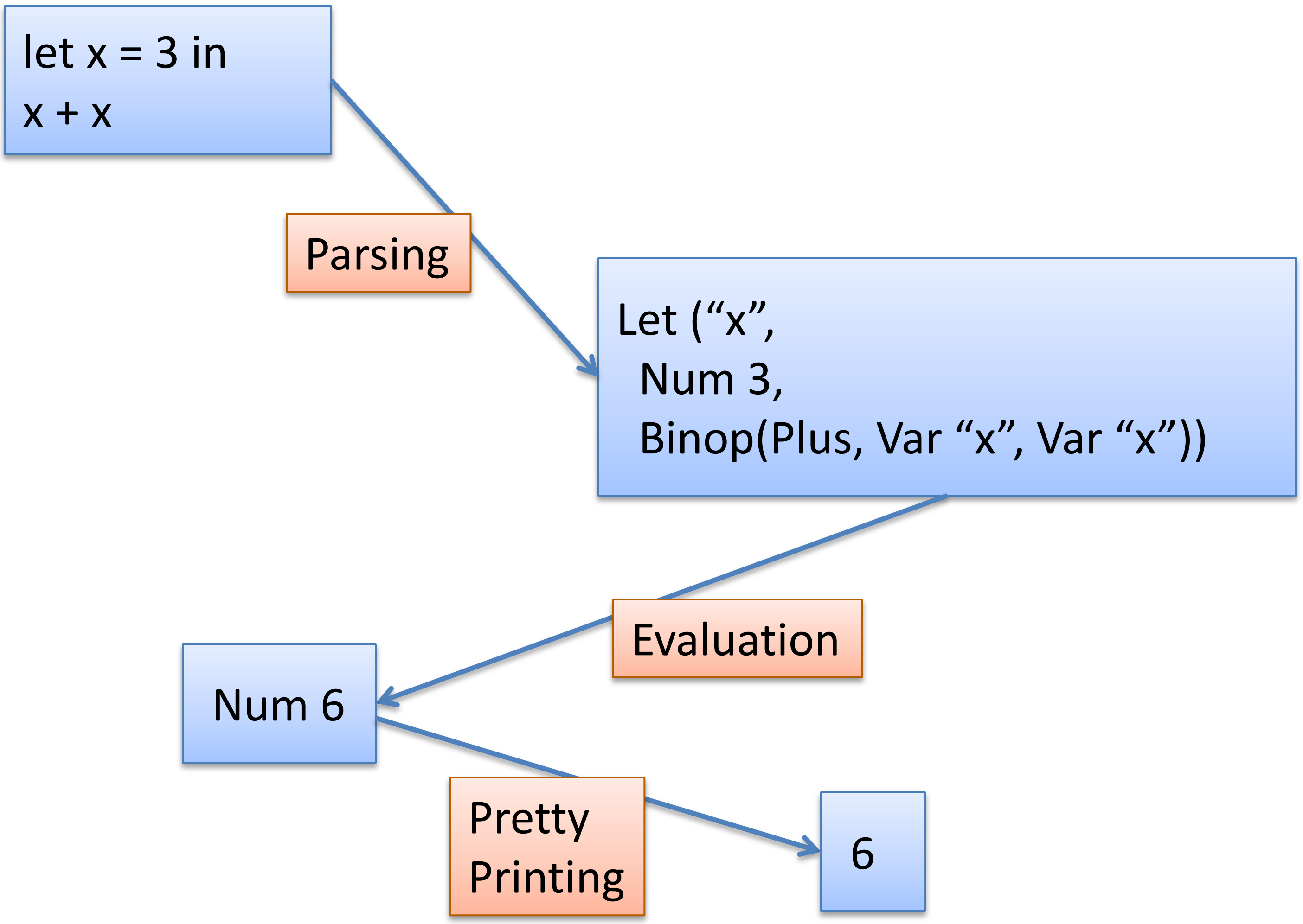
