Simple Type Inference
Announcements

- Quiz 5, pushed back to Thursday
An aside… Monads

- The Monad quote:
  - “A monad is just a monoid in the category of endofunctors, what's the problem?”

- Monad type class and the monad laws
- Maybe monad
- List monad
- IO monad
- State monad
Monads

- A way to cleanly compose computations
  - E.g., \( f \) may return a value of type \( a \) or Nothing

Composing computations becomes tedious:
```haskell
case (f s) of
    Nothing \rightarrow Nothing
    Just m \rightarrow case (f m) ...
```

- In Haskell, monads model IO and other imperative features
An Example: Cloned Sheep

type Sheep = ...
father :: Sheep → Maybe Sheep
father = ...
mother :: Sheep → Maybe Sheep
mother = ...
(Note: a sheep has both parents; a cloned sheep has one)
maternalGrandfather :: Sheep → Maybe Sheep
maternalGrandfather s = case (mother s) of
  Nothing → Nothing
  Just m → father m
An Example

mothersPaternalGrandfather :: Sheep → Maybe Sheep

mothersPaternalGrandfather s = case (mother s) of
    Nothing → Nothing
    Just m → case (father m) of
        Nothing → Nothing
        Just gf → father gf

- Tedious, unreadable, difficult to maintain
- Monads help!
The Monad Class

- Haskell’s Monad type class requires 2 operations, >>= (bind) and return

class Monad m where

  // >>= (the bind operation) takes a monad // m a, and a function that takes a and turns // it into a monad m b, and returns m b
  (>>=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b

  // return encapsulates a value into the monad
  return :: a \rightarrow m a
The **Maybe Monad**

```haskell
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return          = Just
```

- Back to our example:

```haskell
mothersPaternalGrandfather s =
    (return s) >>= mother >>= father >>= father
```

(Note: if at any point, some function returns Nothing, it gets cleanly propagated.)
The List Monad

- The List type constructor is a monad
  \[ li >>= f = \text{concat} \ (\text{map} \ f \ li) \]
  \[ \text{return} \ x = [x] \]

Note: \text{concat}::[[a]] \to [a]

e.g., \text{concat} [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

- Use \textbf{any} \ f \ s.t. \ f::a\to[b]. \ f \ may \ return \ a \ list \ of
  0,1,2,... \ elements \ of \ type \ b, \ e.g.,
  
  > f \ x = [x+1]
  > [1,2,3] >>= f \ // \ returns \ [2,3,4]
The List Monad

parents :: Sheep → [Sheep]
parents s = MaybeToList (mother s) ++
            MaybeToList (father s)

grandParents :: Sheep → [Sheep]
grandParents s = (parents s) >>= parents
Last Week

- Introduction to types and type systems
- Simply typed lambda calculus (System $F_1$)
  - Language syntax, type expression syntax
  - Static semantics
  - Dynamic semantics
  - Type soundness: Safety = Progress + Preservation
    - Proved for the simply typed lambda calculus
- Intro to simple type inference
Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost

- Parametric polymorphism (next time…)

- Hindley Milner type inference. Algorithm W
Reading

- “Types and Programming Languages”, by Benjamin Pierce, Chapter 22, 23

- Lecture notes based partially on MIT 2015 Program Analysis OCW
Core Typing Rules

Γ |- c : int

x:τ ∈ Γ

Γ |- x : τ

Γ,x:σ |- E₁ : τ

Γ |- (λx:σ. E₁) : σ → τ

Γ |- E₁ : σ→τ ∆ Γ |- E₂ : σ

Γ |- (E₁ E₂) : τ
Extensions to Core Typing Rules

\[ \Gamma |- c : \text{int} \]

\[ \Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int} \]

\[ \Gamma |- E_1 + E_2 : \text{int} \]

\[ \Gamma |- E_1 = E_2 : \text{bool} \]

\[ \Gamma |- E_1 : \tau \quad \Gamma |- E_2 : \tau \]

\[ \Gamma |- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau \]
Type Inference, Strategy 1

