Type Checking

Part 3: Type Inference (Simple Types)

Speaker: David Walker

COS 326

Princeton University

Robin Milner: Turing Award Winner 1991

Robin Milner

For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

We will be studying Hindley-Milner type inference.
Discovered by Hindley, rediscovered by Milner. Formalized by Damas.
Broken several times when effects were added to ML.

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
```

It’s very convenient we don’t have to annotate f and l with their types, as is required by our type checking algorithm.

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 *principal 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.

E.g.:

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

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

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

Language Design for Type Inference

Principal 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 principal 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

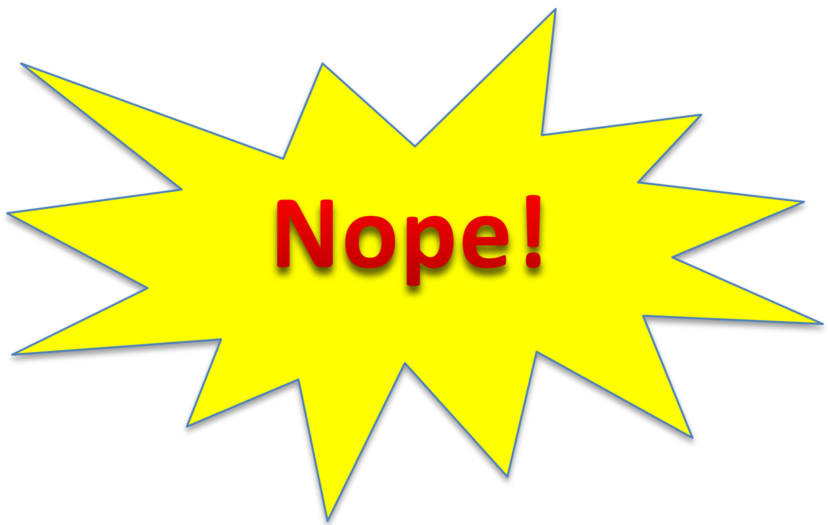
We showed that removing types from the language causes a failure of principal types.

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 has universal types on the outside ("prenex quantification"):

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

It does not have types like this:

```
(forall 'a.'a -> int) -> int -> bool
```

argument type has its own polymorphic quantifier

Language Design for Type Inference

Consider this program:

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

notice that parameter g is used inside f as if:

1. its argument can have type bool, *AND*
2. its argument can have type int

Language Design for Type Inference

Consider this program:

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

notice that parameter g is used inside f as if:

1. its argument can have type bool, *AND*
2. its argument can have type int

Does the following type work?

```
f:  ('a -> int) -> int * int
```

Language Design for Type Inference

Consider this program:

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

notice that parameter g is used inside f as if:

1. its argument can have type bool, *AND*
2. its argument can have type int

Does the following type work?

```
f:  ('a -> int) -> int * int
```

NO: this says g’s argument can be any type ‘a (it could be int or bool)

Consider g is (fun x -> x + 2) : int -> int.

Unfortunately, f g goes wrong when g applied to true inside f.

Language Design for Type Inference

Consider this program again:

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

We might want to give it this type:

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

Notice that the universal quantifier appears left of ->

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 undecidable.

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.

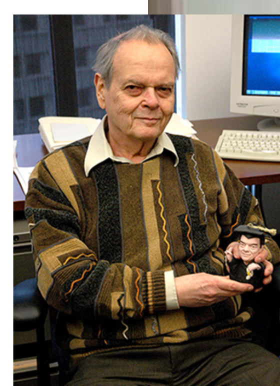
```
let f g = (g true, g 3)
```

```
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 2^{nd}-order arithmetic.

