Programming Languages and Types

Klaus Ostermann

based on slides by Benjamin C. Pierce

Subtyping

Motivation

With our usual typing rule for applications

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

the term

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

is *not* well typed.

But this is silly: all we're doing is passing the function a *better* argument than it needs.

Polymorphism

A *polymorphic* function may be applied to many different types of data.

Varieties of polymorphism:

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

Our topic for today is *subtype* polymorphism, which is based on the idea of *subsumption*.

Subsumption

More generally: 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.

We can formalize this intuition 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`

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

Example

We will define subtyping between record types so that, for example,

$$\texttt{\{x:Nat, y:Nat\}} <: \texttt{\{x:Nat\}}$$

So, by subsumption,

$$\vdash \texttt{\{x=0,y=1\}} : \texttt{\{x:Nat\}}$$

and hence

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

is well typed.

The Subtype Relation: Records

"Width subtyping" (forgetting fields on the right):

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

Intuition: `{x:Nat}` is the type of all records with *at least* a numeric `x` field.

Note that the record type with *more* fields is a *sub*type of the record type with fewer fields.

Reason: the type with more fields places a *stronger constraint* on values, so it describes *fewer values*.

The Subtype Relation: Records

Permutation of fields:

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

By using S-RcdPerm together with S-RcdWidth and S-Trans allows us to drop arbitrary fields within records.

The Subtype Relation: Records

"Depth subtyping" within fields:

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

The types of individual fields may change.

Example

$$\dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\}} <: \texttt{\{a:Nat\}}}\ \textsc{S-RcdWidth} \qquad \dfrac{}{\texttt{\{m:Nat\}} <: \texttt{\{\}}}\ \textsc{S-RcdWidt}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat\},y:\{\}\}}}\ \textsc{S-RcdDept}$$

Variations

Real languages often choose not to adopt all of these record subtyping rules. For example, in Java,

- A subclass may not change the argument or result types of a method of its superclass (i.e., no depth subtyping)
  - Changed in Java 5, covariant return types now allowed.
- Each class has just one super*class* ("single inheritance" of classes)

  $\longrightarrow$ *each class member (field or method) can be assigned a single index, adding new indices "on the right" as more members are added in subclasses (i.e., no permutation for classes)*

- A class may implement multiple *interfaces* ("multiple inheritance" of interfaces)
  I.e., permutation is allowed for interfaces.

The Subtype Relation: Arrow types

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

Note the order of T_1 and S_1 in the first premise. The subtype relation is *contravariant* in the left-hand sides of arrows and *covariant* in the right-hand sides.

Intuition: if we have a function f of type $S_1 \to S_2$, then we know that f accepts elements of type S_1; clearly, f will also accept elements of any subtype T_1 of S_1. The type of f also tells us that it returns elements of type S_2; we can also view these results belonging to any supertype T_2 of S_2. That is, any function f of type $S_1 \to S_2$ can also be viewed as having type $T_1 \to T_2$.

The Subtype Relation: Top

It is convenient to have a type that is a supertype of every type. We introduce a new type constant `Top`, plus a rule that makes `Top` a maximum element of the subtype relation.

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

Cf. `Object` in Java.

The Subtype Relation: General rules

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

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

Subtype relation

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

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

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

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

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

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

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

Properties of Subtyping

Safety

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

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.

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

Preservation

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

Proof: By induction on typing derivations - see textbook.

Inversion Lemma for Typing

Lemma: If $\Gamma \vdash \lambda x{:}S_1.s_2 : T_1 \to T_2$, then $T_1 <: S_1$ and $\Gamma, x{:}S_1 \vdash s_2 : T_2$.

Proof: Induction on typing derivations - see textbook.

Subtyping with Other Features

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

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

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

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

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

Subtyping and Lists

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

I.e., `List` is a covariant type constructor.

Subtyping and References

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

We have not discussed typing of references in detail; informally think of a value of type `Ref T` as a box or variable of type `T`.

`Ref` is *not* a covariant (nor a contravariant) type constructor. Why?