- We can figure out all types even without explicit types for variables
  - \((\lambda f. f\ 5)\ (\lambda x. x+1) : ?\)
  - Type inference

Type inference, Strategy 1

- Use typing rules to define type constraints
- Solve type constraints
- Aka constraint-based typing (e.g., Pierce)
We Can Infer All Types!

\[(\lambda f. f \ 5) \ (\lambda x. x+1) : ?\]

1. App
   \[\Gamma = []\]
   \[t_2 = t_4 \rightarrow t_1\]

2. Abs
   \[\Gamma = []\]
   \[t_2 = t_f \rightarrow t_3\]

3. App
   \[\Gamma = [f:t_f]\]
   \[t_f = \text{int} \rightarrow t_3\]
   \[\lambda f: t_f\]
   \[\text{Var } f\]
   \[\text{Const } 5\]

4. Abs
   \[\Gamma = [x:t_x]\]
   \[\lambda x: t_x\]

5. +
   \[\Gamma = [x:t_x]\]
   \[t_5 = \text{int}\]
   \[t_x = \text{int}\]
   \[\text{Var } x\]
   \[\text{Const } 1\]
Type Constraints

- We constructed a system of type constraints
- Let’s solve the system of constraints

\[
\begin{align*}
t_2 &= t_4 \rightarrow t_1 \\
t_2 &= t_f \rightarrow t_3 \\
t_4 &= t_x \rightarrow t_5 \\
t_f &= \text{int} \rightarrow t_3 \\
t_5 &= \text{int}, \ t_x = \text{int} \\
\end{align*}
\]

We inferred all \( t \)'s!

\[
\begin{align*}
t_1 &= \text{int} \\
t_2 &= (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \\
t_3 &= \text{int} \\
t_4 &= \text{int} \rightarrow \text{int} \\
t_f &= \text{int} \rightarrow \text{int} \\
\end{align*}
\]

\((\lambda f:\text{int} \rightarrow \text{int}. \ f \ 5) \ (\lambda x:\text{int}. \ x+1) : \text{int} \ (t_1)\)
Another Example

- twice f x = f (f x)
- What is the type of \texttt{twice}?
Another Example

- \texttt{twice \, f \, x = f \, (f \, x)}
- What is the type of \texttt{twice}?
  - It is $t_f \rightarrow t_x \rightarrow t_1$ ($t_1$ is the type of $f \, (f \, x)$)
- Based on the syntax tree of $f \, (f \, x)$ we have:
  \[
  t_f = \underbrace{t_2 \rightarrow t_1}_{(1)}
  \]
  \[
  t_f = \underbrace{t_x \rightarrow t_2}_{(2)}
  \]
  Thus, $t_x = t_1 = t_2$, $t_f = t_x \rightarrow t_x$ and type of \texttt{twice} is $(t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x$

Note: $t_x$ is a free type variable! Polymorphism!
Type Constraints from Typing Rules, as Attribute Grammar

Syntax:  \[ E ::= x \mid c \mid \lambda x.E \mid E_1 E_2 \mid E_1 + E_2 \]

Grammar rule:  Attribute rule:

\[ E ::= x \quad C_E = \{ t_E = \Gamma_E(x) \} \]

\[ E ::= c \quad C_E = \{ t_E = \text{int} \} \]

\[ E ::= \lambda x.E_1 \quad \Gamma_{E_1} = \Gamma_E ; x : t_x \]

\[ C_E = C_{E_1} \cup \{ t_E = t_x \rightarrow t_{E_1} \} \]

\[ E ::= E_1 E_2 \quad \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \]

\[ C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \]

\[ E ::= E_1 + E_2 \quad \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \]

\[ C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = \text{int}, t_{E_2} = \text{int}, t_E = \text{int} \} \]
Type Constraints from Typing Rules, as Attribute Grammar

\[ E ::= \lambda x.E_1 \]
\[ \Gamma_{E_1} = \Gamma_E ; x : t_x \]
\[ C_E = C_{E_1} \cup \{ t_E = t_x \rightarrow t_{E_1} \} \]

\[ t_E \] is “fresh” type variable for term represented by \( E \)’s subtree.

