \mathtt{S_1} <: \mathtt{Sink}\ \mathtt{T_1}} \qquad \text{(S-Sink)}$$

$$\mathtt{Ref}\ \mathtt{T_1} <: \mathtt{Source}\ \mathtt{T_1} \qquad \text{(S-RefSource)}$$

$$\mathtt{Ref}\ \mathtt{T_1} <: \mathtt{Sink}\ \mathtt{T_1} \qquad \text{(S-RefSink)}$$

Capabilities

Other kinds of capabilities can be treated similarly, e.g.,

- send and receive capabilities on communication channels,
- encrypt/decrypt capabilities of cryptographic keys,
- ...

Intersection and Union Types

Intersection Types

The inhabitants of $T_1 \wedge T_2$ are terms belonging to *both* S and T —i.e., $T_1 \wedge T_2$ is an order-theoretic meet (greatest lower bound) of T_1 and T_2.

$$T_1 \wedge T_2 <: T_1 \qquad \text{(S-INTER1)}$$

$$T_1 \wedge T_2 <: T_2 \qquad \text{(S-INTER2)}$$

$$\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \wedge T_2} \qquad \text{(S-INTER3)}$$

$$\boxed{S \rightarrow T_1 \wedge S \rightarrow T_2 <: S \rightarrow (T_1 \wedge T_2) \qquad \text{(S-INTER4)}}$$

Intersection Types

Intersection types permit a very *flexible form* of *finitary overloading*.

$+ : (\mathtt{Nat}\to\mathtt{Nat}\to\mathtt{Nat}) \wedge (\mathtt{Float}\to\mathtt{Float}\to\mathtt{Float})$

This form of overloading is extremely powerful.

Every strongly normalizing untyped lambda-term can be typed in the simply typed lambda-calculus with intersection types.

type reconstruction problem is undecidable

Intersection types *have not been used much* in language designs (too powerful!), but are being *intensively investigated* as type systems *for intermediate languages* in highly optimizing compilers (cf. Church project).

Union types

Union types are also useful.

$T_1 \vee T_2$ is an untagged (non-disjoint) union of T_1 and T_2.

No tags : no *case* construct. The only operations we can safely perform on elements of $T_1 \vee T_2$ are ones *that make sense for both* T_1 and T_2.

N. B: untagged union types in C are a source of *type safety violations* precisely because they ignores this restriction, allowing any operation on an element of $T_1 \vee T_2$ that makes sense for *either* T_1 or T_2.

Union types are being used recently in type systems for XML processing languages (cf. Xduce, Xtatic).

Varieties of Polymorphism

- Parametric polymorphism (ML-style)
- Subtype polymorphism (OO-style)
- Ad-hoc polymorphism (overloading)

Chap 16

Metatheory of Subtyping

Algorithmic Subtyping

Algorithmic Typing

Joins and Meets

Developing an algorithmic subtyping relation

Subtype Relation

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

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

$$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n+k}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \qquad \text{(S-RcdWidth)}$$

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

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

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

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

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.

What to do?

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

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad (\text{T-App})$$

Syntax-directed rules

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

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

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

Technically, the reason this works is that we can *divide the "positions"* of the typing relation 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 *"subgoals"* from the subexpressions 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*, in the sense that, for every "*input*" Γ and t, *there is 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 we extend the system 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 plus 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 subgoal are exactly the same as the *inputs* to the main goal!

   (Hence, if we translated 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.
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?

1. *Observation*: We don't *need* lots of ways to prove a given typing or subtyping statement — *one is enough*.

   $\longrightarrow$ *Think more carefully about the typing and subtyping systems to see where we can get rid of excess flexibility.*

2. Use the resulting intuitions to formulate new "*algorithmic*" (i.e., syntax-directed) typing and subtyping relations.

3. Prove that the algorithmic relations are "*the same as*" the original ones in an appropriate sense.

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

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

Simpler subtype relation

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

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

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

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

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

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

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

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

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

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

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

$$\boxed{} \mapsto S <: Top \qquad (\boxed{S\text{A}}\text{-T}\textsc{op})$$

$$\frac{\mapsto T_1 <: S_1 \qquad \mapsto S_2 <: T_2}{\mapsto S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \qquad (\text{SA-A}\textsc{rrow})$$

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

Soundness and completeness

Theorem: $S <: T$ 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.)

Decision Procedures

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 $u \in R$.

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 $subtype(\mathrm{S}, \mathrm{T}) = true$, 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 $\mathrm{S} <: \mathrm{T}$)

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

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

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

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

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:

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

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

Note that, we are saying nothing about *computability.*

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

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

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*

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

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

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

subtype(S, T) =

if $T = Top$, then *true*

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

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

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

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*.

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 $S = S_1 \longrightarrow S_2$ and $T = T_1 \longrightarrow T_2$

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.

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?

Issue

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

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

Q: where is this rule really needed?

For applications, e.g., the term

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

is *not typable* without using subsumption.

Issue

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

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

Q: where is this rule really needed?

For applications, e.g., the term

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

is *not typable* without using subsumption.

Where else??

Issue

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

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

Q: where is this rule really needed?

For *applications*, e.g., the term

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

is *not typable* without using subsumption.

Where else??

Nowhere else!

Uses of subsumption to help typecheck *applications* are the only interesting ones.

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)

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

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*. So 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)

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

becomes

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

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

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)

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

Example (nested uses of T-Sub)

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

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 of T-App or
  2. the root of the derivation tree.
- In both cases, multiple uses of T-Sub can be coalesced 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 of 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 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.

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

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.

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

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

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

$$\frac{\Gamma \blacktriangleright \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 = \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \blacktriangleright \mathtt{t}_2 : \mathtt{T}_2 \qquad \blacktriangleright \mathtt{T}_2 <: \mathtt{T}_{11}}{\Gamma \blacktriangleright \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}}$$

(TA-APP)

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

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

Completeness of the algorithmic rules

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

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.

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

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

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

A Problem with Conditional Expressions

For the algorithmic presentation of the system, however, we encounter a little difficulty.

What is the minimal type of

$$if\ true\ then\ \{x = true, y = false\}\ else\ \{x = true, z = ture\}$$

?

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.

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

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.

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

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?

Examples

What are the joins of the following pairs of types?

1. $\{x: \text{Bool}, y: \text{Bool}\}$ and $\{y: \text{Bool}, z: \text{Bool}\}$?
2. $\{x: \text{Bool}\}$ and $\{y: \text{Bool}\}$?
3. $\{x: \{a: \text{Bool}, b: \text{Bool}\}\}$ and $\{x: \{b: \text{Bool}, c: \text{Bool}\}, y: \text{Bool}\}$?
4. $\{\}$ and Bool?
5. $\{x: \{\}\}$ and $\{x: \text{Bool}\}$?
6. $\text{Top} \longrightarrow \{x: \text{Bool}\}$ and $\text{Top} \longrightarrow \{y: \text{Bool}\}$?
7. $\{x: \text{Bool}\} \longrightarrow \text{Top}$ and $\{y: \text{Bool}\} \longrightarrow \text{Top}$?

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., $\mathrm{Bool} \longrightarrow \mathrm{Bool}$ 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.

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?

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 $\{y: Bool\} \longrightarrow Top$?

Calculating Joins

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

Calculating Meets

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

Homework☺

- Read and digest chapter 16 & 17

- HW#1: 16.2.5
- HW#2: 16.2.6