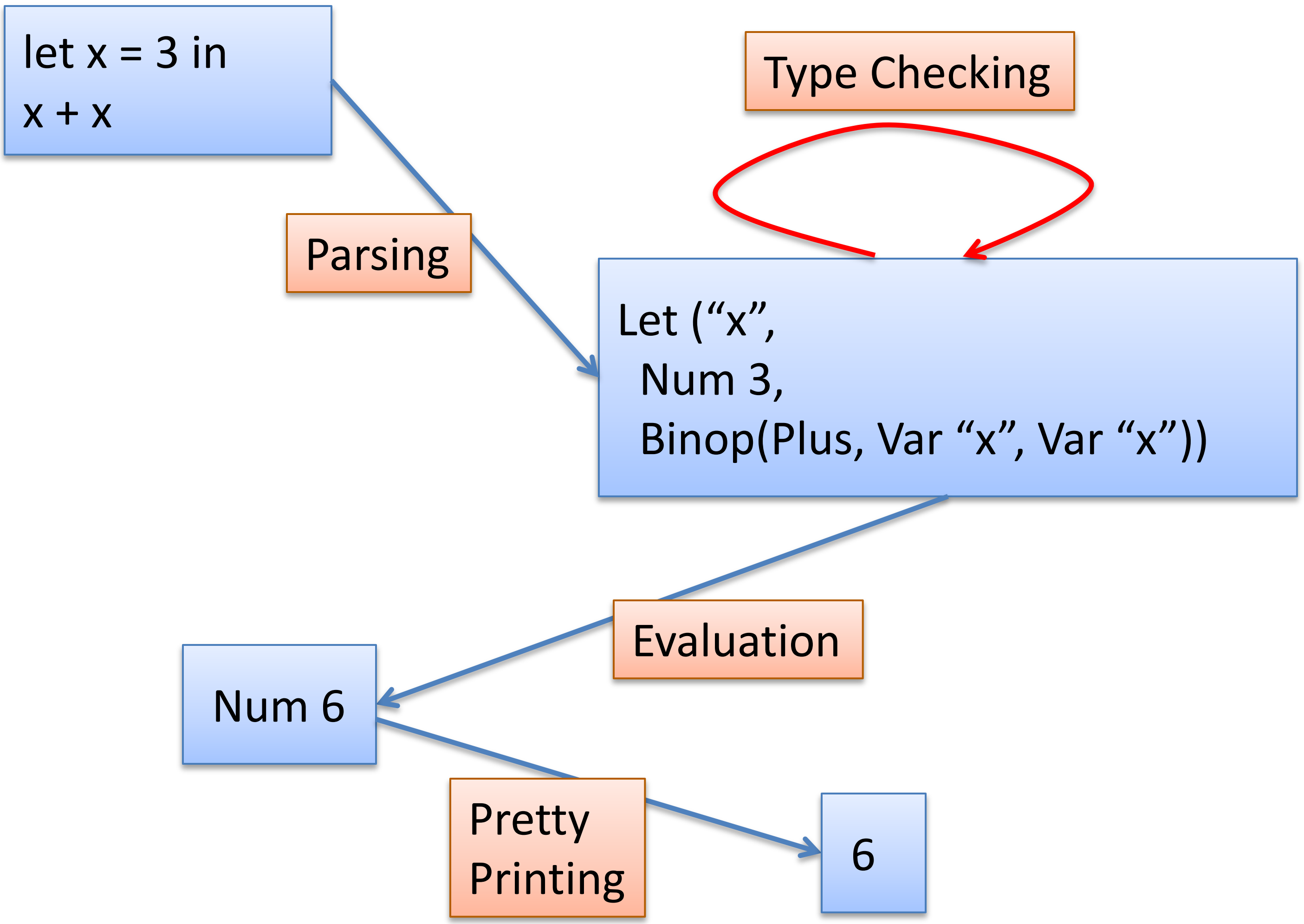
Type Checking

