Recap on Exceptions

Recapitulation: Errors

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

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\mathtt{error}$	*run-time error*

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

$$\mathtt{error}\ \mathtt{t}_2 \longrightarrow \mathtt{error} \qquad \text{(E-AppErr1)}$$

$$\mathtt{v}_1\ \mathtt{error} \longrightarrow \mathtt{error} \qquad \text{(E-AppErr2)}$$

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

$$\Gamma \vdash \mathtt{error} : \mathtt{T} \qquad \text{(T-Error)}$$

Recapitulation: Error handling

→ *error* *try* Extends $\lambda_{\rightarrow}$ with errors (14-1)

New syntactic forms

$\texttt{t} ::= \ldots$ *terms:*

$\quad \texttt{try t with t}$ *trap errors*

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

$$\texttt{try } \texttt{v}_1 \texttt{ with } \texttt{t}_2 \longrightarrow \texttt{v}_1 \quad \text{(E-TryV)}$$

$$\texttt{try error with } \texttt{t}_2 \longrightarrow \texttt{t}_2 \quad \text{(E-TryError)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{try } \texttt{t}_1 \texttt{ with } \texttt{t}_2 \longrightarrow \texttt{try } \texttt{t}_1' \texttt{ with } \texttt{t}_2} \quad \text{(E-Try)}$$

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

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}}{\Gamma \vdash \texttt{try } \texttt{t}_1 \texttt{ with } \texttt{t}_2 : \texttt{T}} \quad \text{(T-Try)}$$

Recapitulation: Exceptions carrying values

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

New syntactic forms

t	::=	...	*terms:*
		raise t	*raise exception*
		try t with t	*handle exceptions*

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

$$(\text{raise } v_{11})\ t_2 \longrightarrow \text{raise } v_{11} \qquad \text{(E-AppRaise1)}$$

$$v_1\ (\text{raise } v_{21}) \longrightarrow \text{raise } v_{21} \qquad \text{(E-AppRaise2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{raise } t_1 \longrightarrow \text{raise } t_1'} \qquad \text{(E-Raise)}$$

$$\text{raise } (\text{raise } v_{11}) \longrightarrow \text{raise } v_{11} \qquad \text{(E-RaiseRaise)}$$

$$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1 \qquad \text{(E-TryV)}$$

$$\text{try raise } v_{11} \text{ with } t_2 \longrightarrow t_2\ v_{11} \qquad \text{(E-TryRaise)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t_1' \text{ with } t_2} \qquad \text{(E-Try)}$$

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

$$\frac{\Gamma \vdash t_1 : T_{exn}}{\Gamma \vdash \text{raise } t_1 : T} \qquad \text{(T-Exn)}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T_{exn} \rightarrow T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \qquad \text{(T-Try)}$$

Recapitulation: Type safety

The preservation theorem requires no changes when we add error: if a term of type T reduces to error, that's fine, since error has every type T.

Progress, though, requires a little more care.

Recapitulation: Progress

First, we do *not* plan to extend the set of values to include error, since this would make our new rule for propagating errors through applications.

$$\mathtt{v_1\ error} \longrightarrow \mathtt{error} \qquad \text{(E-AppErr2)}$$

overlap with our existing computation rule for applications:

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

e.g, the term

$$(\lambda\ x{:}\ Nat.\ 0)\ error$$

could evaluate to either 0 (which would be wrong) or error (which is what we intend).

Recapitulation: Progress

Instead, we keep error as a *non-value normal form*, and refine the statement of progress to explicitly mention the possibility that terms may evaluate to error instead of to a value.

Theorem [Progress]: *Suppose* t *is a closed, well-typed normal form. Then either* t *is a value or* $t = error$.

Recapitulation

- Raising exception is more than an error mechanism: it's a programmable control structure
  - Sometimes a way to quickly escape from the computation

- E.g., Exceptions are used in OCaml as a *control mechanism*, either to signal errors, or to control the flow of execution. When an exception is raised, the current execution is aborted, and control is thrown to the most recently entered active exception handler, which may choose to handle the exception, or pass it through to the next exception handler.

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

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

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

$$(\lambda\mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Subtyping $\boxed{\mathtt{S} <: \mathtt{T}}$

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

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

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

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

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

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

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\rightarrow\mathtt{T}_2} \qquad \text{(T-Abs)}$$

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

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

Safety

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

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. The rule T-SUB could appear anywhere.

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

Ascription and Casting

Ordinary ascription:

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

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

Ascription and Casting

Ordinary ascription:

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

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

Casting (cf. Java):

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

$$\frac{\vdash v_1 : T}{v_1 \text{ as } T \longrightarrow v_1} \qquad \text{(E-Cast)}$$

Subtyping and Variants

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

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

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

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

Subtyping and Lists

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

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

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

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 $\text{Ref } S_1$, 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...

