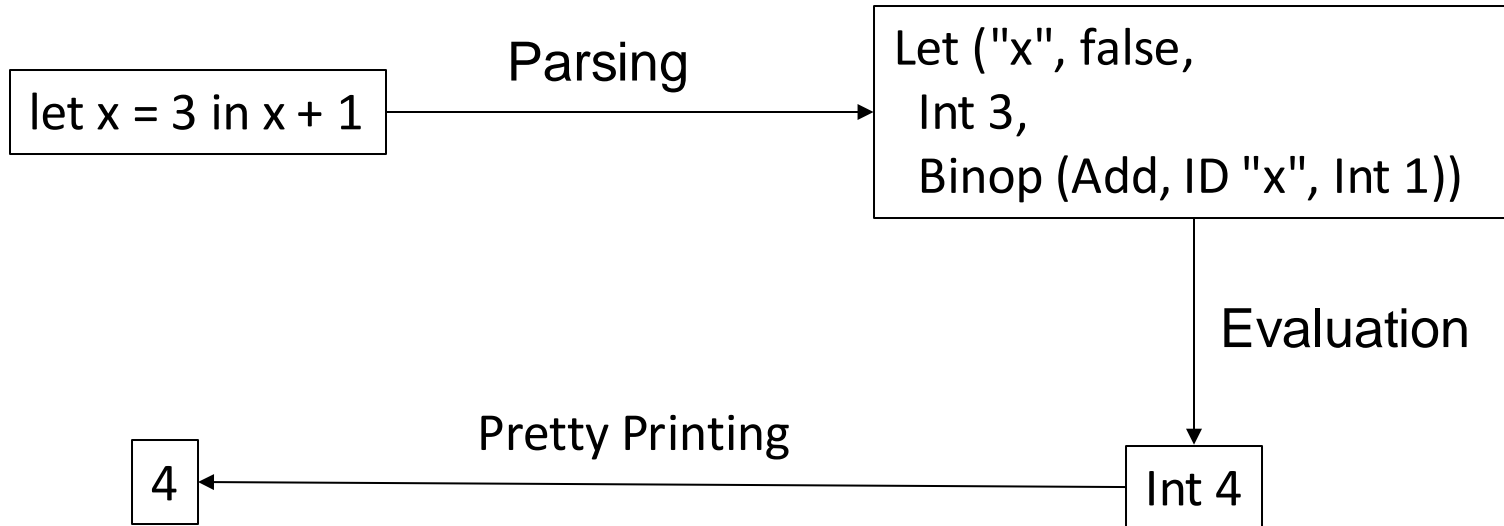


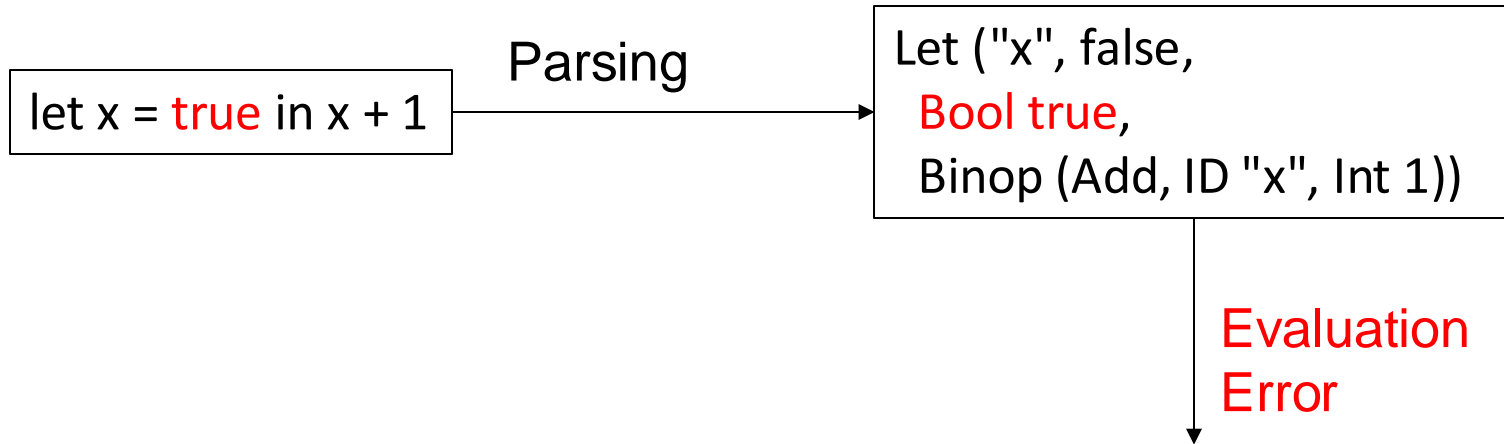
CMSC 330: Organization of Programming Languages

