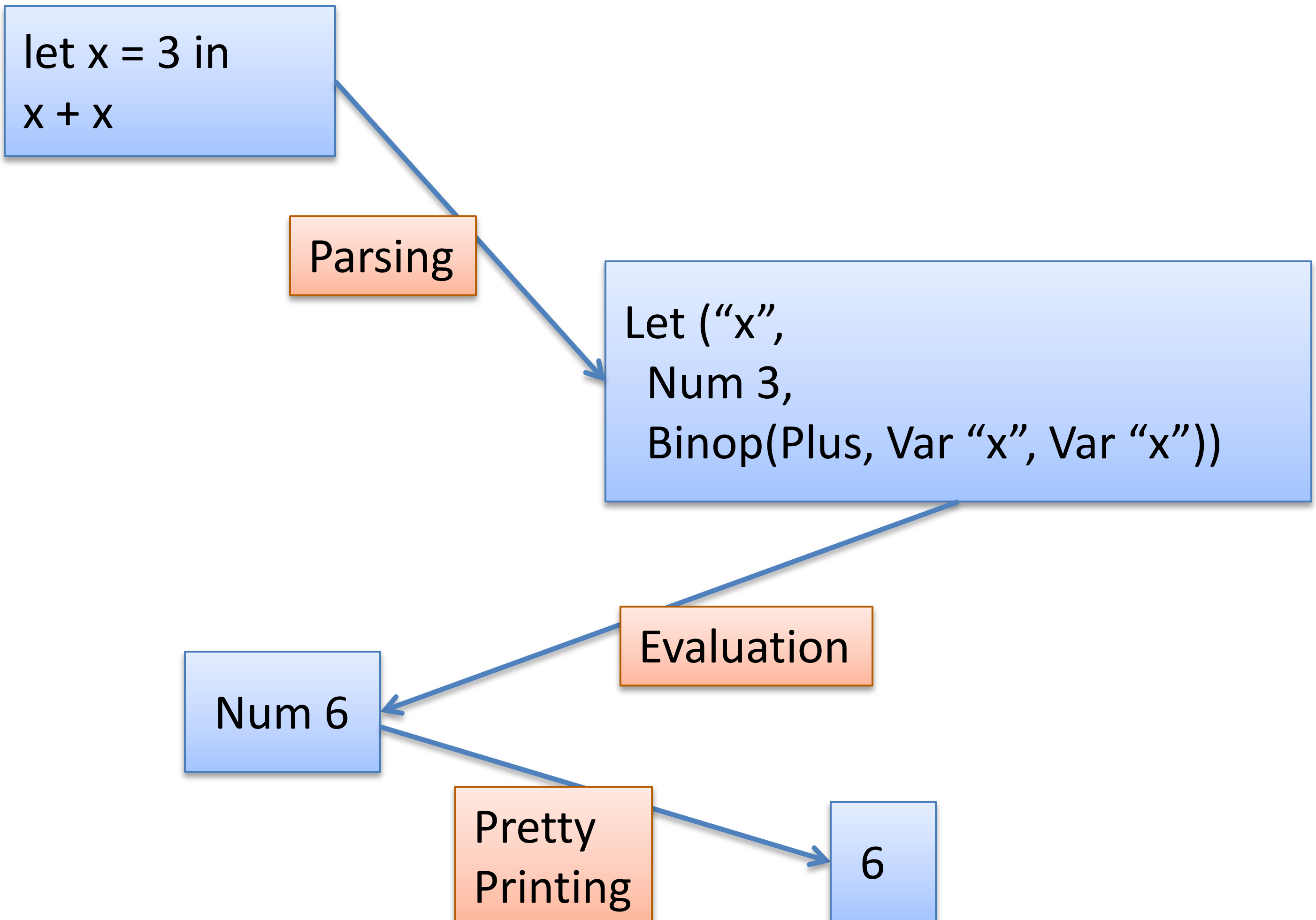
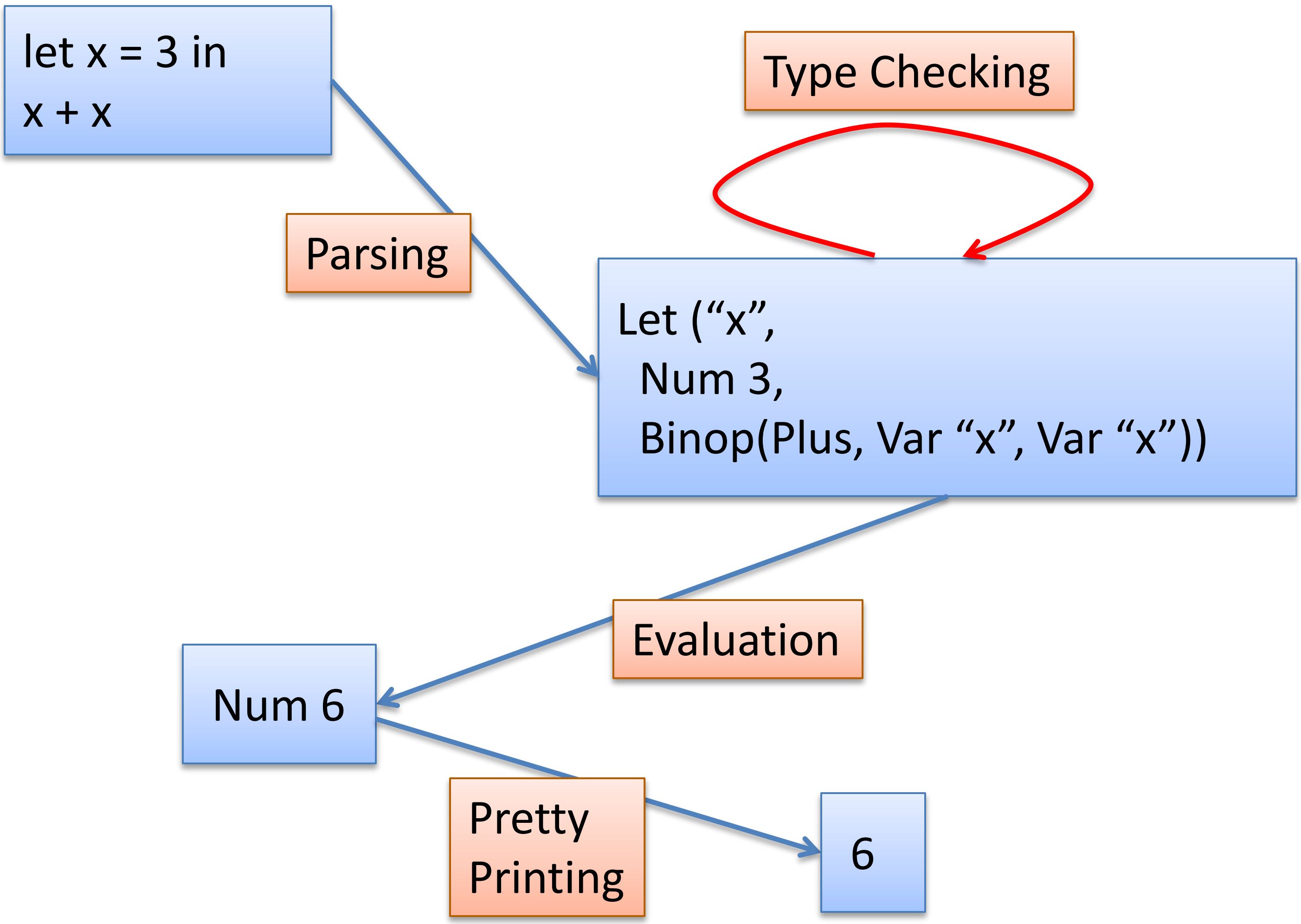
Type Checking

