Type Inference

COS 326
David Walker
Princeton University
Midterm Exam

Wed Oct 25, 2017
In Class (11:00-12:20)
Midterm Week

Be there or be square!
TYPE INFERENCE
The ML language and type system is designed to support a very strong form of type inference.
The ML language and type system is designed to support a very strong form of type inference.

```ml
let rec map f l =
  match l with
  [ ] -> [ ]
  | hd::tl -> f hd :: map f tl
```

ML finds this type for map:

```
map : ('a -> 'b) -> 'a list -> 'b list
```
The ML language and type system is designed to support a very strong form of type inference.

ML finds this type for `map`:

```ocaml
let rec map f l =
  match l with
  [ ] -> [ ]
| hd::tl -> f hd :: map f tl
```

which is really an abbreviation for this type:

```ocaml
map : forall 'a,'b.('a -> 'b) -> 'a list -> 'b list
```
Language Design for Type Inference

We call this type the *principle type (scheme)* for map.

Any other ML-style type you can give map is *an instance* of this type, meaning we can obtain the other types via *substitution* of types for parameters from the principle type.

Eg:

(map : ('a -> 'b) -> 'a list -> 'b list)

(bool -> int) -> bool list -> int list

('a -> int) -> 'a list -> int list

('a -> 'a) -> 'a list -> 'a list
Principle types are great:

• the type inference engine can make a *best choice* for the type to give an expression

• the engine doesn't have to guess (and won't have to guess wrong)

The fact that principle types exist is surprisingly brittle. If you change ML's type system a little bit in either direction, it can fall apart.
Language Design for Type Inference

Suppose we take out polymorphic types and need a type for id:

\[
\text{let id } x = x
\]

Then the compiler might guess that id has one (and only one) of these types:

\[
id : \text{bool} \rightarrow \text{bool}
\]

\[
id : \text{int} \rightarrow \text{int}
\]
Suppose we take out polymorphic types and need a type for \texttt{id}:

\begin{verbatim}
let id \ x = x
\end{verbatim}

Then the compiler might guess that \texttt{id} has one (and only one) of these types:

\begin{verbatim}
id : bool \rightarrow bool
\end{verbatim}

\begin{verbatim}
id : int \rightarrow int
\end{verbatim}

But later on, one of the following code snippets won't type check:

\begin{verbatim}
id true
\end{verbatim}

\begin{verbatim}
id 3
\end{verbatim}

So whatever choice is made, a different one might have been better.
Language Design for Type Inference

We showed that removing types from the language causes a failure of principle types.

Does adding more types always make type inference easier?
Language Design for Type Inference

We showed that removing types from the language causes a failure of principle types.

Does adding more types always make type inference easier?

Nope!
Ocaml only has universal types on the outside:

Consider this program:

```
let f g = (g true, g 3)
```

It won't type check in Ocaml. We might want to give it this type:

```
f : (forall 'a.'b. ('a -> 'b) -> 'a list -> 'b list) -> bool * int
```

Notice that the universal quantifier appears under an `->`
System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check f.

```ocaml
let f g = (g true, g 3)
```

```ocaml
f : (forall a.a->a) -> bool * int
```

Unfortunately, type inference in System F is undecideable.
System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check f.

\[
\text{let } f \ g \ = \ (g \ \text{true}, \ g \ 3)
\]

\[
f : (\forall a. a \rightarrow a) \rightarrow \text{bool} \times \text{int}
\]

Unfortunately, type inference in System F is undecideable.

Developed in 1972 by logician Jean Yves-Girard who was interested in the consistency of a logic of 2\textsuperscript{nd}-order arithmetic.

Rediscovered as programming language by John Reynolds in 1974.
Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```
let f x = x + x
```
Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```
let f x = x + x

f : int -> int  ?
```

```
f : float -> float  ?
```
Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```
let f x = x + x
f : int -> int  ?

f : float -> float  ?

f : 'a -> 'a  ?
```
Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```ocaml
let f x = x + x
f : int -> int
f : float -> float
f : 'a -> 'a
```

No type in OCaml's type system works. In Haskell:

```haskell
f : Num 'a => 'a -> 'a
```
INFERRING SIMPLE TYPES
A **type scheme** contains type variables that may be filled in during type inference.

\[ s ::= a \mid \text{int} \mid \text{bool} \mid s \to s \]