Type Checking

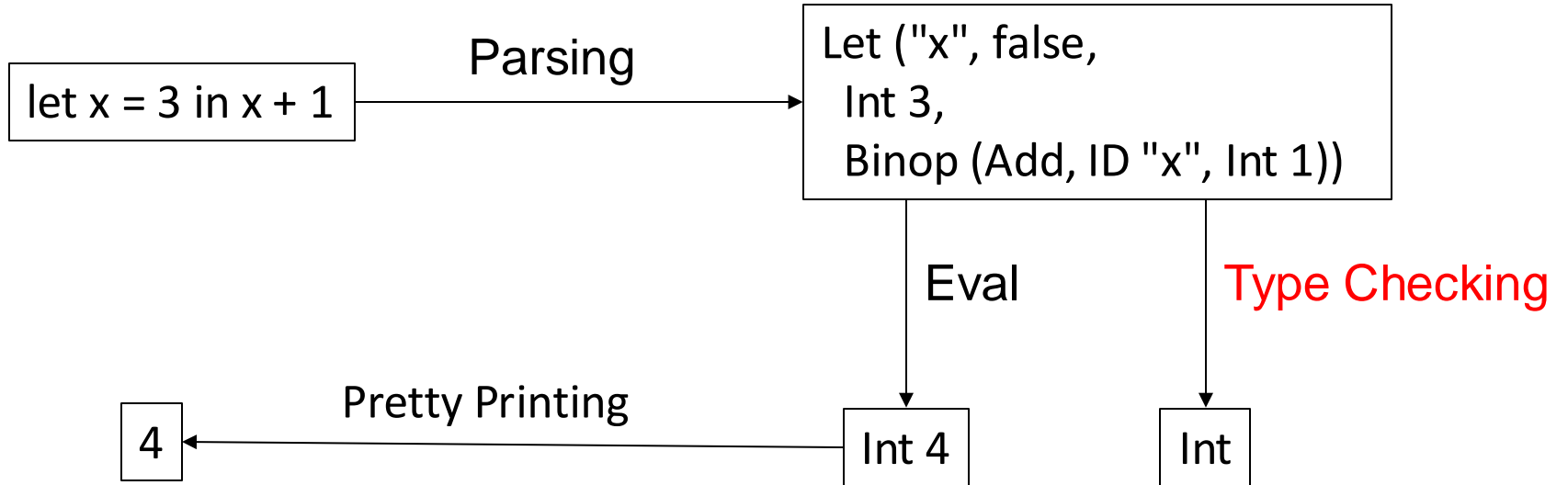
Implementing an Interpreter



Implementing an Interpreter: type error



Type Checking



Type Systems

- A **type system** is a series of **rules** that ascribe types to expressions
 - The rules prove statements $e : t$
 - A mechanism for distinguishing good programs from bad
 - Good programs = well typed
 - Bad programs = ill-typed or not typable
 - Example:
 - $0 + 1$ // well typed
 - $false\ 0$ // ill-typed: can't apply a Boolean
 - $1 + (\text{if true then } 0 \text{ else false})$ // ill-typed: can't add boolean to integer
- The process of applying these rules is called **type checking**
 - Or simply, **typing**
- Different languages have different type systems

Recall Inference Rules

- ▶ When defining how evaluation worked, we used this notation:

$$A; e1 \Rightarrow v1 \quad A, x: v1; e2 \Rightarrow v2$$

$$A; \text{let } x = e1 \text{ in } e2 \Rightarrow v2$$

- ▶ We used inference **rules** to define **judgment** $A: e \Rightarrow v$ and translated rules into an **interpreter** for the MicroOCaml language.
- ▶ $A: e \Rightarrow v$ was read in English as “e evaluates to v in an Environment A

Type Checking

- ▶ Inference rules can also be used to specify a program's **static semantics**, i.e., the rules for type checking
- ▶ Judgment

$G \vdash e : t$

- ▶ is read in English as "**e** has type **t** in context **G**."
- ▶ We define inference rules for this judgment, just as with the operational semantics

Typing Contexts

- ▶ What is the type checking context G ?
 - G is a (partial) function that maps variable names to types.

