Type Reconstruction and Polymorphism
We now come to the question of type checking and type reconstruction.

**Type checking:** Given $\Gamma$, $t$, and $T$, check whether $\Gamma \vdash t : T$

**Type reconstruction:** Given $\Gamma$ and $t$, find a type $T$ such that $\Gamma \vdash t : T$

Type checking and reconstruction seem difficult since parameters in lambda calculus do not carry their types with them.

Type reconstruction also suffers from the problem that a term can have many types.

**Idea:** We construct all type derivations in parallel, reducing type reconstruction to a unification problem.
From Judgements to Equations

\[ TP : \text{Judgement} \rightarrow \text{Equations} \]

\[ TP(\Gamma \vdash t : T) = \]

\[ \text{case } t \text{ of} \]

\[ x : \{ \Gamma(x) \doteq T \} \]

\[ \lambda x.t' : \text{let } a, b \text{ fresh in} \]

\[ \{(a \rightarrow b) \doteq T\} \cup TP(\Gamma, x : a \vdash t' : b) \]

\[ tt' : \text{let } a \text{ fresh in} \]

\[ TP(\Gamma \vdash t : a \rightarrow T) \cup TP(\Gamma \vdash t' : a) \]
**Soundness and Completeness I**

**Definition:** In general, a type reconstruction algorithm $\mathcal{A}$ assigns to an environment $\Gamma$ and a term $t$ a set of types $\mathcal{A}(\Gamma, t)$.

The algorithm is **sound** if for every type $T \in \mathcal{A}(\Gamma, t)$ we can prove the judgement $\Gamma \vdash t : T$.

The algorithm is **complete** if for every provable judgement $\Gamma \vdash t : T$ we have that $T \in \mathcal{A}(\Gamma, t)$. 
Theorem: $TP$ is sound and complete. Specifically:

$$\Gamma \vdash t : T \iff \exists \bar{b}.[T/a]EQNS$$

where

$a$ is a new type variable

$EQNS = TP(\Gamma \vdash t : a)$

$$\bar{b} = tv(EQNS) \setminus tv(\Gamma)$$

Here, $tv$ denotes the set of free type variables (of a term, and environment, an equation set).
Type Reconstruction and Unification

**Problem:** Transform set of equations

\[ \{T_i \equiv U_i\}_{i=1, \ldots, m} \]

into equivalent substitution

\[ \{a_j \equiv T'_j\}_{j=1, \ldots, n} \]

where type variables do not appear recursively on their right hand sides (directly or indirectly). That is:

\[ a_j \not\in tv(T'_k) \quad \text{for } j = 1, \ldots, n, k = j, \ldots, n \]
Substitutions

A substitution \( s \) is an idempotent mapping from type variables to types which maps all but a finite number of type variables to themselves.

We often represent a substitution as a set of equations \( a \doteq T \) with \( a \) not in \( tv(T) \).

Substitutions can be generalized to mappings from types to types by defining

\[
\begin{align*}
\text{s}(T \rightarrow U) &= \text{sT} \rightarrow \text{sU} \\
\text{s}(K[T_1, \ldots, T_n]) &= K[sT_1, \ldots, sT_n]
\end{align*}
\]

Substitutions are idempotent mappings from types to types, i.e. \( s(s(T)) = s(T) \). (why?)

The \( \circ \) operator denotes composition of substitutions (or other functions): \((f \circ g) x = f(g(x))\).
A Unification Algorithm

We present an incremental version of Robinson’s algorithm (1965).

\[
mgu : \text{(Type} \hat{=} \text{Type)} \rightarrow \text{Subst} \rightarrow \text{Subst}
\]

\[
mgu(T \hat{=} U) s = mgu'(sT \hat{=} sU) s
\]

\[
mgu'(a \hat{=} a) s = s
\]

\[
mgu'(a \hat{=} T) s = s \cup \{a \hat{=} T\} \quad \text{if } a \not\in \text{tv}(T)
\]

\[
mgu'(T \hat{=} a) s = s \cup \{a \hat{=} T\} \quad \text{if } a \not\in \text{tv}(T)
\]

\[
mgu'(T \rightarrow T' \hat{=} U \rightarrow U') s = (mgu(T' \hat{=} U') \circ mgu(T \hat{=} U)) s
\]

\[
mgu'(K[T_1, \ldots, T_n] \hat{=} K[U_1, \ldots, U_n]) s
\]

\[
= (mgu(T_n \hat{=} U_n) \circ \ldots \circ mgu(T_1 \hat{=} U_1)) s
\]

\[
mgu'(T \hat{=} U) s = \text{error} \quad \text{in all other cases}
\]
Soundness and Completeness of Unification

**Definition:** A substitution $u$ is a unifier of a set of equations \{${T_i \equiv U_i}\}_{i=1,...,m}$ if $uT_i = uU_i$, for all $i$. It is a most general unifier if for every other unifier $u'$ of the same equations there exists a substitution $s$ such that $u' = s \circ u$.