- When a reference is *read*, the context expects a $\texttt{T}_1$, so if $\texttt{S}_1 \texttt{ <: } \texttt{T}_1$ then an $\texttt{S}_1$ is ok.
- When a reference is *written*, the context provides a $\texttt{T}_1$ and if the actual type of the reference is $\texttt{Ref S}_1$, someone else may use the $\texttt{T}_1$ as an $\texttt{S}_1$. So we need $\texttt{T}_1 \texttt{ <: } \texttt{S}_1$.

Subtyping and Arrays

Similarly...

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

$$\frac{\texttt{S}_1 \texttt{ <: } \texttt{T}_1}{\texttt{Array } \texttt{S}_1 \texttt{ <: Array } \texttt{T}_1} \qquad \text{(S-ArrayJava)}$$

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

Algorithmic Subtyping

Syntax-directed rules

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

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

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 $T_{11} \to T_{12}$
3. finding (recursively) a type for t_2
4. checking that it is the same as T_{11}

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)

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

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 $\mathtt{t}$, there is only one rule that can be used to derive typing statements involving $\mathtt{t}$.

E.g., if $\mathtt{t}$ 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 $\mathtt{t}$. If it fails, then we know that $\mathtt{t}$ 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!
   (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

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

   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* 1000 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.

Developing an algorithmic subtyping relation

Subtype relation

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

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

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

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

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

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

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

Issues

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

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

Step 1: simplify record subtyping

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

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

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

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

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

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

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

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

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

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

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

Soundness and completeness

Theorem: $S <: T$ iff $\vdash S <: T$.

Terminology:

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

Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

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

if $\mathtt{T} = \mathtt{Top}$, then *true*
else if $\mathtt{S} = \mathtt{S}_1 \to \mathtt{S}_2$ and $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$
then $\mathit{subtype}(\mathtt{T}_1, \mathtt{S}_1) \;\wedge\; \mathit{subtype}(\mathtt{S}_2, \mathtt{T}_2)$
else if $\mathtt{S} = \{\mathtt{k}_j : \mathtt{S}_j \;^{j \in 1..m}\}$ and $\mathtt{T} = \{\mathtt{l}_i : \mathtt{T}_i \;^{i \in 1..n}\}$
then $\{\mathtt{l}_i \;^{i \in 1..n}\} \subseteq \{\mathtt{k}_j \;^{j \in 1..m}\}$
$\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $\mathtt{k}_j = \mathtt{l}_i$
and $\mathit{subtype}(\mathtt{S}_j, \mathtt{T}_i)$
else *false*.

Algorithmic Typing

Algorithmic typing

- How do we implement a type checker for the lambda-calculus with subtyping?
- Given a context Γ and a term $\mathtt{t}$, how do we determine its type $\mathtt{T}$, such that $\Gamma \vdash \mathtt{t} : \mathtt{T}$?

Issue

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

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

We observed above that this rule is sometimes *required* when typechecking applications:

E.g., the term

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

is not typable without using subsumption.

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

Plan

1. Investigate how subsumption is used in typing derivations by looking at examples of how it can be "pushed through" other rules
2. Use the intuitions gained from this exercise to design a new, algorithmic typing relation that
   - omits subsumption
   - compensates for its absence by enriching the application rule
3. Show that the algorithmic typing relation is essentially equivalent to the original, declarative one

Example (T-Sub with 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

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

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-App with (T-Sub on the left)