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

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

- ▶ Example context: $x:int, y:bool, z:int$
- ▶ When G is empty, we just write: $e:t$

Typing Contexts and Free Variables

- 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**.

e = Binop (Add, ID "x", Binop (Add, ID "y", ID "z"))

G:

ID	Type
x	Int
y	Int
z	Int

Type Checking Rules

- ▶ **Goal:** Give rules that define the relation " $G \vdash e : t$ ".
 - We give one rule for every sort of expression.

type expr =

```
  Int of int
| Bool of bool
| ID of var
| Fun of var * exptype * expr
| Not of expr
| Binop of op * expr * expr
| If of expr * expr * expr
| App of expr * expr
| Let of var * bool * expr * expr
| Record of (label * expr) list
| Select of label * expr
```

Type Checking Rules: Booleans

- ▶ Boolean constants have type `bool`

$$\frac{}{G \vdash \text{true} : \text{bool}}$$
$$\frac{}{G \vdash \text{false} : \text{bool}}$$

- ▶ Boolean constants `b` *always* have type `bool`, no matter what the context `G` is"

Type Checking Rules: Integers

- ▶ Integers have type Int

$$G \vdash n : \text{Int}$$

- ▶ Integer constants n *always* have type **Int**, no matter what the context G is"

Type Checking Rules: Binary Operators

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

- ▶ Where:
 - $\text{optype}(+, -, *, /) = (\text{Int}, \text{Int}, \text{Int})$
 - $\text{optype}(=, !=) = ('a, 'a, \text{Bool})$
 - $\text{optype}(<, >, <=, >=) = (\text{int}, \text{int}, \text{bool})$
 - $\text{optype}(\&\&, ||) = (\text{Bool}, \text{Bool}, \text{Bool})$
- ▶ **$e1 \text{ op } e2$** has type **$t3$** , if **$e1$** has type **$t1$** , **$e2$** has type **$t2$** and **op** is an operator that takes arguments of type **$t1$** and **$t2$** and returns a value of type **$t3$**

Type Checking Rules: Variables

Rule for variables:

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

- ▶ Variable x has the type given by the context

Type Checking Rules: Conditionals

- ▶ Eq0:

$$\frac{G \vdash e : \text{int}}{G \vdash \text{eq0 } e : \text{bool}}$$

- ▶ If

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

- ▶ If **e1** has type **bool**, and **e2** has type **t**, and **e3** has (the same) type **t** then **if e1 then e2 else e3** has type **t**

Type Checking Rules: Let

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

- ▶ 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**

Type Checking Rules: Functions

$$\frac{G, x:t1 \vdash e : t2}{G \vdash \text{fun } x:t1 \rightarrow e : t1 \rightarrow t2}$$

- ▶ if G extended with $x:t1$ proves e has type $t2$ then $\text{fun } x \rightarrow e$ has type $t1 \rightarrow t2$

Type Checking Rules: Function Call

$$\frac{G \vdash e1 : t1 \rightarrow t2 \quad G \vdash e2 : t1}{G \vdash e1 \ e2 : t2}$$

- ▶ If **e1** has type **t1** → **t2** and **e2** has type **t1** then **e1 e2** has type **t2**

Type Checking Rules: Record

- ▶ Record:

$$\frac{G \vdash e_1 : t_1 \dots G \vdash e_n : t_n}{G \vdash \{l_1 = e_1 \dots l_n = e_n\} : l_1 : t_1 \dots l_n : t_n}$$

- ▶ Select

$$\frac{\begin{array}{l} G \vdash e_1 : t_1 \dots G \vdash e_n : t_n, \\ G \vdash \{l_1 = e_1 \dots l_n = e_n\} : l_1 : t_1 \dots l_n : t_n \end{array}}{G \vdash e.l_i : t_i}$$