**Theorem:** Given a set of equations $EQNS$. If $EQNS$ has a unifier then $mgu EQNS {}$ computes the most general unifier of $EQNS$. If $EQNS$ has no unifier then $mgu EQNS {}$ fails.
From Judgements to Substitutions

\[ TP : \text{Judgement} \to \text{Subst} \to \text{Subst} \]

\[ TP(\Gamma \vdash t : T) = \]

\textit{case} \( t \) \textit{of}

\[ \begin{array}{ll}
  x & : \text{mgu}(\Gamma(x) \vdash T') \\
  \lambda x.t' & : \text{let } a, b \text{ fresh in}
  \begin{array}{l}
    \text{mgu}((a \to b) \vdash T')
  \end{array} \circ
  TP(\Gamma, x : a \vdash t' : b) \\
  t \ t' & : \text{let } a \text{ fresh in}
  \begin{array}{l}
    TP(\Gamma \vdash t : a \to T')
  \end{array} \circ
  TP(\Gamma \vdash t' : a)
\end{array} \]
Soundness and Completeness II

One can show by comparison with the previous algorithm:

**Theorem:** \( TP \) is sound and complete. Specifically:

\[
\Gamma \vdash t : T \quad \text{iff} \quad T = r(s(a))
\]

where

- \( a \) is a new type variable
- \( s = TP (\Gamma \vdash t : a) \{\} \)
- \( r \) is a substitution on \( tv(s \ a) \setminus tv(s \ \Gamma) \)
Strong Normalization

Question: Can $\Omega$ be given a type?

$$\Omega = (\lambda x.xx)(\lambda x.xx) : ?$$

What about $Y$?

Self-application is not typable!

In fact, we have more:

Theorem: (Strong Normalization) If $\vdash t : T$, then there is a value $V$ such that $t \rightarrow^* V$.

Corollary: Simply typed lambda calculus is not Turing complete.
Polymorphism

In the simply typed lambda calculus, a term can have many types. But a variable or parameter has only one type.

Example:

\[(\lambda x. xx)(\lambda y. y)\]

is untypable. But if we substitute actual parameter for formal, we obtain

\[(\lambda y. y)(\lambda y. y) : a \rightarrow a\]

Functions which can be applied to arguments of many types are called polymorphic.
Polymorphism in Programming

Polymorphism is essential for many program patterns.

Example: map

def map f xs =
    if (isEmpty (xs)) nil
    else cons (f (head xs)) (map (f, tail xs))
...

names: List[String]
nums : List[Int]
...

map toUpperCase names
map increment nums

Without a polymorphic type for map one of the last two lines is always illegal!
Forms of Polymorphism

Polymorphism means “having many forms”.

Polymorphism also comes in several forms.

- **Universal polymorphism**, sometimes also called **generic types**: The ability to instantiate type variables.

- **Inclusion polymorphism**, sometimes also called **subtyping**: The ability to treat a value of a subtype as a value of one of its supertypes.

- **Ad-hoc polymorphism**, sometimes also called **overloading**: The ability to define several versions of the same function name, with different types.

We first concentrate on universal polymorphism.

Two basic approaches: **explicit** or **implicit**.
Explicit Polymorphism

We introduce a polymorphic type $\forall a. T$, which can be used just as any other type.

We then need to make introduction and elimination of $\forall$'s explicit.

Typing rules:

\[
\begin{align*}
(\forall E) & \quad \frac{}{\Gamma \vdash t : \forall a. T} \\
(\forall I) & \quad \frac{}{\Gamma \vdash \Lambda a. t : \forall a. T}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : \forall a. T & \quad \frac{}{\Gamma \vdash t[U] : [U/a]T} \\
\Gamma \vdash t : T & \quad \frac{}{\Gamma \vdash \Lambda a. t : \forall a. T}
\end{align*}
\]
We also need to give all parameter types, so programs become verbose.

**Example:**

def map [a][b] (f: a -> b) (xs: List[a]) =
    if (isEmpty [a] (xs)) nil [a]
    else cons [b] (f (head [a] xs)) (map [a][b] (f, tail [a] xs))

names: List[String]
nums : List[Int]

... 
map [String] [String] toUpperCase names
map [Int] [Int] increment nums
Implicit Polymorphism

Implicit polymorphism does not require annotations for parameter types or type instantiations.

**Idea:** In addition to types (as in simply typed lambda calculus), we have a new syntactic category of type schemes. Syntax:

\[
\text{Type Scheme} \quad S \ ::= \ T \mid \forall a. S
\]

Type schemes are not fully general types; they are used only to type named values, introduced by a `let` construct.