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

becomes

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

Example: T-App with (T-Sub 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

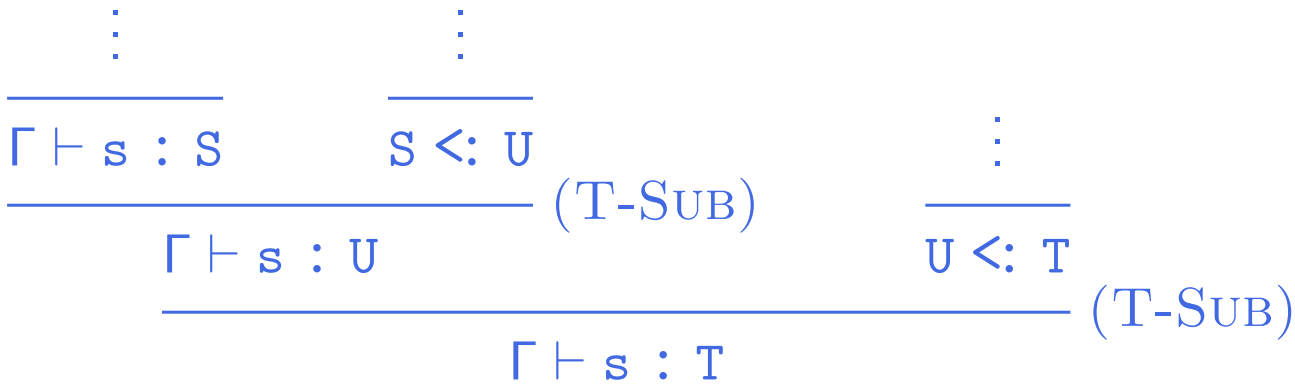
$$\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11}\to T_{12}} \qquad \dfrac{\dfrac{\vdots}{T_2 <: T_{11}} \qquad \dfrac{}{T_{12} <: T_{12}}\,(\text{S-Refl})}{T_{11}\to T_{12} <: T_2\to T_{12}}\,(\text{S-Arrow})}{\Gamma \vdash s_1 : T_2\to T_{12}}\,(\text{T-Sub}) \qquad \dfrac{\vdots}{\Gamma \vdash s_2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

$$\Gamma \vdash s_1\ s_2 : T_{12}\,(\text{T-A}$$

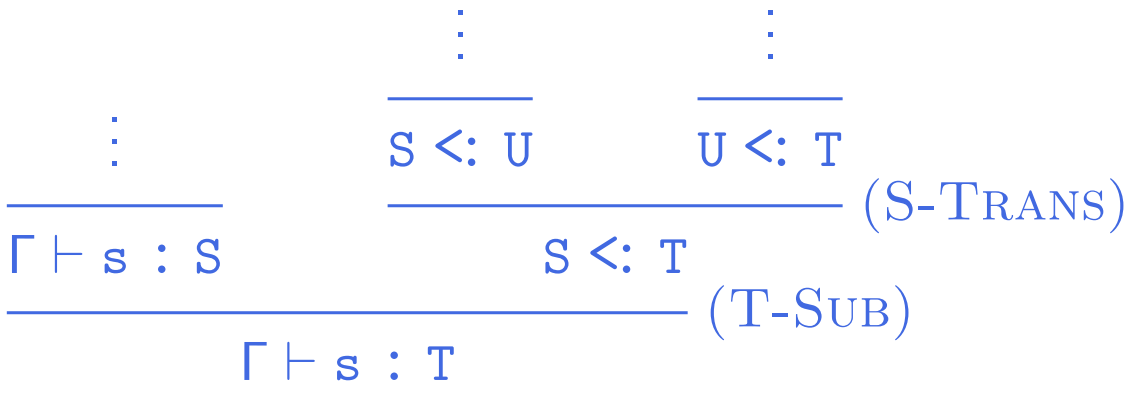
Intuitions

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

Example (nested uses of T-Sub)



becomes



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 fo the derivation tree.
- In both cases, multiple uses of T-Sub can be collapsed into a single one.

This suggests a notion of "normal form" for typing derivations, in which there is

- exactly one use of T-Sub before each use of T-App
- one use of T-Sub at the very end of the derivation
- no uses of T-Sub anywhere else.

Algorithmic Typing

The next step is to "build in" the use of subsumption in application rules, by changing the T-APP rule to incorporate a subtyping premise.

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

Given any typing derivation, we can now

1. normalize it, to move all uses of subsumption to either just before applications (in the right-hand premise) or at the very end
2. replace uses of T-APP with T-SUB in the right-hand premise by uses of the extended rule above

This yields a derivation in which there is just *one* use of subsumption, at the very end!