A **term scheme** is a term that contains type schemes rather than proper types. eg, for functions:

```
fun (x:s) -> e
```

```
let rec f (x:s) : s = e
```
The Generic Type Inference Algorithm

1) Add distinct variables in all places type schemes are needed
The Generic Type Inference Algorithm

1) Add distinct variables in all places type schemes are needed

2) Generate constraints (equations between types) that must be satisfied in order for an expression to type check
   • Notice the difference between this and the type checking algorithm from last time. Last time, we tried to:
     • eagerly deduce the concrete type when checking every expression
     • reject programs when types didn't match. eg:
       
       \[ f \ e \quad \text{-- f's argument type must equal e} \]
     • This time, we'll collect up equations like:
       
       \[ a \rightarrow b = c \]
The Generic Type Inference Algorithm

1) Add distinct variables in all places type schemes are needed

2) Generate constraints (equations between types) that must be satisfied in order for an expression to type check
   • Notice the difference between this and the type checking algorithm from last time. Last time, we tried to:
     • eagerly deduce the concrete type when checking every expression
     • reject programs when types didn't match. eg:

   f e -- f's argument type must equal e

   • This time, we'll collect up equations like:

   a -> b = c

3) Solve the equations, generating substitutions of types for var's
Example: Inferring types for map

```
let rec map f l =
  match l with
  | [] -> []
  | hd::tl -> f hd :: map f tl
```
let rec map (f : a) (l : b) : c =
  match l with
  | [] -> []
  | hd::tl -> f hd :: map f tl
Step 2: Generate Constraints

let rec map (f:a) (l:b) : c =
    match l with
    [] -> []
    | hd::tl -> f hd :: map f tl

b = d list
a = d -> f
...

...
Step 2: Generate Constraints

```
let rec map (f:a) (l:b) : c =
  match l with
  | [] -> []
  | hd::tl -> f hd ::: map f tl
```

```
final constraints:
```

```
b = b' list
b = b'' list
b = b''' list
a = a
b = b''' list
a = b'' -> a'
c = c' list
a' = c'
d list = c' list
d list = c
```
Step 3: Solve Constraints

```
let rec map (f:a) (l:b) : c =
match l with
  [] -> []
| hd::tl -> f hd :: map f tl
```

**final constraints:**
- \( b = b' \text{ list} \)
- \( b = b'' \text{ list} \)
- \( b = b''' \text{ list} \)
- \( a = a \)
- \( b = b''' \text{ list} \)
- \( a = b'' \rightarrow a' \)
- \( c = c' \text{ list} \)
- \( a' = c' \)
- \( d \text{ list }= c' \text{ list} \)
- \( d \text{ list }= c \)

**final solution:**
- \([b' \rightarrow c'/a]\)
- \([b' \text{ list}/b]\)
- \([c' \text{ list}/c]\)
Step 3: Solve Constraints

let rec map (f:a) (l:b) : c =
  match l with
  [] -> []
  | hd::tl -> f hd :: map f tl

final solution:
[b' -> c'/a]
[b' list/b]
[c' list/c]

let rec map (f:b' -> c') (l:b' list) : c' list =
  match l with
  [] -> []
  | hd::tl -> f hd :: map f tl
Step 3: Solve Constraints

let rec map (f:a) (l:b) : c =
    match l with
    | [] -> []
    | hd::tl -> f hd :: map f tl

renaming type variables:

let rec map (f:a -> b) (l:a list) : b list =
    match l with
    | [] -> []
    | hd::tl -> f hd :: map f tl
Step 4: Generate types

Generate types from type schemes

– **Option 1**: pick an instance of the most general type when we have completed type inference on the entire program
  
  • $\text{map} : (\text{int} \to \text{int}) \to \text{int list} \to \text{int list}$

– **Option 2**: generate polymorphic types for program parts and continue (polymorphic) type inference
  
  • $\text{map} : \forall \text{a,b,c}. (\text{a} \to \text{b}) \to \text{a list} \to \text{b list}$
Type Inference Details

Type constraints are sets of equations between type schemes

- $q ::= \{s_1 = s_2, \ldots, s_n = s_n\}$

- eg: $\{b = \text{b’ list}, a = b \to c\}$
Syntax-directed constraint generation

- our algorithm crawls over abstract syntax of untyped expressions and generates
  - a term scheme
  - a set of constraints