\[ E ::= E_1 E_2 \]
\[ \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \]
\[ C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \ldots \]

\( \Gamma \) is inherited. Propagates top-down the tree.
\( C \) collects constraints. It is synthesized. Propagates bottom-up the tree.
Solving Constraints

Two key concepts

Equality

- What does it mean for two types to be equal?
- Structural equality (aka structural equivalence)

Unification

- Can two types be made equal by choosing appropriate substitutions for their type variables?
- Robinson’s unification algorithm (which you already know from Prolog!)
Equality and Unification

What does it mean for two types $\tau_a$ and $\tau_b$ to be equal?

Structural equality

- Suppose $\tau_a = t_1 \rightarrow t_2$
  
  $\tau_b = t_3 \rightarrow t_4$

- Structural equality entails

  $\tau_a = \tau_b$ means $t_1 \rightarrow t_2 = t_3 \rightarrow t_4$ iff $t_1 = t_3$ and $t_2 = t_4$
Equality and Unification

- Can two types be made equal by choosing appropriate substitutions for their type variables?
- Robinson’s unification algorithm
  - Suppose $\tau_a = \text{int} \rightarrow t_1$
    - $\tau_b = t_2 \rightarrow \text{bool}$
  - Can we unify $\tau_a$ and $\tau_b$? Yes, if $\text{bool} / t_1$ and $\text{int} / t_2$
  - Suppose $\tau_a = \text{int} \rightarrow t_1$
    - $\tau_b = \text{bool} \rightarrow \text{bool}$
  - Can we unify $\tau_a$ and $\tau_b$? No.

Program Analysis CSCI 4450/6450, A Milanova (based on MIT 2015 Program Analysis OCW)
Example

\[ t_1 \rightarrow \text{bool} = (\text{int} \rightarrow t_2) \rightarrow t_3 \]

Yes, if \text{int} \rightarrow t_2 / t_1 \text{ and bool} / t_3
Simple Type Substitution
(essential to define unification)

- **Language of types**
  \[ \tau ::= b \quad // \text{primitive type, e.g., int, bool} \]
  \[ | t \quad // \text{type variable} \]
  \[ | \tau \rightarrow \tau \quad // \text{function type} \]

- **A substitution is a map**
  - \( S : \text{Type Variable} \rightarrow \text{Type} \)
  - \( S = [\tau_1/t_1, \ldots, \tau_n/t_n] \quad // \text{substitute type } \tau_i \text{ for type var } t_i \)

- **A substitution instance** \( \tau' = S \tau \)
  - \( S = [ t_0 \rightarrow \text{bool} / t_1 ] \quad \tau = t_1 \rightarrow t_1 \quad \text{then} \)
  - \( S(\tau) = S(t_1 \rightarrow t_1) = (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) \)
Simple Type Substitution (essential to define unification)

- Substitutions can be composed
  - $S_1 = [ t_0 \rightarrow \text{bool} / t_1 ]$
  - $S_2 = [ \text{int} / t_0 ]$
  - $\tau = t_1 \rightarrow t_1$
  - $S_2 S_1(\tau) = S_2( S_1(t_1 \rightarrow t_1) ) = $
Examples

- Substitutions can be composed
  - $S_1 = [ \frac{t_x}{t_1} ]$
  - $S_2 = [ \frac{t_x}{t_2} ]$

- $\tau = t_2 \rightarrow t_1$
- $S_2 \, S_1(\tau) = ?$
Examples

- Substitutions can be composed
  - \( S_1 = [ t_1 / t_2 ] \)
  - \( S_2 = [ t_3 / t_1 ] \)
  - \( S_3 = [ t_4 \rightarrow \text{int} / t_3 ] \)

- \( \tau = t_1 \rightarrow t_2 \)
- \( S_3 S_2 S_1(\tau) = ? \)
Some Terminology...