Minimal Types

But... if subsumption is only used at the very end of derivations, then it is actually *not needed* in order to show that any term is typable!
It is just used to give *more* types to terms that have already been shown to have a type.

In other words, if we dropped subsumption completely (after refining the application rule), we would still be able to give types to exactly the same set of terms — we just would not be able to give as many types to some of them.

If we drop subsumption, then the remaining rules will assign a *unique, minimal* type to each typable term.

For purposes of building a typechecking algorithm, this is enough.

Final Algorithmic Typing Rules

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

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

$$\frac{\Gamma \vdash\!\blacktriangleright t_1 : T_1 \qquad T_1 = T_{11} \to T_{12} \qquad \Gamma \vdash\!\blacktriangleright t_2 : T_2 \qquad \vdash\!\blacktriangleright T_2 <: T_{11}}{\Gamma \vdash\!\blacktriangleright t_1\ t_2 : T_{12}} \qquad \text{(TA-App)}$$

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

$$\frac{\Gamma \vdash\!\blacktriangleright t_1 : R_1 \qquad R_1 = \{l_1{:}T_1 \ldots l_n{:}T_n\}}{\Gamma \vdash\!\blacktriangleright t_1.l_i : T_i} \qquad \text{(TA-Proj)}$$

Soundness and Completeness of the algorithmic rules

Theorem: If $\Gamma \vdash\!\blacktriangleright t : T$, then $\Gamma \vdash t : T$.

Theorem : If $\Gamma \vdash t : T$, then $\Gamma \vdash\!\blacktriangleright t : S$ for some $S <: T$.

Meets and Joins

Adding Booleans

Suppose we want to add booleans and conditionals to the language we have been discussing.
For the *declarative* presentation of the system, we just add in the appropriate syntactic forms, evaluation rules, and typing rules.

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

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

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

A Problem with Conditional Expressions

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

What is the minimal type of

```
if true then {x=true,y=false} else {x=true,z=true}
```

?

The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

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

any type that is a possible type of both t_2 and t_3.

So the *minimal* type of the conditional is the *least common supertype* (or *join*) of the minimal type of t_2 and the minimal type of t_3.

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

Does such a type exist for every T_2 and T_3??

Existence of Joins

Theorem: For every pair of types `S` and `T`, there is a type `J` such that

1. `S <: J`
2. `T <: J`
3. If `K` is a type such that `S <: K` and `T <: K`, then `J <: K`.

I.e., `J` is the smallest type that is a supertype of both `S` and `T`.

Examples

What are the joins of the following pairs of types?

1. `{x:Bool,y:Bool}` and `{y:Bool,z:Bool}`?
2. `{x:Bool}` and `{y:Bool}`?
3. `{x:{a:Bool,b:Bool}}` and
   `{x:{b:Bool,c:Bool}, y:Bool}`?
4. `{}` and `Bool`?
5. `{x:{}}` and `{x:Bool}`?
6. `Top→{x:Bool}` and `Top→{y:Bool}`?
7. `{x:Bool}→Top` and `{y:Bool}→Top`?

Meets

To calculate joins of arrow types, we also need to be able to calculate *meets* (greatest lower bounds)!

Unlike joins, meets do not necessarily exist.
E.g., `Bool→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`.

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→{x:Bool}` and `Top→{y:Bool}`?
7. `{x:Bool}→Top` and `{y:Bool}→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

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

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

Universal Types

Motivation

In the simply typed lambda-calculus, we often have to write several versions of the same code, differing only in type annotations.