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.

$$G, x: \text{int} \vdash x: \text{int} \quad G, x: \text{int} \vdash 2: \text{int}$$

$$G, x: \text{int} \vdash x+2: \text{int}$$

$$G \vdash \text{fun } x: \text{int} \rightarrow (x+2) : \text{int} \rightarrow \text{int}$$

Type Safety

- ▶ A **well-typed** program is accepted by the language's **type system**
- ▶ A program **going wrong** is one that the language's semantics gives no definition (**undefined**)
 - If the program were to be run, anything could happen
 - `char buf[4]; buf[4] = 'x'; // undefined!`
- ▶ A **type-safe language** is one in which for every program, **well-typed** \Rightarrow **well-defined**
 - Or, *Well-typed programs never go wrong*, in the words of Robin Milner in 1978

Dynamic Type Checking

- ▶ The run-time checks performed by dynamic languages often called **dynamic type checking**
 - These languages may be said to have a **dynamic type system**
- ▶ The “type” of an expression checked as needed
 - Values keep **tag**, set when the value is created, indicating its type (e.g., what class it has)
- ▶ Disallowed operations cause run-time exception
 - **Type errors may be latent in code for a long time**

Quiz 1

- ▶ When is the type of a variable determined in a **dynamically typed** language?
 - A. When the program is compiled
 - B. At run-time, when the variable is used
 - C. At run-time, when that variable is first assigned to
 - D. At run-time, when the variable is last assigned to

Quiz 1

- ▶ When is the type of a variable determined in a **dynamically typed** language?
 - A. When the program is compiled
 - B. At run-time, when the variable is used
 - C. At run-time, when that variable is first assigned to
 - **D. At run-time, when the variable is last assigned to**

Quiz 2

- ▶ When is the type of a variable determined in a **statically typed** language?
 - A. When the program is compiled
 - B. At run-time, when the variable is used
 - C. At run-time, when that variable is first assigned to
 - D. At run-time, when the variable is last assigned to

Quiz 2

- ▶ When is the type of a variable determined in a **statically typed** language?
 - A. When the program is compiled
 - B. At run-time, when the variable is used
 - C. At run-time, when that variable is first assigned to
 - D. At run-time, when the variable is last assigned to

Static vs. Dynamic Type Systems

- ▶ OCaml, Java, Haskell, etc. are **statically typed**
- ▶ Ruby, Python, etc. are **dynamically typed**
- ▶ But we can *view* dynamically typed languages as statically typed in a particular sense:
 - Can view all expressions as having a static type **Dyn**
 - The language is **uni-typed**
 - *All* operations are permitted on values of this type
 - E.g., in Ruby, all objects accept any method call
 - **But: Some operations result in a run-time exception**
 - Those not supported by the value's dynamic "type" (tag)
 - Nevertheless, such behavior is **well defined**

Soundness and Completeness

- ▶ Type safety is a **soundness** property
 - That a term type checks implies its execution will be well-defined
- ▶ **Static** type systems are rarely **complete**
 - That a term is well-defined *does not* imply that it will type check
 - `if true then 0 else 4+"hi"`
- ▶ **Dynamic** type systems are often **complete**
 - *All* expressions are well defined and (statically) type check
 - `4+"hi"` well-defined: it gives a run-time exception

Quiz 3

- ▶ Which of the following is well-defined in OCaml, but is **not well-typed**?
 - A. `let f g = (g 1, g "hello") in f (fun x -> x)`
 - B. `List.map (fun x -> x + x) [1; "hello"]`
 - C. `let x = 0 in 5 / x`
 - D. `let x = Array.make 1 1 in x.(2)`

Quiz 3

- ▶ Which of the following is well-defined in OCaml, but is **not well-typed**?

Functions as arguments cannot be used polymorphically

- A. `let f g = (g 1, g "hello") in f (fun x -> x)`
- B. `List.map (fun x -> x + x) [1; "hello"]`
- C. `let x = 0 in 5 / x`
- D. `let x = Array.make 1 1 in x.(2)`

Ill-typed and ill-defined

well-typed and well-defined

well-typed and well-defined

Perfect Type System? Impossible

- ▶ **No type system** can do all of following
 - (1) always terminate, (2) be sound, (3) be complete
 - While trying to eliminate all run-time exceptions, e.g.,
 - Using an int as a function
 - Accessing an array out of bounds
 - Dividing by zero, ...
- ▶ Doing so would be **undecidable**
 - by reduction to the halting problem
 - Eg., `while (...) {...} arr[-1] = 1;`
 - *Error tantamount to proving that the while loop terminates*

Static vs. Dynamic Type Checking

Having carefully stated facts about static checking, can *now* consider arguments about which is *better*:

static checking or dynamic checking

Poll: Which Do You Prefer?