COS 326
David Walker
Princeton University

Implementing an Interpreter



Implementing an Interpreter



Language Syntax

```
type t = IntT | BoolT | ArrT of t * t



type x = string   (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
```

Language Syntax

```
type t = IntT | BoolT | ArrT of t * t




type x = string   (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
```

Notice that we require a type annotation here.

We'll see why this is required for our type checking algorithm later.

Language Syntax (BNF Definition)

```
type t = IntT | BoolT | ArrT of t * t




type x = string   (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
```

```
t ::= int | bool | t -> t

b      -- ranges over booleans
n      -- ranges over integers

x      -- ranges over variable names
c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Recall Inference Rule Notation

When defining how evaluation worked, we used this notation:

$$\frac{e1 \;\text{-->}^*\; \lambda x.e \qquad e2 \;\text{-->}^*\; v2 \qquad e[v2/x] \;\text{-->}^*\; v}{e1\; e2 \;\text{-->}^*\; v}$$

In English:

> "if e1 evaluates to a function with argument x and body e
> and e2 evaluates to a value v2
> and e with v2 substituted for x evaluates to v
> then e1 applied to e2 evaluates to v"

And we were also able to translate each rule into 1 case of a function in OCaml. Together all the rules formed the basis for an interpreter for the language.

The evaluation judgement

This notation:

e -->* v

was read in English as "e evaluates to v."

It described a relation between two things – an expression e and a value v. (And e was related to v whenever e evaluated to v.)

Note also that we usually thought of e on the left as "given" and the v on the right as computed from e (according to the rules).

The typing judgement

This notation:

$$G \mid\!- e : t$$

is read in English as "e has type t in context G." It is going to define how type checking works.

It describes a relation between three things – a type checking context G, an expression e, and a type t.

We are going to think of G and e as given, and we are going to compute t. The typing rules are going to tell us how.

Typing Contexts

What is the type checking context G?

Technically, I'm going to treat G as if it were a (partial) function that maps variable names to types. Notation:

G(x) -- look up x's type in G

G,x:t -- extend G so that x maps to t

When G is empty, I'm just going to omit it. So I'll sometimes just write: |- e : t

Example Typing Contexts

Here's an example context:

x:int, y:bool, z:int

Think of a context as a series of "assumptions" or "hypotheses"

Read it as the assumption that "x has type int, y has type bool and z has type int"

In the subsitution model, if you assumed x has type int, that means that when you run the code, you had better actually wind up substituting an integer for x.

Typing Contexts and Free Variables

One more bit of intuition:

If an expression e contains free variables x, y, and z then we need to supply a context G that contains types for at least x, y and z. If we don't, we won't be able to type check e.

Type Checking Rules

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Goal: Give rules that define the relation "G |- e : t".

To do that, we are going to give one rule for every sort of expression.

(We can turn each rule into a case of a recursive function that implements it pretty directly.)

Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b

o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for constant booleans:

$$\frac{}{G \mid\!- \ b : bool}$$

English:

"boolean constants b *always* have type bool, no matter what the context G is"

Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for constant integers:

$$\frac{}{G \mid\!- n : int}$$

English:

"integer constants n *always* have type int, no matter what the context G is"

Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b

o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for operators:

$$\frac{G \mid\!- e1 : t1 \qquad G \mid\!- e2 : t2 \qquad optype(o) = (t1, t2, t3)}{G \mid\!- e1\ o\ e2 : t3}$$

where

optype (+) = (int, int, int)