Algorithm defined as set of inference rules:

\[ G \vdash u \Rightarrow e : t, q \]

Constraints that must be solved

In OCaml:

```ocaml
gen : ctxt -> exp -> ann_exp * scheme * constraints
```
Constraint Generation

Simple rules:

- $G |-- x \Rightarrow x : s, \{ \}$  \hspace{1em} (if $G(x) = s$)

- $G |-- 3 \Rightarrow 3 : \text{int}, \{ \}$  \hspace{1em} (same for other ints)

- $G |-- \text{true} \Rightarrow \text{true} : \text{bool}, \{ \}$

- $G |-- \text{false} \Rightarrow \text{false} : \text{bool}, \{ \}$
 Operators

G |-- u1 ==> e1 : t1, q1         G |-- u2 ==> e2 : t2, q2
------------------------------------------------------------------------
G |-- u1 + u2 ==> e1 + e2 : int, q1 U q2 U {t1 = int, t2 = int}
------------------------------------------------------------------------
G |-- u1 ==> e1 : t1, q1         G |-- u2 ==> e2 : t2, q2
-----------------------------------------------
G |-- u1 < u2 ==> e1 + e2 : bool, q1 U q2 U {t1 = int, t2 = int}
If statements

\[ G |-- u_1 ==> e_1 : t_1, q_1 \]
\[ G |-- u_2 ==> e_2 : t_2, q_2 \]
\[ G |-- u_3 ==> e_3 : t_3, q_3 \]

\[ G |-- \text{if } u_1 \text{ then } u_2 \text{ else } u_3 \implies \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

: \text{a, q_1 U q_2 U q_3 U \{t_1 = \text{bool}, a = t_2, a = t_3\}}
G |-- u1 ==> e1 : t1, q1
G |-- u2 ==> e2 : t2, q2 (for a fresh a)

G |-- u1 u2 ==> e1 e2

: a, q1 U q2 U {t1 = t2 -> a}
G, x : a |-- u ==> e : t, q  \hspace{1cm} \text{(for fresh a)}

\begin{align*}
\text{-----------------------------} \\
G |-- \text{fun } x \rightarrow e == \rightarrow \text{fun (x : a)} \rightarrow e \\
\end{align*}

: a --> b, q U \{t = b\}
G, f : a -> b, x : a |-- u ==> e : t, q           (for fresh a,b)
-----------------------------------------------------------------------
G |-- rec f(x) = u ==> rec f (x : a) : b = e
: a -> b, q U {t = b}
A solution to a system of type constraints is a substitution $S$

- a function from type variables to types
- assume substitutions are defined on all type variables:
  - $S(a) = a$  (for almost all variables $a$)
  - $S(a) = s$  (for some type scheme $s$)
- $\text{dom}(S) =$ set of variables s.t. $S(a) \neq a$
Solving Constraints

A solution to a system of type constraints is a substitution $S$

- a function from type variables to types
- assume substitutions are defined on all type variables:
  - $S(a) = a$ (for almost all variables $a$)
  - $S(a) = s$ (for some type scheme $s$)
- $\text{dom}(S) = \text{set of variables s.t. } S(a) \neq a$

We can also apply a substitution $S$ to a full type scheme $s$.

apply: $[ \text{int}/a, \text{int}->\text{bool}/b ]$

to: $b -> a -> b$

returns: $(\text{int}->\text{bool}) -> \text{int} -> (\text{int}->\text{bool})$
We can apply a substitution $S$ to a full type scheme:

**Example:** apply $[\text{int}/a, \text{int} \rightarrow \text{bool}/b]$ to $b \rightarrow a \rightarrow b$

returns: $(\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow (\text{int} \rightarrow \text{bool})$
Substitutions

When is a substitution $S$ a solution to a set of constraints?

Constraints: \( \{ s_1 = s_2, s_3 = s_4, s_5 = s_6, \ldots \} \)

When the substitution makes both sides of all equations the same.

Eg:

constraints:
\[
\begin{align*}
a &= b ightarrow c \\
c &= \text{int} \rightarrow \text{bool}
\end{align*}
\]
When is a substitution $S$ a solution to a set of constraints?

Constraints: $\{ s_1 = s_2, s_3 = s_4, s_5 = s_6, \ldots \}$

When the substitution makes both sides of all equations the same.