- A substitution $S_1$ is **less specific** (i.e., more general) than substitution $S_2$ if $S_2 = S \cdot S_1$ for some substitution $S$
  - E.g., $S_1 = [ t_1 \rightarrow t_1 / t_2 ]$ is more general than $S_2 = [ \text{int} \rightarrow \text{int} / t_2 ]$ because $S_2 = S \cdot S_1$ for $S = [ \text{int} / t_1 ]$

- A **principal unifier** of a constraint set $C$ is a substitution $S_1$ that satisfies $C$, and $S_1$ is more general than any $S_2$ that satisfies $C$
Examples

Find principal unifiers (when they exist) for

- \{ \text{int} \rightarrow \text{int} = t_1 \rightarrow t_2 \}
- \{ \text{int} = \text{int} \rightarrow t_2 \}
- \{ t_1 = \text{int} \rightarrow t_2 \}

- \{ t_1 = \text{int}, t_2 = t_1 \rightarrow t_1 \}
- \{ t_1 \rightarrow t_2 = t_2 \rightarrow t_3, t_3 = t_4 \rightarrow t_5 \}
Unification
(essential for type inference!)

- **Unify**: tries to unify \(\tau_1\) and \(\tau_2\) and returns a principal unifier for \(\tau_1 = \tau_2\) if unification is successful.

```python
def Unify(\(\tau_1, \tau_2\)) =
  case \((\tau_1, \tau_2)\)
    \((\tau_1, t_2) = [\tau_1/t_2]\) provided \(t_2\) does not occur in \(\tau_1\)
    \((t_1, \tau_2) = [\tau_2/t_1]\) provided \(t_1\) does not occur in \(\tau_2\)
    \((b_1, b_2) = \text{if } (\text{eq? } b_1 b_2) \text{ then } [ ] \text{ else } \text{fail}\)
    \((\tau_{11} \rightarrow \tau_{12}, \tau_{21} \rightarrow \tau_{22}) = \text{let } S_1 = \text{Unify}(\tau_{11}, \tau_{21})
    S_2 = \text{Unify}(S_1(\tau_{12}), S_1(\tau_{22}))
    \text{in } S_2 S_1 // \text{compose substitutions}\)
  otherwise = \text{fail}
```

This is the occurs check!
Examples

- Unify \((\text{int} \rightarrow \text{int}, t_1 \rightarrow t_2)\) yields ?

- Unify \((\text{int}, \text{int} \rightarrow t_2)\) yields ?

- Unify \((t_1, \text{int} \rightarrow t_2)\) yields ?
Unify Set of Constraints C

- **UnifySet**: tries to unify C and returns a principal unifier for C if unification is successful.

```python
def UnifySet(C) =
  if C is Empty Set then []
  else let
    C = { τ₁=τ₂ } U C'
    S = Unify(τ₁,τ₂) // Unify returns a substitution S
  in
    UnifySet(S(C')) ° S
// Composition of substitutions
```
Examples

- \{ t_1 = \text{int}, t_2 = t_1 \rightarrow t_1 \} \\
- \{ t_1 \rightarrow t_2 = t_2 \rightarrow t_3, t_3 = t_4 \rightarrow t_5 \} \\
- \{ t_f = t_2 \rightarrow t_1, t_f = t_x \rightarrow t_2 \} \\
- \{ t_2 = t_4 \rightarrow t_1, t_2 = t_f \rightarrow t_3, t_4 = t_x \rightarrow t_5, t_f = \text{int} \rightarrow t_3, t_5 = \text{int}, t_x = \text{int} \}
Type Inference, Strategy 1

- Aka constraint-based typing (e.g., Pierce)

- Traverse parse tree to derive a set of type constraints $C$
  - These are equality constraints
  - (Pseudo code in earlier slides)

- Solve type constraints offline
  - Use unification algorithm
  - (Pseudo code in slide 36)
Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism
- Hindley Milner type inference. Algorithm W
Type Inference, Strategy 2