optype (-) = (int, int, int)
optype (<) = (int, int, bool)

English:

"e1 o e2 has type t3, if e1 has type t1, e2 has type t2 and o is an operator that takes arguments of type t1 and t2 and returns a value of type t3"

Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b

o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for variables:

look up x in context G

$$\frac{}{G \mid\!- x : G(x)}$$

English:

“variable x has the type given by the context"

Note: this is rule explains (part) of why the context needs to provide types for all of the free variables in an expression

Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b

o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for if:

$$\frac{G \mid\!- \; e1 : bool \qquad G \mid\!- \; e2 : t \qquad G \mid\!- \; e3 : t}{G \mid\!- \; \text{if } e1 \text{ then } e2 \text{ else } e3 : t}$$

English:

"if e1 has type bool
and e2 has type t
and e3 has (the same) type t
then e1 then e2 else e3 has type t "

Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for functions:

$$\frac{G, x{:}t \mid\!- \; e : t2}{G \mid\!- \; \lambda x{:}t.e : t \rightarrow t2}$$

Notice that to know how to extend the context G, we need the typing annotation on the function argument

English:

"if G extended with x:t proves e has type t2 then λx:t.e has type t -> t2 "

Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for function call:

$$\frac{G \mid\!- e1 : t1 \rightarrow t2 \qquad G \mid\!- e2 : t1}{G \mid\!- e1\ e2 : t2}$$

English:

"if G extended with x:t proves e has type t2 then λx:t.e has type t -> t2 "

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for let:

$$\frac{G \ |\!- \ e1 : t1 \qquad G,x{:}t1 \ |\!- \ e2 : t2}{G \ |\!- \ \text{let } x = e1 \text{ in } e2 : t2}$$

English:

"if e1 has type t1
and G extended with x:t1 proves e2 has type t2
then let x = e1 in e2 has type t2 "

A Typing Derivation

A typing derivation is a "proof" that an expression is well-typed in a particular context.

Such proofs consist of a tree of valid rules, with no obligations left unfulfilled at the top of the tree. (ie: no axioms left over).

notice that "int" is associated with x in the context

G, x:int |- x : int   G,x:int |- 2 : int

G, x:int |- x + 2 : int

G |- λx:int. x + 2 : int -> int

Key Properties

Good type systems are *sound*.

- ie, well-typed programs have "well-defined" evaluation
  - ie, our interpreter should not raise an exception part-way through because it doesn't know how to continue evaluation
  - colloquial phrase: "sound type systems do not go wrong"

Examples of OCaml expressions that go wrong:

- true + 3 (addition of booleans not defined)
- let (x,y) = 17 in ... (can't extract fields of int)
- true (17) (can't use a bool as if it is a function)

Sound type systems *accurately* predict run time behavior

- if e : int and e terminates then e evaluates to an integer

Soundness = Progress + Preservation

Proving soundness boils down to two theorems:

Progress Theorem:

If |- e : t then either:

(1) e is a value, or

(2) e --> e'

Preservation Theorem:

If |- e : t and e --> e' then |- e' : t

See COS 510 for proofs of these theorems.

But you have most of the necessary techniques:

Proof by induction on the structure of ...

... various inductive data types. :-)

The typing rules also define an algorithm for
... type checking ...

If you view G and e as inputs,
the rules for “G |- e : t” tell you how to compute t

(see demo code)

TYPE INFERENCE

Robin Milner

Robin Milner
Turing Award, 1991

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.

Robin Milner

Robin Milner
Turing Award, 1991

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 *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

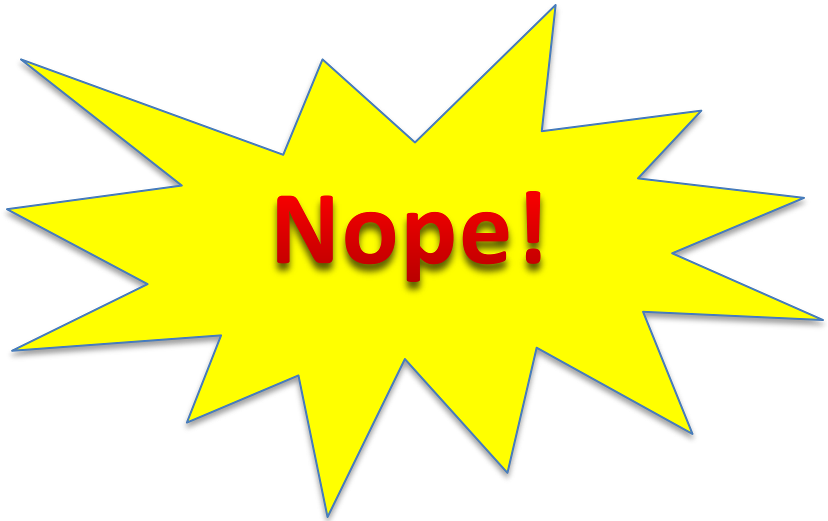
We showed that removing types from the language causes a failure of principle types.

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 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. it's argument can have type bool, *AND*
2. it's 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. it's argument can have type bool, *AND*
2. it's argument can have type int

Does the following type work?