Eg:

<table>
<thead>
<tr>
<th>constraints:</th>
<th>solution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b \rightarrow c$</td>
<td>$b \rightarrow (\text{int} \rightarrow \text{bool})/a$</td>
</tr>
<tr>
<td>$c = \text{int} \rightarrow \text{bool}$</td>
<td>$\text{int} \rightarrow \text{bool}/c$</td>
</tr>
<tr>
<td>$b/b$</td>
<td>$b/b$</td>
</tr>
</tbody>
</table>
Substitutions

When is a substitution S a solution to a set of constraints?

Constraints: \{ s1 = s2, s3 = s4, s5 = s6, ... \}

When the substitution makes both sides of all equations the same.

Eg:

constraints:

\[
\begin{align*}
    a & \rightarrow c \\
    c & \rightarrow \text{int} \rightarrow \text{bool}
\end{align*}
\]

solution:

\[
\begin{align*}
    b & \rightarrow (\text{int} \rightarrow \text{bool}) / a \\
    \text{int} & \rightarrow \text{bool} / c \\
    b / b
\end{align*}
\]

constraints with solution applied:

\[
\begin{align*}
    b & \rightarrow (\text{int} \rightarrow \text{bool}) = b \rightarrow (\text{int} \rightarrow \text{bool}) \\
    \text{int} & \rightarrow \text{bool} = \text{int} \rightarrow \text{bool}
\end{align*}
\]
Substitutions

When is a substitution \( S \) a solution to a set of constraints?

Constraints: \( \{ s1 = s2, s3 = s4, s5 = s6, \ldots \} \)

When the substitution makes both sides of all equations the same.

A second solution

Constraints:

- \( a = b \to c \)
- \( c = \text{int} \to \text{bool} \)

Solution 1:

- \( b \to (\text{int} \to \text{bool})/a \)
- \( \text{int} \to \text{bool}/c \)
- \( b/b \)

Solution 2:

- \( \text{int} \to (\text{int} \to \text{bool})/a \)
- \( \text{int} \to \text{bool}/c \)
- \( \text{int}/b \)
When is one solution better than another to a set of constraints?

constraints:

\[ a = b \rightarrow c \]
\[ c = \text{int} \rightarrow \text{bool} \]

solution 1:

\[ b \rightarrow (\text{int} \rightarrow \text{bool})/a \]
\[ \text{int} \rightarrow \text{bool}/c \]
\[ b/b \]

solution 2:

\[ \text{int} \rightarrow (\text{int} \rightarrow \text{bool})/a \]
\[ \text{int} \rightarrow \text{bool}/c \]
\[ \text{int}/b \]

type b -> c with solution applied:

solution 1:

\[ b \rightarrow (\text{int} \rightarrow \text{bool}) \]

solution 2:

\[ \text{int} \rightarrow (\text{int} \rightarrow \text{bool}) \]
Substitutions

Solution 1 is "more general" – there is more flex.
Solution 2 is "more concrete"
We prefer solution 1.
Solution 1 is "more general" – there is more flex.
Solution 2 is "more concrete"
We prefer the more general (less concrete) solution 1.
Technically, we prefer T to S if there exists another substitution U and for all types t, S (t) = U (T (t))
There is always a **best** solution, which we can a *principle solution*. The best solution is (at least as) preferred as any other solution.
S is the principal (most general) solution of a constraint q if
- $S \models q$ (it is a solution)
- if $T \models q$ then $T \leq S$ (it is the most general one)

Lemma: If q has a solution, then it has a most general one

We care about principal solutions since they will give us the most general types for terms
Composition of Substitutions

We will need to compare substitutions: \( T \leq S \). eg:

- \( T \leq S \) if \( T \) is “more specific”/"less general" than \( S \)
- If there is a

- Formally: \( T \leq S \) if and only if \( T = U \circ S \) for some \( U \)
Composition of Substitutions

Composition \((U \circ S)\) applies the substitution \(S\) and then applies the substitution \(U\):

\[- (U \circ S)(a) = U(S(a))\]

We will need to compare substitutions

\[- T \leq S \text{ if } T \text{ is “more specific” than } S\]

\[- T \leq S \text{ if } T \text{ is “less general” than } S\]

\[- \text{Formally: } T \leq S \text{ if and only if } T = U \circ S \text{ for some } U\]
Composition of Substitutions

Examples:

– example 1: any substitution is less general than the identity substitution I:
  • $S \leq I$ because $S = S \circ I$

– example 2:
  • $S(a) = \text{int}, S(b) = \text{c} \rightarrow \text{c}$
  • $T(a) = \text{int}, T(b) = \text{int} \rightarrow \text{int}$
  • we conclude: $T \leq S$
  • if $T(a) = \text{int}, T(b) = \text{int} \rightarrow \text{bool}$ then $T$ is unrelated to $S$ (neither more nor less general)
Solving a Constraint

S |= q if S is a solution to the constraints q

\[
\begin{align*}
S(s1) &= S(s2) & S &= q \\
\quad &= \{s1 = s2\} \cup q
\end{align*}
\]

- any substitution is a solution for the empty set of constraints
- a solution to an equation is a substitution that makes left and right sides equal
Example 1

- q = \{a=\text{int}, \, b=a\}
- principal solution S:
Example 1

- \( q = \{a=\text{int}, \ b=a\} \)
- principal solution \( S \):
  - \( S(a) = S(b) = \text{int} \)
  - \( S(c) = c \) (for all \( c \) other than \( a,b \))
Example 2

- \( q = \{ a=\text{int}, \ b=a, \ b=\text{bool} \} \)
- principal solution \( S \):
Example 2

- $q = \{a=\text{int}, \ b=a, \ b=\text{bool}\}$
- principal solution $S$:
  - does not exist (there is no solution to $q$)
**Unification**

Unification: An algorithm that provides the principal solution to a set of constraints (if one exists)

- Unification systematically simplifies a set of constraints, yielding a substitution
  - Starting state of unification process: \((I, q)\)
  - Final state of unification process: \((S, \{\})\)
We can specify unification as a transition system:

\[(S_1, q_1) \rightarrow (S_2, q_2)\]

Base types & simple variables:

\[(S, \{\text{bool} = \text{bool}\} \cup q) \rightarrow (S, q)\]

\[(S, \{\text{int} = \text{int}\} \cup q) \rightarrow (S, q)\]

\[(S, \{\text{a} = \text{a}\} \cup q) \rightarrow (S, q)\]
Unification Machine

Functions:

(S, \{s11 -> s12 = s21 -> s22\} U q) ->
(S, \{s11 = s21, s12 = s22\} U q)

when a is not in FreeVars(s)

breaks down into smaller constraints

Variable definitions

(S, \{a=s\} U q) -> ([a=s] o S, [s/a]q)

(S, \{s=a\} U q) -> ([a=s] o S, [s/a]q)

do S and then do [a=b]
Recall this program:

\[
\text{fun } x \rightarrow x \times x
\]

It generates the constraints: \( a \rightarrow a = a \)

What is the solution to \( \{ a = a \rightarrow a \} \)?
Recall this program:

\[
\text{fun } x \rightarrow x \times \times
\]

It generates the constraints: \(a \rightarrow a = a\)

What is the solution to \(\{a = a \rightarrow a\}\)?

There is none!

\[
(S, \{s=a\} \cup q) \rightarrow ([a=s] \circ S, [s/a]q)
\]

"when a is not in FreeVars(s)"

the "occurs check"
Irreducible States

When all the constraints have been processed, we win!

But sometimes we get stuck, with an equation like this

\[\text{int} = \text{bool} \]
\[\text{s1} \rightarrow \text{s2} = \text{int} \]
\[\text{s1} \rightarrow \text{s2} = \text{bool} \]
\[\text{a} = \text{s} \quad (\text{s} \text{ contains } \text{a}) \]
\[\text{or is symmetric to one of the above} \]

Stuck states arise when constraints are unsolvable & the program does not type check.
Termination

We want unification to terminate (to give us a type reconstruction algorithm)

In other words, we want to show that there is no infinite sequence of states

- \((S_1, q_1) \rightarrow (S_2, q_2) \rightarrow \ldots\)
We associate an ordering with constraints

- $q < q'$ if and only if
  
  - $q$ contains fewer variables than $q'$
  
  - $q$ contains the same number of variables as $q'$ but fewer type constructors (i.e., fewer occurrences of int, bool, or “-“)

- This is a lexicographic ordering
  
  - we can prove (by contradiction) that there is no infinite decreasing sequence of constraints
Lemma: Every step reduces the size of q
  – Proof: By cases (i.e., induction) on the definition of the reduction relation.