- Strategy 1 collects all constraints, then solves them offline

- Strategy 2 solves constraints on the fly
  - Builds the substitution map incrementally
  - Key reason: infers types as parser parses program!
Add a New Attribute, Substitution Map $S$

Grammar rule:  

\[
E ::= x
\]

\[
E ::= c
\]

\[
E ::= \lambda x. E_1
\]

\[
E ::= E_1 E_2
\]

Attribute rule:  

\[
T_E = \Gamma_E(x) \quad S_E = [ ]
\]

\[
T_E = \text{int} \quad S_E = [ ]
\]

\[
\Gamma_{E_1} = \Gamma_E; x: t_x
\]

\[
T_E = S_{E_1}(t_x) \rightarrow T_{E_1} \quad S_E = S_{E_1}
\]

\[
\Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = S_{E_1}(\Gamma_E)
\]

\[
S = \text{Unify}(S_{E_2}(T_{E_1}), T_{E_2} \rightarrow t_E)
\]

\[
T_E = S(t_E) \quad S_E = S S_{E_2} S_{E_1}
\]

$T_E$ is the inferred type of $E$. $S_E$ is the substitution map resulting from inferring $T_E$. $t_x, t_E$ are fresh type variables.
Example: \((\lambda f. f \ 5) \ (\lambda x. \ x)\)

\((\lambda f. f \ 5) \ (\lambda x. \ x) : ?\)

1. App

\[\Gamma_1 = []\]
\[T_1 = \text{int}\]
\[S_1 = [\ \text{int/t}_x, \ \text{int/t}_3, \ \text{int/t}_1, \ \text{int} \rightarrow \text{int}/t_f]\]

2. Abs

\[\Gamma_2 = []\]
\[T_2 = (\text{int} \rightarrow \text{t}_3) \rightarrow \text{t}_3\]
\[S_2 = [\ \text{int} \rightarrow \text{t}_3/t_f]\]

3. App

\[\Gamma_3 = [f:t_f]\]
\[T_3 = \text{t}_3\]
\[S_3 = [\ \text{int} \rightarrow \text{t}_3/t_f]\]

4. Abs

\[\Gamma_4 = S_2(\Gamma_1) = []\]
\[T_4 = \text{t}_x \rightarrow \text{t}_x\]
\[S_4 = []\]

\[\Gamma = [x:t_x]\]
\[T = \text{t}_x\]
\[S = []\]

from Unify\((t_f, \text{int} \rightarrow \text{t}_3)\)
Example: $\lambda f. \lambda x. (f (f x))$
The Let Construct

- In dynamic semantics, \( \text{let } x = E_1 \text{ in } E_2 \) is equivalent to \( (\lambda x.E_2) \ E_1 \)

- Typing rule

\[
\Gamma \vdash E_1 : \sigma \quad \Gamma; x: \sigma \vdash E_2 : \tau \\
\hline
\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau
\]

- In static semantics \( \text{let } x = E_1 \text{ in } E_2 \) is not equivalent to \( (\lambda x.E_2) \ E_1 \)
  - In \textit{let}, the type of “argument” \( E_1 \) is inferred/checked \textit{before} the type of function body \( E_2 \)
  - \textit{let} construct enables Hindley Milner style polymorphism!
The Let Construct

- Typing rule

\[ \Gamma |- E_1 : \sigma \quad \Gamma; x: \sigma |- E_2 : \tau \]
\[ \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau \]

- Attribute grammar rule

\[ E ::= \text{let } x = E_1 \text{ in } E_2 \]
\[ \Gamma_{E_1} = \Gamma_E \]
\[ \Gamma_{E_2} = S_{E_1}(\Gamma_E) + \{x: T_{E_1}\} \]
\[ T_E = T_{E_2} \quad S_E = S_{E_2} S_{E_1} \]
The Letrec Construct

