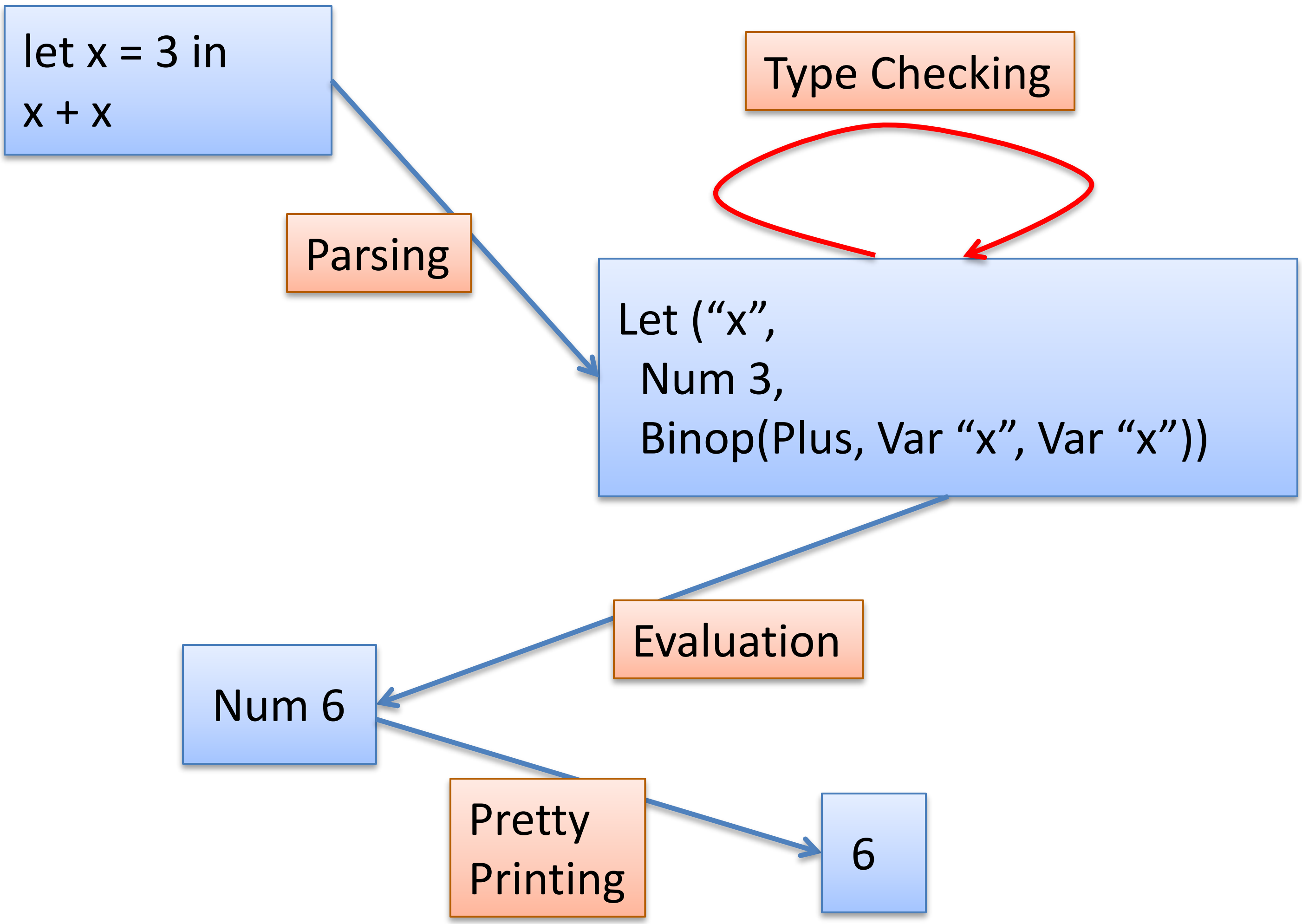
Type Checking
Part 1: Formal Rules

Speaker: David Walker
COS 326
Princeton University

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 * t * 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 * t * 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 * t * 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:

$$G \vdash e : t$$

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: ⊢ 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.)

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

Rule for constant booleans:

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

English:

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

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

Rule for constant integers:

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

English:

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

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
```

For any constant (where c is an int or a bool) we might use the following rule if we have a function around like "const" to tell us the type of the constant.

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

English:

"const c *always* has the type t that the function const says it does, no matter what the context G is"

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

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"

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

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

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

Rule for if:

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

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 "

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

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 "

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

Rule for function call:

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

English:

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

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
```

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. (ie: no axioms left over).

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

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

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: "well-typed programs 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 $\vdash e : t$ then either:

(1) e is a value, or

(2) e --> e'

Preservation Theorem:

If $\vdash e : t$ and e --> e' then $\vdash e' : t$

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. :-)

Next Time

From typing rules to a type checker implementation!