Type Checking

COS 326
David Walker
Princeton University
let x = 3 in
x + x

Parsing
Let ("x",
    Num 3,
    Binop(Plus, Var "x", Var "x"))

Evaluation
Num 6

Pretty Printing
6
Implementing an Interpreter

```
let x = 3 in
x + x
```

1. Parsing
2. Type Checking
3. Evaluation
4. Pretty Printing
5. Num 6
6. Num 6
7. Let ("x", Num 3, Binop(Plus, Var "x", Var "x"))
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
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

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

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

x -- ranges over variable names

c ::= n | b

x ::= + | - | <

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 x = e in e
Recall Inference Rule Notation

When defining how evaluation worked, we used this notation:

\[
\begin{align*}
e_1 \rightarrow \lambda x. e & \quad e_2 \rightarrow v_2 & \quad e[v_2/x] \rightarrow v \\
es_1 \ e_2 \rightarrow v
\end{align*}
\]

In English:

“if e_1 evaluates to a function with argument x and body e and e_2 evaluates to a value v_2 and e with v_2 substituted for x evaluates to v then e_1 applied to e_2 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:

\[ e \rightarrow v \]

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).
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: $\vdash e : t$
Here's an example context:

\[ x: \text{int}, \ y: \text{bool}, \ z: \text{int} \]

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 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 $e$. 
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.)
Rule for constant booleans:

\[ G |- b : \text{bool} \]

English:

“boolean constants b \textit{always} have type bool, no matter what the context G is"
Rule for constant integers:

\[ G |- n : \text{int} \]

English:

“integer constants \( n \) \textit{always} have type \( \text{int} \), no matter what the context \( G \) is"
Typing Contexts and Free Variables

\[ t ::= \text{int} \mid \text{bool} \mid t \rightarrow t \]

\[ c ::= n \mid b \]

\[ o ::= + \mid - \mid < \]

\[ e ::= c \mid \text{e e} \mid x \mid \text{if e then e else e} \mid \lambda x: t. e \mid \text{let x = e in e} \]

Rule for constant integers:

\[
\begin{align*}
G |- e1 : t1 & \quad G |- e2 : t2 & \quad \text{optype}(o) = (t1, t2, t3) \\
G |- e1 o e2 : t3
\end{align*}
\]

where

\[
\begin{align*}
\text{optype} (+) &= (\text{int}, \text{int}, \text{int}) \\
\text{optype} (-) &= (\text{int}, \text{int}, \text{int}) \\
\text{optype} (<) &= (\text{int}, \text{int}, \text{bool})
\end{align*}
\]

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

\[ t ::= \text{int} \mid \text{bool} \mid t \rightarrow t \]
\[ c ::= n \mid b \]
\[ o ::= + \mid - \mid < \]
\[ e ::= c \]
\[ e \circ e \]
\[ x \]
\[ \text{if } e \text{ then } e \text{ else } e \]
\[ \lambda x : t . e \]
\[ e \]
\[ e \]
\[ \text{let } x = e \text{ in } e \]

Rule for variables:

\[
G(x) = t \\
G |- x : t
\]

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 
| \(\lambda x : t . e\) 
| e e 
| let x = e in e

Rule for if:

\[
\begin{align*}
G &\vdash e_1 : \text{bool} & G &\vdash e_2 : t & G &\vdash e_3 : t \\
G &\vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\end{align*}
\]

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

\[
t ::= \text{int} | \text{bool} | t \rightarrow t
\]

\[
c ::= n | b
\]

\[
o ::= + | - | <
\]

\[
e ::= c
\]

\[
| e \circ e
\]

\[
| x
\]

\[
| \text{if } e \text{ then } e \text{ else } e
\]

\[
| \lambda x : t. e
\]

\[
| e \ e
\]

\[
| \text{let } x = e \text{ in } e
\]

Rule for functions:

\[
\frac{G, x : t \vdash e : t_2}{G \vdash \lambda x : t. e : t \rightarrow t_2}
\]

English:

“if \( G \) extended with \( x : t \) proves \( e \) has type \( t_2 \) then \( \lambda x : t. e \) has type \( t \rightarrow t_2 \)"
Typing Contexts and Free Variables

\[ t ::= \text{int} \mid \text{bool} \mid t \to t \]

\[ c ::= n \mid b \]

\[ o ::= + \mid - \mid < \]

\[ e ::= c \]

\[ | e \circ e \]

\[ | x \]

\[ | \text{if } e \text{ then } e \text{ else } e \]

\[ | \lambda x : t. e \]

\[ | e \ e \]

\[ | \text{let } x = e \text{ in } e \]

Rule for function call:

\[ G \mid - e_1 : t_1 \to t_2 \quad G \mid - e_2 : t_1 \]

\[ G \mid - e_1\ e_2 : t_2 \]

English:

"if G extended with x:t proves e has type t2 then \( \lambda x : t . e \) has type t \to t2 "
Typing Contexts and Free Variables

**Rule for let:**

\[
\begin{align*}
G |\vdash e_1 : t_1 & \quad G, x : t_1 |\vdash e_2 : t_2 \\
G |\vdash \text{let } x = e_1 \text{ in } e_2 : t_2
\end{align*}
\]

**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 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).

\[
\begin{align*}
G, x: \text{int} & \vdash (x) = \text{int} \\
G, x: \text{int} & \vdash - x : \text{int} & \text{G,x:int} & \vdash 2 : \text{int} \\
G, x: \text{int} & \vdash - x + 2 : \text{int} \\
G & \vdash \lambda x: \text{int}. x + 2 : \text{int} \rightarrow \text{int}
\end{align*}
\]
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. \(e \rightarrow e'\)

**Preservation Theorem:**
If \(- e : t\) and \(e \rightarrow e'\) 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 ...