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

$$\frac{S_1 <: T_1}{\mathtt{Array}\ S_1 <: \mathtt{Array}\ T_1} \quad \text{(S-ARRAYJAVA)}$$

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

Syntax

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

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-APP1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-APP2)}$$

$$(\lambda\mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-APPABS)}$$

Subtyping $\boxed{\mathtt{S} <: \mathtt{T}}$

$$\mathtt{S} <: \mathtt{S} \quad \text{(S-REFL)}$$

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

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

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

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

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

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\rightarrow\mathtt{T}_2} \quad \text{(T-ABS)}$$

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

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

Records

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

New syntactic forms

$$
\begin{array}{lll}
\mathtt{t} ::= & \ldots & \textit{terms:} \\
 & \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} & \textit{record} \\
 & \mathtt{t.l} & \textit{projection}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{v} ::= & \ldots & \textit{values:} \\
 & \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\} & \textit{record value}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{T} ::= & \ldots & \textit{types:} \\
 & \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} & \textit{type of records}
\end{array}
$$

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

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

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

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

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 $\boxed{S <: T}$

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

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

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

Chap 16

Metatheory of Subtyping

Algorithmic Subtyping

Algorithmic Typing

Joins and Meets

Algorithmic Typing and the Bottom Type

Developing an algorithmic subtyping relation

Subtype Relation

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

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

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

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

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

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

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

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.

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

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

Technically, the reason this works is that we can divide the "positions" of the typing relation into *input positions* (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 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)

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

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'd 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

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

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

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

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

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

Step 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

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

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

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

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

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

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

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

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

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

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 $u \in R$.

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

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:

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

Note that, for now, we are saying absolutely 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:

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

The function p' whose graph is

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

is *not* a decision function for R.

Decision Procedures

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:

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

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 $\mathrm{u} \in R$.

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

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

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 a decision procedure for R.

Subtyping Algorithm

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

else if $\mathrm{S} = \{\mathrm{k}_\mathrm{j}\colon\ \mathrm{S}_\mathrm{j}{}^{\mathrm{j}\in 1..\mathrm{m}}\}$ and $\mathrm{T} = \{\mathrm{l}_\mathrm{i}\colon\ \mathrm{T}_\mathrm{i}{}^{\mathrm{i}\in 1..n}\}$

then $\{\mathrm{l}_\mathrm{i}{}^{\mathrm{i}\in 1..\mathrm{n}}\} \subseteq \{\mathrm{k}_\mathrm{j}{}^{\mathrm{j}\in 1..\mathrm{m}}\}$

$\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $\mathrm{k}_\mathrm{j} = \mathrm{l}_\mathrm{i}$

and $\mathit{subtype}(\mathrm{S}_\mathrm{j}, \mathrm{T}_\mathrm{i})$

else *false*.

Subtyping Algorithm

This 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 $\mathrm{S} = \{\mathrm{k}_\mathrm{j}\colon\ \mathrm{S}_\mathrm{j}{}^{\mathrm{j}\in 1..\mathrm{m}}\}$ and $\mathrm{T} = \{\mathrm{l}_\mathrm{i}\colon\ \mathrm{T}_\mathrm{i}{}^{\mathrm{i}\in 1..n}\}$

then $\{\mathrm{l}_\mathrm{i}{}^{\mathrm{i}\in 1..\mathrm{n}}\} \subseteq \{\mathrm{k}_\mathrm{j}{}^{\mathrm{j}\in 1..\mathrm{m}}\}$

$\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $\mathrm{k}_\mathrm{j} = \mathrm{l}_\mathrm{i}$ and $subtype(\mathrm{S}_\mathrm{j}, \mathrm{T}_\mathrm{i})$

else *false*.

To show this, we need to prove:

1. that it returns *true* whenever $\mathrm{S} <: \mathrm{T}$, 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.

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

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

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

is not typable without using subsumption.

Issue

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

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

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.

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

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??

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

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

Example (T-Sub with T-Rcd)

$$
\dfrac{\text{for each } \mathtt{i} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t}_i : \mathtt{S}_i} \qquad \dfrac{\vdots}{\mathtt{S}_i <: \mathtt{T}_i}}{\Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}\ (\text{T-Sub})}{\Gamma \vdash \{\mathtt{l}_i = \mathtt{t}_i\ ^{i \in 1..n}\} : \{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\}}\ (\text{T-Rcd})
$$

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–Sub with T-APP on the left)

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

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

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

becomes

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

Example (T–Sub with T-APP on the right)

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

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 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 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.

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

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.

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

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

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

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

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

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

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.

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

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.

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

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

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

Calculating Joins

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

Calculating Meets

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

Homework☺

- Read chapter 16 & 17

- HW#1: 16.2.6, 16.3.2
- HW#2: Based on the codes of chap 17 and fulfill the exercise of 17.3.1