\[
\begin{align*}
(S, \{\text{int} = \text{int}\} \cup q) &\rightarrow (S, q) \\
(S, \{\text{bool} = \text{bool}\} \cup q) &\rightarrow (S, q) \\
(S, \{a = a\} \cup q) &\rightarrow (S, q) \\
(S, \{s_{11} \rightarrow s_{12} = s_{21} \rightarrow s_{22}\} \cup q) &\rightarrow (S, \{s_{11} = s_{21}, s_{12} = s_{22}\} \cup q) \\
(S, \{s = s\} \cup q) &\rightarrow ([s = s] \circ S, [s/a] q)
\end{align*}
\]
A complete solution for \((S,q)\) is a substitution \(T\) such that

- \(T \leq S\)
- \(T \models q\)
Lemma 1:

- Every final state \((S, \{ \})\) has a complete solution.
  - It is \(S\):
    - \(S \subseteq S\)
    - \(S \models \{\}\)
Lemma 2

– No stuck state has a complete solution (or any solution at all)
  • it is impossible for a substitution to make the necessary equations equal
    – int ≠ bool
    – int ≠ t1 -> t2
    – ...

Properties of Solutions
Lemma 3

– If \((S,q) \rightarrow (S',q')\) then

  • \(T\) is complete for \((S,q)\) iff \(T\) is complete for \((S',q')\)
  • proof by?
  • in the forward direction, this is the preservation theorem for the unification machine!
By termination, \((I,q) \rightarrow^* (S,q')\) where \((S,q')\) is irreducible. Moreover:

- If \(q' = \{\}\) then
  
  - \((S,q')\) is final (by definition)
  - \(S\) is a principal solution for \(q\)
    
    - Consider any \(T\) such that \(T\) is a solution to \(q\).
    
    - Now notice, \(S\) is complete for \((S,q')\) (by lemma 1)
    
    - \(S\) is complete for \((I,q)\) (by lemma 3)
    
    - Since \(S\) is complete for \((I,q)\), \(T \leq S\) and therefore \(S\) is principal.
... Moreover:

- If $q'$ is not $\{\}$ (and $(I,q) \rightarrow^* (S,q')$ where $(S,q')$ is irreducible) then
  
  * $(S,q)$ is stuck. Consequently, $(S,q)$ has no complete solution. By lemma 3, even $(I,q)$ has no complete solution and therefore $q$ has no solution at all.
MORE TYPE INFERENCE
Where do we introduce polymorphic values?

\[
\text{let } x = v \quad \implies \quad \text{let } x : \forall a_1, \ldots, a_n. s = v
\]

\[
\text{if } v : s \text{ and } a_1, \ldots, a_n \text{ are the variables of } s
\]

And place \( x : \forall a_1, \ldots, a_n. s \) in the context.
Where do we introduce polymorphic values?

\[
\text{let } x = v \implies \text{let } x : \forall a_1, \ldots, a_n.s = v
\]

if \( v : s \) and \( a_1, \ldots, a_n \) are the variables of \( s \)

And place \( x : \forall a_1, \ldots, a_n.s \) in the context.

Where and how do we use a polymorphic value?

\[
G |- x \implies x : s[b_1/a_1, \ldots, b_n/a_n], \{}
\]

when \( G(x) = \forall a_1, \ldots, a_n.s \) and \( b_1, \ldots, b_n \) are fresh
What is the cost of type inference?

In practice? Linear in the size of the program

In theory, DEXPTIME-complete.

Why? Because we can generate a program that has a type that is exponentially large:

```
let f1 x = x       f1 : a -> a
let f2 = f1 f1     f2 : (a -> a) -> (a -> a)
let f3 = f2 f2     f3 : ((a -> a) -> (a -> a)) ->
                      ((a -> a) -> (a -> a))
let f4 = f3 f3
...
```
Summary: Type Inference

Given a context G, and untyped term u:

- Find $e, t, q$ such that $G |- u \Rightarrow e : t, q$

- Find principal solution $S$ of $q$ via unification
  - if no solution exists, there is no reconstruction

- Apply $S$ to $e$, ie our solution is $S(e)$
  - $S(e)$ contains schematic type variables $a,b,c$, etc that may be instantiated with any type

- Since $S$ is principal, $S(e)$ characterizes all reconstructions.

- If desired, use the type checking algorithm to validate