COS 326
Andrew Appel
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 (Abstract) 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:

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

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:

$$G \vdash e : t$$

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: $\vdash e : t$

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 substitution 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:

$$\frac{}{G \vdash b : \text{bool}}$$

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:

$$\frac{}{G \vdash n : int}$$

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:

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

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

$$\frac{}{G \vdash x : G(x)}$$

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:

$$\frac{G \vdash e1 : bool \qquad G \vdash e2 : t \qquad G \vdash e3 : t}{G \vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : t}$$

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:

$$\frac{G, x{:}t \vdash e : t2}{G \vdash \lambda x{:}t.e : t \to t2}$$

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:

$$\frac{G \vdash e1 : t1 \to t2 \qquad G \vdash e2 : t1}{G \vdash e1\ e2 : t2}$$

English:

"if e1 has type t1->t2 and e2 has type t1 then e1 e2 has type 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:

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

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.

notice that “int” is associated with x in the context

$$\dfrac{\dfrac{\dfrac{}{G,\ x{:}\mathrm{int} \vdash x : \mathrm{int}} \qquad \dfrac{}{G{,}x{:}\mathrm{int} \vdash 2 : \mathrm{int}}}{G,\ x{:}\mathrm{int} \vdash x + 2 : \mathrm{int}}}{G \vdash \lambda x{:}\mathrm{int}.\ x + 2 : \mathrm{int} \text{ -> } \mathrm{int}}$$

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

Recall the OCaml Definition of Our Syntax

```
type t = IntT                                (* type int *)
       | BoolT                               (* type bool *)
       | ArrT of t * t                       (* type t -> t *)

type x = string                              (* variables *)
type c = Int of int | Bool of bool           (* integer and boolean constants *)
type o = Plus | Minus | LessThan             (* operators *)

type e =                                     (* expressions *)
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * t * e                         (* t gives type of argument *)
  | Call of e * e
  | Let of x * e * e
```

Signature for Context Operations

```
(* abstract type of contexts *)
type ctx

(* empty context *)
val empty : ctx

(* update ctx x t:  updates context ctx by binding variable x to type t *)
val update : ctx -> x -> t -> ctx

(* look ctx x: retrieves the type t associated with x in ctx
 *             raises NotFound if x does not appear in ctx *)
exception NotFound
val look : ctx -> x -> t
```