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
```

THE TYPE INFERENCE ALGORITHM

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
```

Two Algorithms for Inferring Types

Algorithm 1:

- Declarative; generates constraints to be solved later
- Easier to understand
- Easier to prove correct
- Less efficient, not used in practice

Algorithm 2:

- Imperative; solves constraints and updates as-you-go
- Harder to understand
- Harder to prove correct
- More efficient, used in practice
- See: http://okmij.org/ftp/ML/generalization.html

Algorithm 1

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 variables or finding that the equations can't be solved

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 -> e
...
```

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
```

CONSTRAINT GENERATION

Type Inference Details

Type constraints are sets of equations between type schemes

- q ::= {s11 = s12, ..., sn1 = sn2}

- e.g.: {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:

- $G \vdash u \Rightarrow e : t, q$

constraints that must be solved

type (scheme)

context

annotated expression

unannotated expression

inputs

outputs

in OCaml:

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

Constraint Generation

Simple rules:

G ⊢ x ==> x : s, { }   (if G(x) = s)

G ⊢ n ==> n : int, { }

G ⊢ true ==> true : 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 ⊢ 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

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

Function Application

```
G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2            (for 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 -> u ==> fun (x : a) -> e    :     a -> t,      q
```

```
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}
```

Summary: The Type Inference System

```
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 ⊢ x ==> x : s,  { }      (if G(x) = s)
```

```
G ⊢ n ==> n : int, { }
```

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

```
G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2          (for fresh a)
---------------------------------------------------------------
G ⊢ u1 u2==> e1 e2      :      a,     q1 U q2 U {t1 = t2 -> a}
```

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

```
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}
```

SOLUTIONS TO CONSTRAINTS

Solutions

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

- a function from type variables to type schemes
- 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) $\neq$ a

Solutions

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

- a function from type variables to type schemes
- 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) $\neq$ a

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

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

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

Solutions

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.

solution:

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

constraints:

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

constraints with solution applied:

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

Solutions

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
```

Solutions

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)
```

Solutions

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.

Solutions

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)

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))

Solutions

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)

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

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

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:
  - does not exist (there is no solution to q)

UNIFICATION

Unification

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

- Unification systematically simplifies a set of constraints:
  - Starting state of unification process: (I,q)
  - Final state of unification process: (S, { })

Unification

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

- Unification systematically simplifies a set of constraints:
  - Starting state of unification process: (I,q)
  - Final state of unification process: (S, { })

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

Unification

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

- Unification systematically simplifies a set of constraints:
  - Starting state of unification process: (I,q)
  - Final state of unification process: (S, { })

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

```
unify_step (S, {bool=bool} U q)   =   (S, q)

unify_step (S, {int=int}      U q)   =   (S, 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:
  - Starting state of unification process: (I,q)
  - Final state of unification process: (S, { })

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

```
unify_step (S, {bool=bool} U q)   =   (S, q)

unify_step (S, {int=int}      U q)   =   (S, q)
```

```
unify_step (S, {a=a}          U q)   =   (S, 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:
  - Starting state of unification process: (I,q)
  - Final state of unification process: (S, { })

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

```
unify_step (S,   {A -> B   =   C -> D}   U   q)

  = (S,  {A = C,   B = D}   U   q)
```

Unification

extend substitution S with additional subsitution of s for a

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

when a is not in FreeVars(s)

"when a is not in FreeVars(s)" is known as the "occurs check"

Occurs Check

Recall this program:

a -> b

a

```
fun (x:a) -> x x
```

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

Notice that a appears in FreeVars(s)!

There is no solution to these constraints!

Summary: Unification

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

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

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

(S,   {A->B = C->D} U q)   -->   (S, {A = C} U {B = D} U q)

(S, {a=s} U q) --> ([s/a] o S,   [s/a]q)   when a is not in FreeVars(s)

Irreducible States

Recall: unification simplifies equations step-by-step until

- there are no equations left to simplify:

$(S, \{\ \})$

no constraints left.
S is the final solution!

Irreducible States

Recall: unification simplifies equations step-by-step until

- there are no equations left to simplify:

(S, { })

no constraints left.
S is the final solution!

- or we find basic equations are inconsistent:
  - int = bool
  - s1->s2 = int
  - s1->s2 = bool
  - a = s (s contains a)

(or is symmetric to one of the above)

In the latter case, the program does not type check.

TYPE INFERENCE:
THINGS TO REMEMBER

Type Inference: Things to remember

Declarative algorithm: Given a context G, and untyped term u:

- Find e, t, q such that G ⊢ u ==> e : t, q
  - understand the constraints that need to be generated
- Find substitution S that acts as a solution to q via unification
  - if no solution exists, the expression does not type check
- 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