```
('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. it's argument can have type bool, *AND*
2. it's argument can have type int

Does the following type work?

```
('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 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.

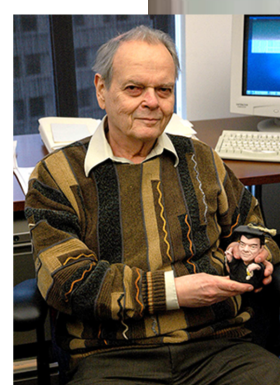
```
let f g = (g true, g 3)
```

```
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

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

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
```

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 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
```

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

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

context

unannotated expression

annotated expression

type (scheme)

constraints that must be solved

inputs

outputs

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

context — unannotated expression (inputs)

annotated expression — type (scheme) — constraints that must be solved (outputs)

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, { }

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) $\neq$ 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)
- dom(S) = set of variables s.t. S(a) $\neq$ 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

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:

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
```

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
```

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)
```

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.

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

Unification simplifies equations step-by-step until

- there are no equations left to simplify, or
- we find basic equations are inconsistent and we fail

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

Unification

Unification simplifies equations step-by-step until

- there are no equations left to simplify, or
- we find basic equations are inconsistent and we fail

```
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 simplifies equations step-by-step until

- there are no equations left to simplify, or
- we find basic equations are inconsistent and we fail

```
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 simplifies equations step-by-step until

- there are no equations left to simplify, or
- we find basic equations are inconsistent and we fail

```
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

Unification simplifies equations step-by-step until

- there are no equations left to simplify, or
- we find basic equations are inconsistent and we fail

```
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

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

when a is not in FreeVars(s)

Unification

the substitution S' defined to:
do S then substitute s for a

the constraints q' defined to:
be like q except s replacing a

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

when a is not in FreeVars(s)

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!

Notice that a does appear in FreeVars(s)

Whenever a appears in FreeVars(s) and s is not just a, there is no solution to the system of constraints.

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!

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

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 MORE DETAILS

Generalization

Where do we introduce polymorphic values? Consider:

```
g (fun x -> 3)
```

It is tempting to do something like this:

```
(fun x -> 3) : forall a. a -> int
```

```
g : (forall a. a -> int) -> int
```

But recall the beginning of the lecture:

if we aren't careful, we run into decidability issues

Generalization

Where do we introduce polymorphic values?

In ML languages: Only when values bound in ”let declarations”

```
g (fun x -> 3)
```

No polymorphism for fun x -> 3!

```
let f : forall a. a -> a = fun x -> 3 in
g f
```

Yes polymorphism for f!

Let Polymorphism

Where do we introduce polymorphic values?

```
let x = v
```

Rule:

- if v is a value (or guaranteed to evaluate to a value without effects)
  - OCaml has some rules for this
- and v has type scheme s
- and s has free variables a, b, c, ...
- and a, b, c, ... do not appear in the types of other values in the context
- then x can have type forall a, b, c. s

Let Polymorphism

Where do we introduce polymorphic values?

```
let x = v
```

Rule:

- if v is a value (or guaranteed to evaluate to a value without effects)
  - OCaml has some rules for this
- and v has type scheme s
- and s has free variables a, b, c, ...
- and a, b, c, ... do not appear in the types of other values in the context
- then x can have type forall a, b, c. s

That's a hell of a lot more complicated than you thought, eh?

Unsound Generalization Example

Consider this function f – a fancy identity function:

```
let f = fun x -> let y = x in y
```

A sensible type for f would be:

```
f : forall a. a -> a
```

Unsound Generalization Example

Consider this function f – a fancy identity function:

```
let f = fun x -> let y = x in y
```

A bad (unsound) type for f would be:

```
f : forall a, b. a -> b
```

Unsound Generalization Example

Consider this function f – a fancy identity function:

```
let f = fun x -> let y = x in y
```

A bad (unsound) type for f would be:

```
f : forall a, b. a -> b
```

```
(f true) + 7
```

goes wrong! but if f can have the bad type,
it all type checks. This *counterexample* to soundness shows
that f can't possible be given the bad type safely

Unsound Generalization Example

Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

x : a

Unsound Generalization Example

Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

x : a

suppose we generalize and allow y : forall a.a

Unsound Generalization Example

Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

then we can use y as if it has any type, such as y : b

x : a

suppose we generalize and allow y : forall a.a

Unsound Generalization Example

Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

then we can use y as if it has any type, such as y : b

x : a

suppose we generalize and allow y : forall a.a

but now we have inferred that (fun x -> ...) : a -> b
and if we generalize again,
f : forall a,b. a -> b

That’s the bad type!

Unsound Generalization Example

Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

x : a

suppose we generalize and allow y : forall a.a

this was the bad step – y can't really have
any type at all. It's type has got to be the same
as whatever the argument x is.

x was in the context when we tried to generalize y!

The Value Restriction

```
let x = v
```

this has got to be a value
to enable polymorphic
generalization

Unsound Generalization Again

not a value!

```
let x = ref [] in
```

x : forall a . a list ref

Unsound Generalization Again

not a value!

```
let x = ref [] in

x := [true];
```

x : forall a . a list ref

use x at type bool as if x : bool list ref

Unsound Generalization Again

```
let x = ref [] in

x := [true];

List.hd (!x) + 3
```

x : forall a . a list ref

use x at type bool as if x : bool list ref

use x at type int as if x : int list ref

and we crash ....

What does OCaml do?

```
let x = ref [] in
```

x : '_weak1 list ref

a "weak" type variable can't be generalized

means "I don't know what type this is but it can only be *one* particular type"

look for the "_" to begin a type variable name

What does OCaml do?

```
let x = ref [] in

x := [true];
```

x : '_weak1 list ref

x : bool list ref

the "weak" type variable
is now fixed as a bool
and can't be anything else

bool was substituted for
'_weak during type
inference

What does OCaml do?

```
let x = ref [] in

x := [true];

List.hd (!x) + 3
```

x : '_weak1 list ref

x : bool list ref

Error: This expression has type bool
but an expression was expected
of type int

type error ...

One other example

notice that the RHS is now a value
– it happens to be a function value
but any sort of value will do

now generalization
is allowed

```
let x = fun () -> ref [] in
```

x : forall ’a. unit -> ‘a list ref

One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

now generalization is allowed

```
let x = fun () -> ref [] in

x () := [true];
```

x : forall 'a. unit -> 'a list ref

x () : bool list ref

One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

now generalization is allowed

```
let x = fun () -> ref [] in

x () := [true];

List.hd (!x ()) + 3
```

x : forall ’a. unit -> ‘a list ref

x () : bool list ref

x () : int list ref

what is the result of this program?

One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

now generalization is allowed

```
let x = fun () -> ref [] in

x () := [true];

List.hd (!x ()) + 3
```

x : forall 'a. unit -> 'a list ref

x () : bool list ref

x () : int list ref

what is the result of this program?

List.hd raises an exception because it is applied to the empty list. why?

One other example

notice that the RHS is now a value
– it happens to be a function value
but any sort of value will do

creates a new, different reference
every time it is called

```
let x = fun () -> ref [] in

x () := [true];

List.hd (!x ()) + 3
```

creates one reference

creates a second totally
different reference

what is the result of this program?

List.hd raises an exception because it is applied to the empty list. why?

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, 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

Type Inference: Things to remember

In order to introduce polymorphic quantifiers, remember:

- Quantifiers must be on the outside only
  - this is called “prenex” quantification
  - otherwise, type inference may become undecidable
- Quantifiers can only be introduced at let bindings:
  - let x = v
  - only the type variables that do not appear in the environment may be generalized
- The expression on the right-hand side must be a value
  - no references or exceptions