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x)
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x)
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x)
```

Bad! Violates a basic principle of software engineering:

> Write each piece of functionality once and parameterize it on the details that vary from one instance to another.

Here, the details that vary are the types!

Idea

We'd like to be able to take a piece of code and "abstract out" some type annotations.

We've already got a mechanism for doing this with terms: λ-abstraction. So let's just re-use the notation.

Abstraction:

```
double = λX. λf:X→X. λx:X. f (f x)
```

Application:

```
double [Nat]
double [Bool]
```

Computation:

```
double [Nat] ⟶ λf:Nat→Nat. λx:Nat. f (f x)
```

(N.b.: Type application is commonly written `t [T]`, though `t T` would be more consistent.)

Idea

What is the *type* of a term like

$$\lambda X.\ \lambda f{:}X{\to}X.\ \lambda x{:}X.\ f\ (f\ x)\ ?$$

This term is a function that, when applied to a type X, yields a term of type $(X{\to}X){\to}X{\to}X$.

I.e., for all types X, it yields a result of type $(X{\to}X){\to}X{\to}X$.

We'll write it like this: $\forall X.\ (X{\to}X){\to}X{\to}X$

System F

System F (aka "the polymorphic lambda-calculus") formalizes this idea by extending the simply typed lambda-calculus with type abstraction and type application.

`t` ::=		*terms*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`λX.t`	*type abstraction*
	`t [T]`	*type application*
`v` ::=		*values*
	`λx:T.t`	*abstraction value*
	`λX.t`	*type abstraction value*

System F: new evaluation rules

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \qquad \text{(E-TApp)}$$

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \qquad \text{(E-TappTabs)}$$

System F: Types

To talk about the types of "terms abstracted on types," we need to introduce a new form of types:

$\mathtt{T}$	::=	*types*
	$\mathtt{X}$	*type variable*
	$\mathtt{T} \to \mathtt{T}$	*type of functions*
	$\forall \mathtt{X}.\mathtt{T}$	*universal type*

System F: Typing Rules

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

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

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

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X}.\mathtt{t}_2 : \forall \mathtt{X}.\mathtt{T}_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \qquad \text{(T-TApp)}$$

History

Interestingly, System F was invented independently and almost simultaneously by a computer scientist (John Reynolds) and a logician (Jean-Yves Girard).

Their results look very different at first sight — one is presented as a tiny programming language, the other as a variety of second-order logic.

The similarity (indeed, isomorphism!) between them is an example of the *Curry-Howard Correspondence*.

Examples

Lists

```
cons : ∀X. X → List X → List X
head : ∀X. List X → X
tail : ∀X. List X → List X
nil : ∀X. List X
isnil : ∀X. List X → Bool

map =
  λX. λY.
    λf: X→Y.
      (fix (λm: (List X) → (List Y).
              λl: List X.
                if isnil [X] l
                  then nil [Y]
                  else cons [Y] (f (head [X] l))
                                (m (tail [X] l))));

l = cons [Nat] 4 (cons [Nat] 3 (cons [Nat] 2 (nil [Nat])));

head [Nat] (map [Nat] [Nat] (λx:Nat. succ x) l);
```

Church Booleans

```
CBool = ∀X.X→X→X;

tru = λX. λt:X. λf:X. t;
fls = λX. λt:X. λf:X. f;

not = λb:CBool. λX. λt:X. λf:X. b [X] f t;
```

Church Numerals

```
CNat = ∀X. (X→X) → X → X;

c0  =  λX. λs:X→X. λz:X. z;
c1   =  λX. λs:X→X. λz:X. s z;
c2   =  λX. λs:X→X. λz:X. s (s z);

csucc = λn:CNat. λX. λs:X→X. λz:X. s (n [X] s z);

cplus = λm:CNat. λn:CNat. m [CNat] csucc n;
```

Properties of System F

Preservation and Progress: unchanged.

(Proofs similar to what we've seen.)

Strong normalization: every well-typed program halts. (Proof is challenging!)

Type reconstruction: undecidable (major open problem from 1972 until 1994, when Joe Wells solved it).

Parametricity

Observation: Polymorphic functions cannot do very much with their arguments.

- The type $\forall \mathtt{X}.\ \mathtt{X}{\to}\mathtt{X}{\to}\mathtt{X}$ has exactly two members (up to observational equivalence).
- $\forall \mathtt{X}.\ \mathtt{X}{\to}\mathtt{X}$ has one.
- etc.

The concept of parametricity gives rise to some useful "free theorems..."