- **letrec** \( x = E_1 \) in \( E_2 \)
  - \( x \) can be referenced from within \( E_1 \)
  - Extends calculus with general recursion
    - No need to type **fix** (we can’t!) but we can still type recursive functions like **plus**, **times**, etc.
  - Haskell’s **let** is a **letrec** actually…

- E.g.,
  letrec **plus** = \( \lambda x. \lambda y. \) if (\( x=0 \)) then \( y \) else ((**plus** \( x-1 \)) \( y+1 \))
  written as
  letrec **plus** \( x \) \( y \) = if (\( x=0 \)) then \( y \) else **plus** \( x-1 \) \( y+1 \)
The Letrec Construct

- **letrec** \( x = E_1 \) in \( E_2 \)

Attribute grammar rule

\( E ::= \text{letrec} \ x = E_1 \) in \( E_2 \)

- Extensions over let rule
  1. \( T_{E_1} \) is inferred in augmented environment \( \Gamma_E + \{x: t_x\} \)
  2. Must unify \( S_{E_1}(t_x) \) and \( T_{E_1} \)
  3. Apply substitution \( S \) on top of \( S_{E_1} \)

Note: Can merge **let** and **letrec**, in **let**

**Unify** and \( S \) have no impact
let/letrec Examples

letrec plus x y = if (x=0) then y else plus (x-1) (y+1)

- Typing **plus** using Strategy 1...

  \[ t_{plus} = t_x \rightarrow t_y \rightarrow t_1 \]
  \[ t_x = \text{int} \quad \text{// because of } x=0 \text{ and } x-1 \]
  \[ t_y = \text{int} \quad \text{// because of } y+1 \]
  \[ \text{Unify}(t_{plus}, \text{int} \rightarrow \text{int} \rightarrow \text{int}) \text{ yields } t_1 = \text{int} \]

- Haskell

  \[ \text{plus :: int -> int -> int} \]
  \[ \text{plus x y = if (x=0) then y else plus (x-1) (y+1)} \]
def W(Γ, E) = case E of
    c  -> ([], TypeOf(c))
    x  -> if (x NOT in Dom(Γ)) then fail
           else let T_E = Γ(x);
               in ([], T_E)
    λx.E_1 -> let (S_E1, T_E1) = W(Γ + {x:t_x}, E_1)
                in (S_E1, S_E1(t_x)→T_E1)
    E_1 E_2 -> let (S_E1, T_E1) = W(Γ, E_1)
                (S_E2, T_E2) = W(S_E1(Γ), E_2)
                S = Unify(S_E2(T_E1), T_E2→t)
                in (S S_E2 S_E1, S(t)) // S S_E2 S_E1 composes substitutions
    let x = E_1 in E_2 -> let (S_E1, T_E1) = W(Γ, E_1)
                        (S_E2, T_E2) = W(S_E1(Γ) + {x:T_E1}, E_2)
                        in (S_E2 S_E1, T_E2)
def W(Γ, E) = case E of
  c   -> ([], TypeOf(c))
  x   -> if (x NOT in Dom(Γ)) then fail
       else let T_E = Γ(x);
           in ([], T_E)
  λx.E_1 -> let (S_E1, T_E1) = W(Γ+{x:t_x},E_1)
       in (S_E1, S_E1(t_x)→T_E1)
  E_1 E_2 -> let (S_E1, T_E1) = W(Γ,E_1)
             (S_E2, T_E2) = W(S_E1(Γ),E_2)
             S = Unify(S_E2(T_E1),T_E2→t)
             in (S S_E2 S_E1, S(t)) // S S_E2 S_E1 composes substitutions
  let x = E_1 in E_2 -> let (S_E1, T_E1) = W(Γ+{x:t_x},E_1)
                       S = Unify(S_E1(t_x),T_E1)
                       (S_E2, T_E2) = W(S S_E1(Γ)+{x:T_E1},E_2)
           in (S_E2 S S_E1, T_E2)
Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism
- Hindley Milner type inference. Algorithm W

Program Analysis CSCI 4450/6450, A Milanova