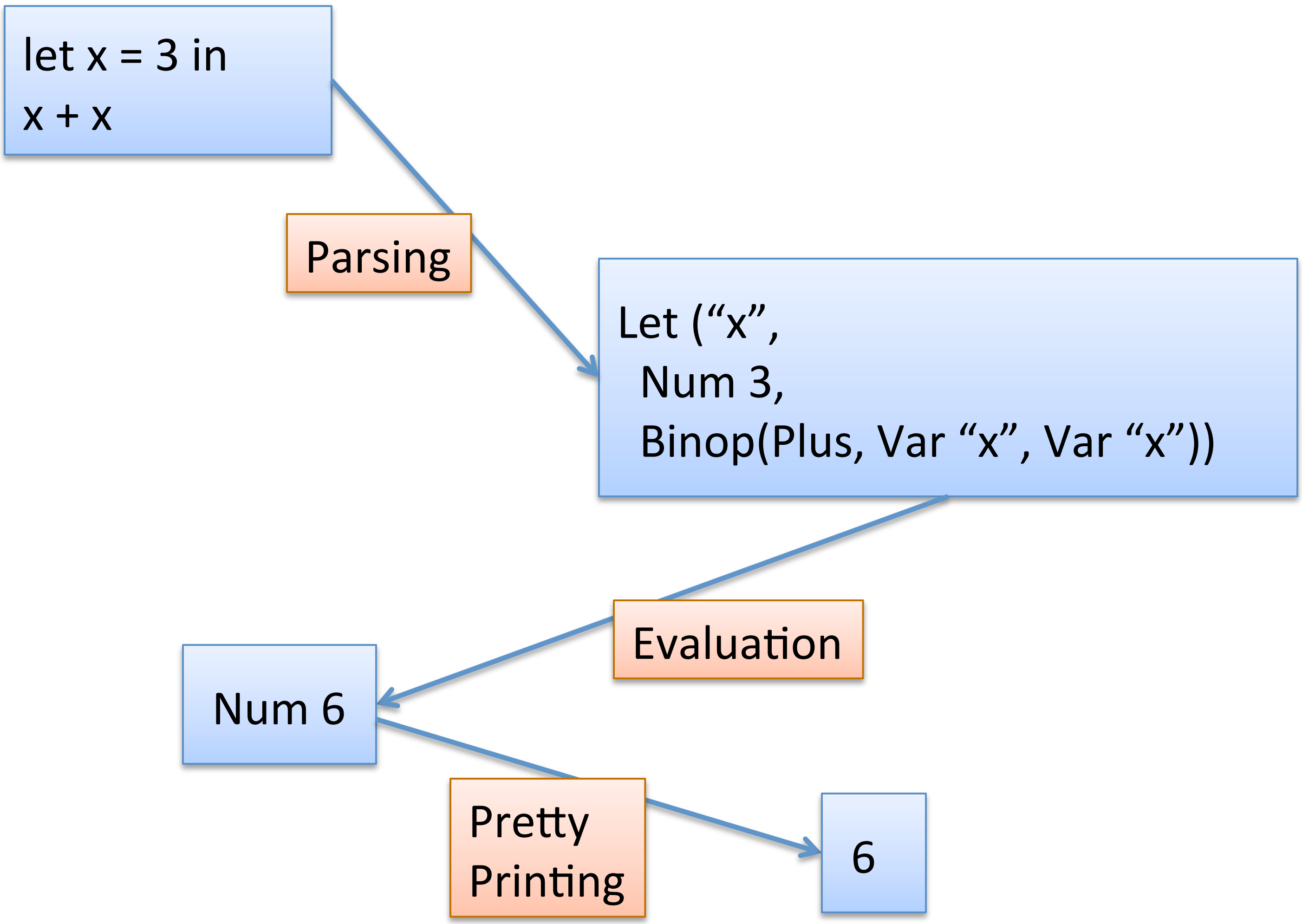
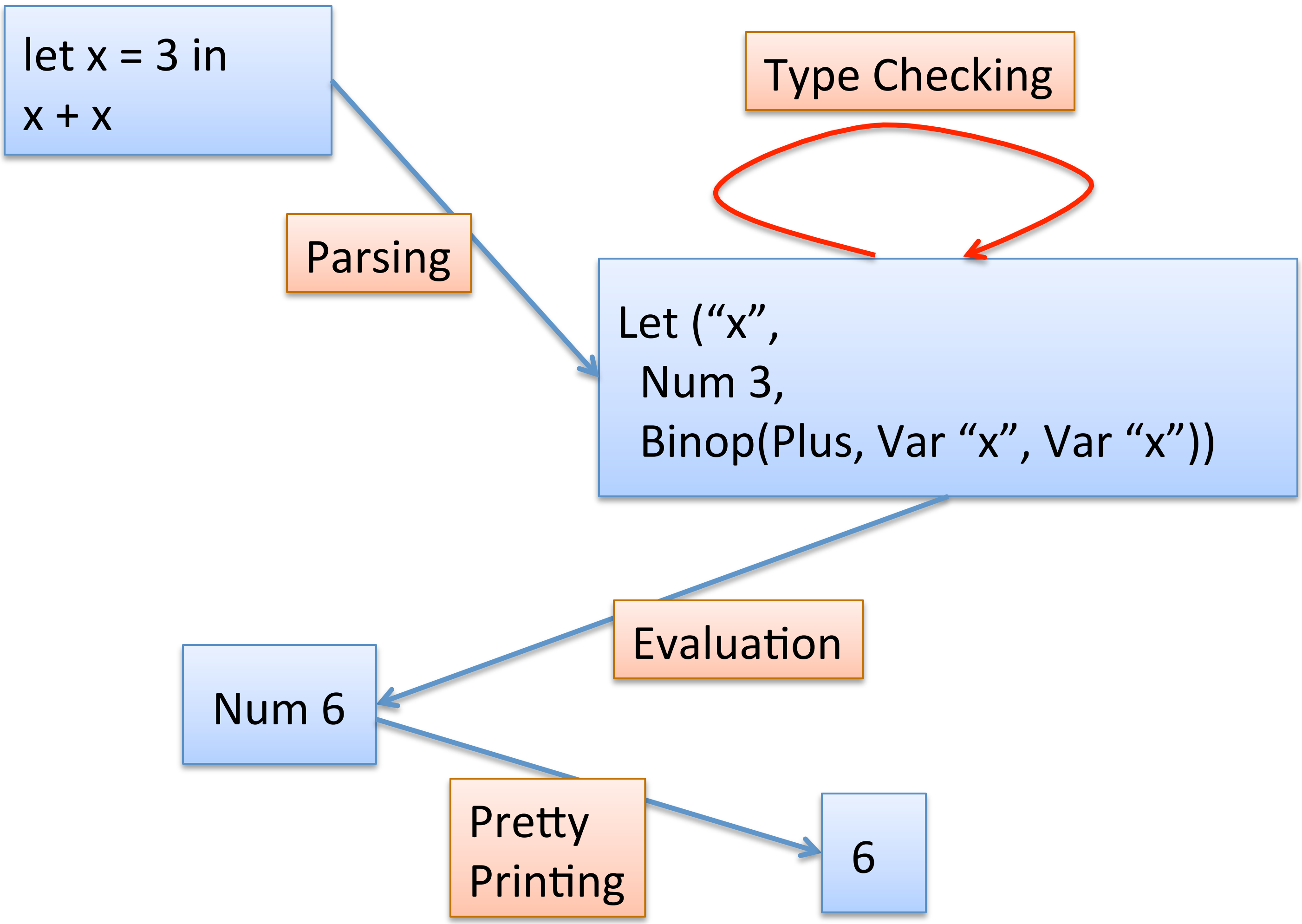
Type Checking

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:

$$\frac{e1 \dashrightarrow \lambda x.e \qquad e2 \dashrightarrow v2 \qquad e[v2/x] \dashrightarrow v}{e1\ e2 \dashrightarrow v}$$

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 |- 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 an "assumption" or "hypothesis"

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 takes an expression as an input and implement rules 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 constant integers:

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

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:

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

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:

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

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 function call:

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

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:

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

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).

$$
\dfrac{\dfrac{\dfrac{\text{G,x:int(x) = int}}{\text{G, x:int |- x : int}} \qquad \dfrac{}{\text{G,x:int |- 2 : int}}}{\text{G, x:int |- x + 2 : int}}}{\text{G |- }\lambda\text{x:int. x + 2 : int -> int}}
$$

Key Properties

Good type systems are *sound*.

In other words, if the type system says that e has type t then e should have "well-defined" evaluation (ie, our interpreter should not raise an exception part-way through because it doesn't know how to continue evaluation).

Also, if e has type t and it terminates and produces a value, then it should produce a value of that type. eg, if t is int, then it should produce a value with type int.

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 ...