Auxiliary Functions

```
(* const c is the type of constant c *)
let const (c : c) : t =
  match c with
  | Int i -> IntT
  | Bool b -> BoolT

(* op o = (t1, t2, t3) when o has type t1 -> t2 -> t3 *)
let op (o : o) : t  =
  match o with
  | Plus -> (IntT, IntT, IntT)
  | ...

(* use err s to signal a type error with message s *)
exception TypeError of string
let err s =  raise (TypeError s)
```

Simple Rules

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Const c -> const c

  | Op (e1, o, e2) ->
      let (t1, t2, t) = op o in     (* op : t1 -> t2 -> t *)
      let t1' = check ctx e1 in
      let t2' = check ctx e2 in
      if (t1 = t1') && (t2 = t2') then
        t
      else
        err "bad argument to operator"
```

$$\frac{\text{const}(c) = t}{G \vdash c : t}$$

$$\frac{\text{optype}(o) = (t1, t2, t3) \quad G \vdash e1 : t1 \quad G \vdash e2 : t2}{G \vdash e1\ o\ e2 : t3}$$

Simple Rules

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Var x ->
      begin
        try look ctx x with
          NotFound -> err ("free variable: " ^ x)
      end
```

$$\frac{}{G \vdash x : G(x)}$$

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Fun (x,t,e) ->
      check (update ctx x t) e
```

$$\frac{G, x{:}t \vdash e : t2}{G \vdash \lambda x{:}t.e : t \text{ -> } t2}$$

Notice that if we did not have the type t as a typing annotation we would not be able to make progress in our type checker at this point. We need to have a type for the variable x in our context in order to recursively check the expression e

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Call (e1, e2) ->
      begin
        let t1 = check ctx e1 in
        match t1 with
         | ArrT (targ, tresult) ->
             let t2 = check ctx e2 in
             if targ = t2 then tresult
             else  err "bad argument to function"
        | _,_ -> err "not a function in call position"
      end
```

$$\frac{G \vdash e1 : targ \to tresult \qquad G \vdash e2 : targ}{G \vdash e1\ e2 : tresult}$$

Exercise: Other Rules

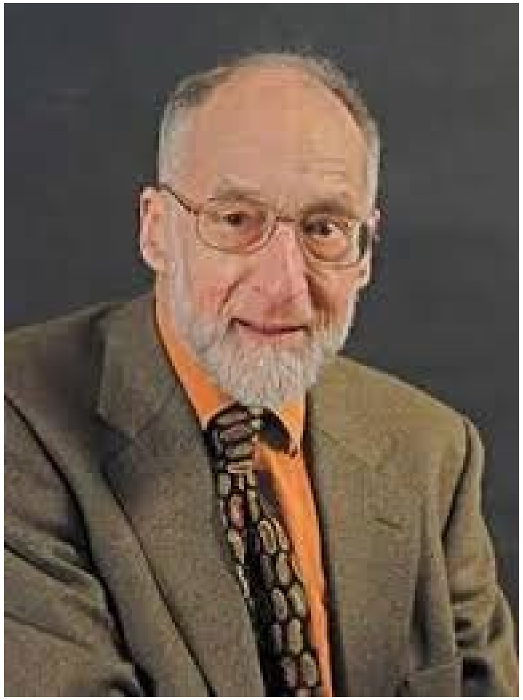
```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | If (e1, e2, e3) -> ...

  | Let (x, e1, e2) -> ...
```

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.

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

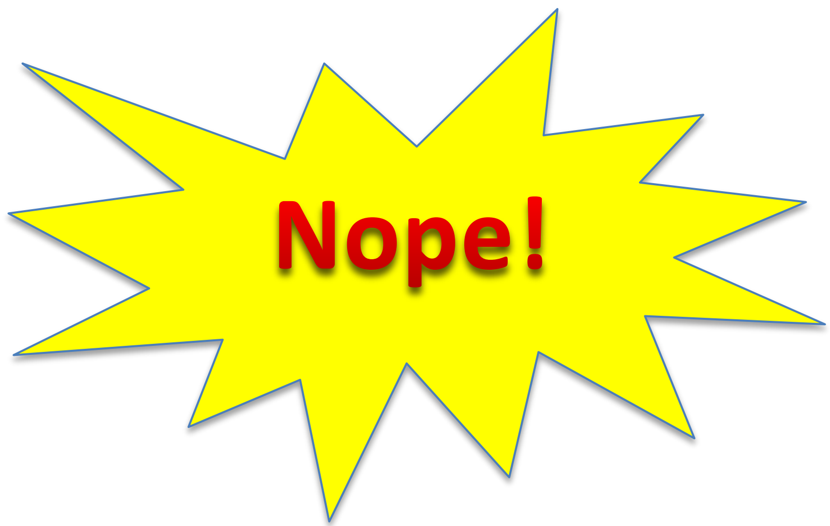
We showed that removing types from the language causes a failure of principal types.

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?

```
('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?

```
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 2nd-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 affect 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 affect 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
```