The resulting type system is called the Hindley/Milner system, after its inventors.
Hindley/Milner Typing rules

(VAR) \( \Gamma, x : S, \Gamma' \vdash x : S \quad (x \not\in \text{dom}(\Gamma')) \)

(\forall E) \quad \frac{\Gamma \vdash t : \forall a.T}{\Gamma \vdash t : [U/a]T} \quad (\forall I) \quad \frac{\Gamma \vdash t : T \quad a \not\in \text{tv}(\Gamma)}{\Gamma \vdash t : \forall a.T}

(LET) \quad \frac{\Gamma \vdash t : S \quad \Gamma, x : S \vdash t' : T}{\Gamma \vdash \text{let } x = t \text{ in } t' : T}

The other two rules are as in simply typed lambda calculus:

(\to I) \quad \frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda x.t : T \to U} \quad (\to E) \quad \frac{\Gamma \vdash M : T \to U \quad \Gamma \vdash N : T}{\Gamma \vdash M N : U}
Here is a formulation of the map example in the Hindley/Milner system.

```
let map = \f. \xs in
  if (isEmpty (xs)) nil
  else cons (f (head xs)) (map (f, tail xs))

...// names: List[String]
// nums : List[Int]
// map : \forall a. \forall b. (a -> b) -> List[a] -> List[b]
...
map toUpperCase names
map increment nums
```
Limitations of Hindley/Milner

Hindley/Milner still does not allow parameter types to be polymorphic. I.e.

\[(\lambda x.x x)(\lambda y.y)\]

is still ill-typed, even though the following is well-typed:

\[
\text{let } id = \lambda y.y \text{ in } id\ id
\]

With explicit polymorphism the expression could be completed to a well-typed term:

\[(\Lambda a.\lambda x : (\forall a : a \rightarrow a).x[a \rightarrow a](x[a]))(\Lambda b.\lambda y : b.y)\]
The Essence of let

We regard

\[
\text{let } x = t \text{ in } t'
\]

as a shorthand for

\[
[t/x]t'
\]

We use this equivalence to get a revised Hindley/Milner system.

**Definition:** Let \( HM' \) be the type system that results if we replace rule \((\text{LET})\) from the Hindley/Milner system \( HM \) by:

\[
\frac{\Gamma \vdash t : T \quad \Gamma \vdash [t/x]t' : U}{\Gamma \vdash \text{let } x = t \text{ in } t' : U}
\]
**Theorem:** \( \Gamma \vdash_{HM} t : S \iff \Gamma \vdash_{HM'} t : S \)

The theorem establishes the following connection between the Hindley/Milner system and the simply typed lambda calculus \( F_1 \):

**Corollary:** Let \( t^* \) be the result of expanding all *let*’s in \( t \) according to the rule

\[
\text{let } x = t \text{ in } t' \rightarrow [t/x]t'
\]

Then

\[
\Gamma \vdash_{HM} t : T \Rightarrow \Gamma \vdash_{F_1} t^* : T
\]

Furthermore, if every *let*-bound name is used at least once, we also have the reverse:

\[
\Gamma \vdash_{F_1} t^* : T \Rightarrow \Gamma \vdash_{HM} t : T
\]
Type Reconstruction for Hindley/Milner

Type reconstruction for the Hindley/Milner system works as for simply typed lambda calculus, but with a clause for \textit{let} expressions and instantiation of type schemes:

\[
\text{newInstance}(\forall a_1, \ldots, a_n. S) = \\
\text{let } b_1, \ldots, b_n \text{ fresh in } \\
[b_1/a_1, \ldots, b_n/a_n] S
\]
\[ TP : \text{Judgement} \rightarrow \text{Subst} \rightarrow \text{Subst} \]

\[ TP(\Gamma \vdash t : T) \, s = \]

\[ \text{case } t \text{ of} \]

\[ \ldots \]

\[ x \quad : \quad \text{mgu}(\text{newInstance}(\Gamma(x)) \triangleq T) \, s \]

\[ \text{let } x = t_1 \text{ in } t_2 : \quad \text{let } a, b \text{ fresh } \text{in} \]

\[ \text{let } s_1 = TP(\Gamma \vdash t_1 : a) \, s \text{ in} \]

\[ TP(\Gamma, x : \text{gen}(s_1 \, \Gamma, s_1 \, a) \vdash t_2 : b) \, s_1 \]

where \[ \text{gen}(\Gamma, T) = \forall tv(T) \setminus tv(\Gamma).T \]
Principal Types

**Definition:** A type $T$ is a *generic instance* of a type scheme $S = \forall \alpha_1 \ldots \forall \alpha_n. T'$ if there is a substitution $s$ on $\alpha_1, \ldots, \alpha_n$ such that $T = sT'$. We write in this case $S \leq T$.

**Definition:** A type scheme $S'$ is a generic instance of a type scheme $S$ iff for all types $T$

$$S' \leq T \implies S \leq T$$

We write in this case $S \leq S'$. 
**Definition:** A type scheme $S$ is **principal** (or: **most general**) for $\Gamma$ and $t$ iff

- $\Gamma \vdash t : S$
- $\Gamma \vdash t : S'$ implies $S \leq S'$
**Definition:** A type system $TS$ has the principal typing property iff, whenever $\Gamma |-_{TS} t : S$ then there exists a principal type scheme for $\Gamma$ and $t$.

**Theorem:**

1. $HM'$ without $let$ has the p.t.p.
2. $HM'$ with $let$ has the p.t.p.
3. $HM$ has the p.t.p.

Proof sketch: 
1. Use type reconstruction result for the simply typed lambda calculus.
2. Expand all $let$'s and apply (1.).
3. Use equivalence between $HM$ and $HM'$. 