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

Language Design for Type Inference

The ML language and type system is designed to support a very strong form of type inference.

Language Design for Type Inference

The ML language and type system is designed to support a very strong form of type inference.

```
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
```

Language Design for Type Inference

The ML language and type system is designed to support a very strong form of type inference.

```
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
```

which is really an abbreviation for this type:

```
map : forall 'a,'b.('a -> 'b) -> 'a list -> 'b list
```

Language Design for Type Inference

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

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:

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

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

```
('a -> 'a) -> 'a list -> 'a list
```

Language Design for Type Inference

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:

```
let id x = x
```

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

```
id : bool -> bool
```

```
id : int -> int
```

Language Design for Type Inference

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

```
let id x = x
```

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

```
id : bool -> bool
```

```
id : int -> int
```

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

```
id true
```

```
id 3
```

So whatever choice is made, a different one might have been better.

Language Design for Type Inference

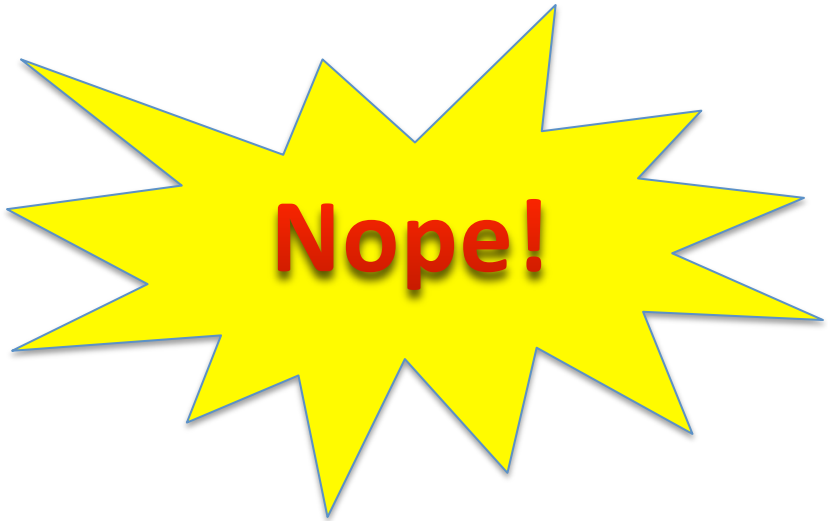
We showed that removing types from the language causes a failure of principle types.

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?



Language Design for Type Inference

OCaml only has universal types on the outside:

```
forall 'a,'b. ('a -> 'b) -> 'a list -> 'b list
```

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.a->a) -> bool * int
```

Notice that the universal quantifier appears under an ->

Language Design for Type Inference

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

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

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

Unfortunately, type inference in System F is undecideable.

.

Language Design for Type Inference

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

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

```
f : (forall a.a->a) -> bool * 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 2nd-order arithemetic.

Rediscovered as programming language by John Reynolds in 1974.

John C. Reynolds (John Barna photo)

Language Design for Type Inference

Even seemingly small changes can effect type inference.

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

```
let f x = x + x
```

Language Design for Type Inference

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   ?
```

Language Design for Type Inference

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  ?
```

Language Design for Type Inference

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  ?
```

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

```
f : Num 'a => 'a -> 'a
```

INFERRING SIMPLE TYPES

Type Schemes

A *type scheme* contains type variables that may be filled in during type inference

s ::= *a* | int | bool | s -> 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    -- f's argument type must equal e
```

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

```
a -> 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
```

Step 1: Annotate

```
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' 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
```

final solution:

```
[b' -> c'/a]
[b' list/b]
[c' 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
  - map : (int -> int) -> int list -> int list

- Option 2: generate polymorphic types for program parts and continue (polymorphic) type inference
  - map : forall a,b,c. (a -> b) -> a list -> b list

Type Inference Details

Type constraints are sets of equations between type schemes

- q ::= {s11 = s12, ..., sn1 = sn2}
- eg: {b = b' list, a = b -> c}

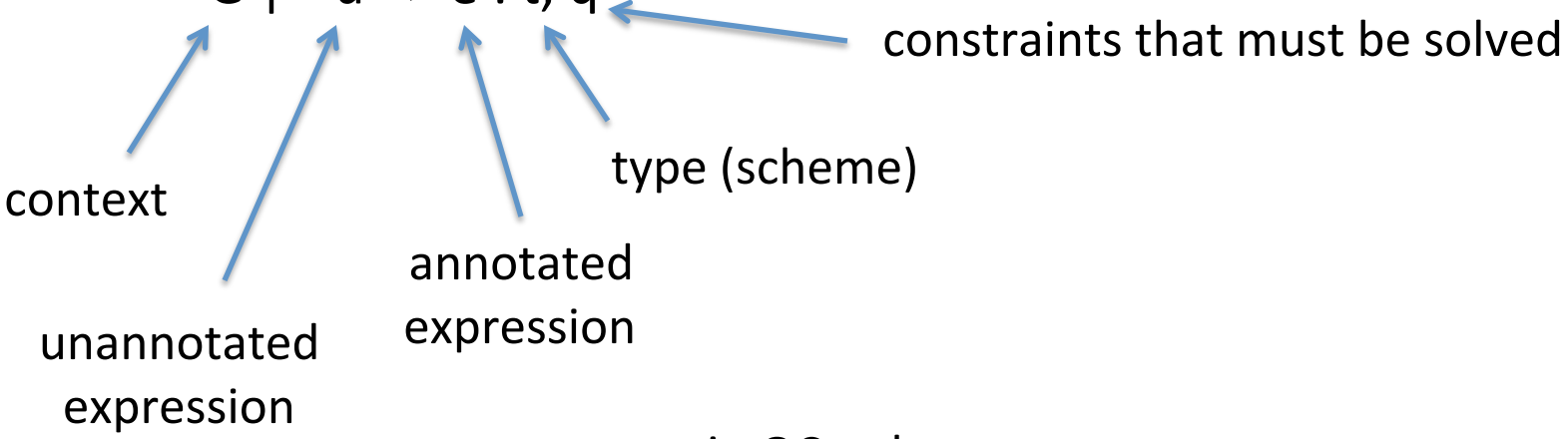
Constraint Generation

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 |-- u => e : t, q



in OCaml:

```
gen : ctxt -> exp -> ann_exp * scheme * constraints
```

Constraint Generation

Simple rules:

- G |-- x ==> x : s, { } (if G(x) = s)
- G |-- 3 ==> 3 : int, { } (same for other ints)
- G |-- true ==> true : bool, { }
- G |-- false ==> false : bool, { }

Operators

$$\frac{G \vdash u1 \Rightarrow e1 : t1, q1 \qquad G \vdash u2 \Rightarrow e2 : t2, q2}{G \vdash u1 + u2 \Rightarrow e1 + e2 : int, q1 \cup q2 \cup \{t1 = int, t2 = int\}}$$

$$\frac{G \vdash u1 \Rightarrow e1 : t1, q1 \qquad G \vdash u2 \Rightarrow e2 : t2, q2}{G \vdash u1 < u2 \Rightarrow e1 + e2 : bool, q1 \cup q2 \cup \{t1 = int, t2 = int\}}$$

If statements

G |-- u1 ==> e1 : t1, q1
G |-- u2 ==> e2 : t2, q2
G |-- u3 ==> e3 : t3, q3

G |-- if u1 then u2 else u3 ==> if e1 then e2 else e3

: a, q1 U q2 U q3 U {t1 = bool, a = t2, a = t3}

Function Application

```
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}
```

Function Declaration

G, x : a |-- u ==> e : t, q (for fresh a)

G |-- fun x -> e ==> fun (x : a) -> e

: a -> b, q U {t = b}

Function Declaration

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}

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)
- dom(S) = set of variables s.t. S(a) ≠ 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)
- dom(S) = set of variables s.t. S(a) ≠ a

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

apply: [int/a, int->bool/b]

to: b -> a -> b

returns: (int->bool) -> int -> (int->bool)

Substitutions

We can apply a substitution S to a full type scheme:

eg: apply [int/a, int->bool/b] to b -> a -> b

returns: (int->bool) -> int -> (int->bool)

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:

a = b -> c
c = int -> bool

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:

a = b -> c
c = int -> bool

solution:

b -> (int -> bool)/a
int -> bool/c
b/b

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:

a = b -> c
c = int -> bool

solution:

b -> (int -> bool)/a
int -> bool/c
b/b

constraints with solution applied:

b -> (int -> bool) = b -> (int -> bool)
int -> bool = int -> bool

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.

A second solution

constraints:

a = b -> c
c = int -> bool

solution 1:

b -> (int -> bool)/a
int -> bool/c
b/b

solution 2:

int -> (int -> bool)/a
int -> bool/c
int/b

Substitutions

When is one solution better than another to a set of constraints?

constraints:

```
a = b -> c
c = int -> bool
```

solution 1:

```
b -> (int -> bool)/a
int -> bool/c
b/b
```

type b -> c with solution applied:

```
b -> (int -> bool)
```

solution 2:

```
int -> (int -> bool)/a
int -> bool/c
int/b
```

type b -> c with solution applied:

```
int -> (int -> bool)
```

Substitutions

solution 1:

```
b -> (int -> bool)/a
int -> bool/c
b/b
```

solution 2:

```
int -> (int -> bool)/a
int -> bool/c
int/b
```

type b -> c with solution applied:

```
b -> (int -> bool)
```

type b -> c with solution applied:

```
int -> (int -> bool)
```

Solution 1 is "more general" – there is more flex.

Solution 2 is "more concrete"

We prefer solution 1.

Substitutions

solution 1:

```
b -> (int -> bool)/a
int -> bool/c
b/b
```

solution 2:

```
int -> (int -> bool)/a
int -> bool/c
int/b
```

type b -> c with solution applied:

```
b -> (int -> bool)
```

type b -> c with solution applied:

```
int -> (int -> bool)
```

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))

Substitutions

solution 1:

```
b -> (int -> bool)/a
int -> bool/c
b/b
```

solution 2:

```
int -> (int -> bool)/a
int -> bool/c
int/b
```

type b -> c with solution applied:

```
b -> (int -> bool)
```

type b -> c with solution applied:

```
int -> (int -> bool)
```

There is always a *best* solution, which we can a *principle solution*.

The best solution is (at least as) preferred as any other solution.

Most General Solutions

S is the principal (most general) solution of a constraint q if

- S |= q (it is a solution)
- if T |= q then T <= 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 <= S. eg:

- T <= S if T is "more specific"/"less general" than S
- If there is a

- Formally: T <= S if and only if T = U o S for some U

Composition of Substitutions

Composition (U o S) applies the substitution S and then applies the substitution U:

- (U o S)(a) = U(S(a))

We will need to compare substitutions

- T <= S if T is "more specific" than S
- T <= S if T is "less general" than S
- Formally: T <= S if and only if T = U o S for some U

Composition of Substitutions

Examples:

- example 1: any substitution is less general than the identity substitution I:
  - S <= I because S = S o I
- example 2:
  - S(a) = int, S(b) = c -> c
  - T(a) = int, T(b) = int -> int
  - we conclude: T <= S
  - if T(a) = int, T(b) = int -> 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

$$\frac{}{S \models \{\}}$$

any substitution is
a solution for the empty
set of constraints

$$\frac{S(s1) = S(s2) \qquad S \models q}{S \models \{s1 = s2\} \cup q}$$

a solution to an equation
is a substitution that makes
left and right sides equal

Examples

Example 1

- q = {a=int, b=a}
- principal solution S:

Examples

Example 1

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

Examples

Example 2

- q = {a=int, b=a, b=bool}
- principal solution S:

Examples

Example 2

- q = {a=int, b=a, b=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, { })

Unification Machine

We can specify unification as a transition system:

$$(S1, q1) \rightarrow (S2, q2)$$

Base types & simple variables:

$$(S,\{bool=bool\} \cup q) \rightarrow (S, q)$$

$$(S,\{int=int\} \cup q) \rightarrow (S, q)$$

$$(S,\{a=a\} \cup q) \rightarrow (S, q)$$

Unification Machine

Functions:

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

breaks down into smaller constraints

Variable definitions

when a is not in FreeVars(s)

(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]

Occurs Check

Recall this program:

```
fun x -> x x
```

It generates the the constraints: a -> a = a

What is the solution to {a = a -> a}?

Occurs Check

Recall this program:

```
fun x -> x x
```

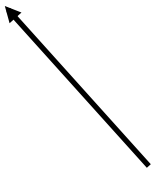
It generates the the constraints: a -> a = a

What is the solution to {a = a -> a}?

There is none!

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

```
   (S1, q1)
-> (S2, q2)
-> (S3, q3)
...
-> (Sn, { })
```

But sometimes we get stuck, with an equation like this

- int = bool
- s1 -> s2 = int
- s1 -> s2 = bool
- a = s (s contains a)
- 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

- (S1,q1) -> (S2,q2) -> ...

Termination

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 (ie: fewer occurrences of int, bool, or "->")
- This is a lexacographic ordering
  - we can prove (by contradiction) that there is no infinite decreasing sequence of constraints

Termination

Lemma: Every step reduces the size of q

- Proof: By cases (ie: induction) on the definition of the reduction relation.

(S,{int=int} U q) -> (S, q)

(S,{bool=bool} U q) -> (S, q)

(S,{a=a} U q) -> (S, q)

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

--- (a not in FV(s))

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

Complete Solutions

A complete solution for (S,q) is a substitution T such that

- T <= S
- T |= q

Properties of Solutions

Lemma 1:

- Every final state (S, { }) has a complete solution.
  - It is S:
    - S <= S
    - S |= { }

Properties of Solutions

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) -> (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!

Summary: Unification

By termination, (I,q) ->* (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 <= S and therefore S is principal.

Summary: Unification (cont.)

... Moreover:

- If q' is not { } (and (I,q) ->* (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

let Generalization

Where do we introduce polymorphic values?

let x = v ==> let x : forall a1,..,an.s = v

if v : s and a1,...,an are the variables of s

And place x : forall a1,...,an.s in the context.

let Generalization

Where do we introduce polymorphic values?

let x = v ==> let x : forall a1,..,an.s = v

if v : s and a1,...,an are the variables of s

And place x : forall a1,...,an.s in the context.

Where and how do we use a polymorphic value?

G |- x ==> x : s[b1/a1,...,bn/an), {}

when G(x) = forall a1,...,an.s and b1,...,bn 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 ==> 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