- ▶ (a) static type systems (e.g., Java, Ocaml)
- ▶ (b) dynamic type systems (e.g., Ruby, Python)

Claim 1: Dynamic is more convenient

- ▶ Dynamic typing lets you build a heterogeneous list or return a “number or a string” without workarounds

Ruby: `a = [1,1.5]`

OCaml:

```
type t =  
  Int of int  
  | Float of float  
  
let a = [Int 1; Float 1.5];;
```

Claim 1: Static is more convenient

- ▶ Can assume data has the expected type without cluttering code with dynamic checks or having errors far from the logical mistake

Ruby:

```
def cube(x)
  if x.is_a?(Numeric)

    x * x * x
  else
    "Bad argument"
  end
end
```

OCaml:

```
let cube x = x * x * x
(* we know x is int *)
```

Claim 2: Static prevents useful programs

- ▶ Any sound static type system forbids programs that do nothing wrong

Ruby:

```
if e1 then
  "lady"
else
  [7, "hi"]
end
```

OCaml:

```
if e1 then "lady" else (7, "hi")
(* does not type-check *)
```

Claim 2: But always workarounds

- ▶ Rather than suffer time, space, and late-errors costs of tagging everything, statically typed languages let programmers “tag as needed” (e.g., with types)

Ruby: Tags everything implicitly (uni-typed)

OCaml: Tag explicitly, as needed (code up unifying type)

```
type tort = Int of int
          | String of string
          | Cons of tort * tort
          | Fun of (tort -> tort)
          | ...
```

```
if e1 then
  String "lady"
else
  Cons (Int 7, String "hi")
```

Claim 3: Static catches bugs earlier

- ▶ Static typing catches many simple bugs as soon as “compiled”.
 - Since such bugs are always caught, no need to test for them.
 - In fact, can code less carefully and “lean on” type-checker

Ruby:

```
def pow (x,y)
  if y == 0 then
    1
  else
    x * pow (y - 1)
  end
end
# can't detect until run
```

OCaml:

```
let pow x y =
  if y = 0 then 1
  else x * pow (y-1)
  (* does not type-check *)
```

Claim 3: Static catches only easy bugs

- ▶ But static often catches only “easy” bugs, so you still have to test your functions, which should find the “easy” bugs too

Ruby:

```
def pow (x,y)
  if y == 0 then
    1
  else
    x + pow (x, (y-1))
  end
end
```

OCaml:

```
let pow x y =
  if y = 0 then 1
  else x + pow x (y-1)

(* oops *)
```

Claim 4: Static typing is faster

- ▶ Language implementation:
 - Does not need to store tags (space, time)
 - Does not need to check tags (time)
 - Can rely on values being a particular type, so it can perform more optimizations
- ▶ Your code:
 - Does not need to check arguments and results beyond what is evidently required

Claim 4: Dynamic typing is not too much slower

- ▶ Language implementation:
 - Can use remove some unnecessary tags and tests despite the lack of types
 - While difficult (impossible) in general, it is often possible for the performance-critical parts of a program
- ▶ Your code:
 - Do not need to “code around” type-system limitations with extra tags, functions etc.

Claim 5: Code reuse easier with dynamic

Without a restrictive type system, more code can just be reused with data of different types

- ▶ If you use cons cells for everything, libraries that work on cons cells are useful
- ▶ Collections libraries are amazingly useful but often have very complicated static types
 - Polymorphism/generics/etc. are hard to understand, but are aiming to provide what dynamic typing gives naturally
- ▶ Etc.

Claim 5: Code reuse easier with static

The type system serves as “checked documentation,” making the “contract” with others’ code easier to understand and use correctly

Redux: Which Do You Prefer?

- ▶ (a) static type systems (e.g., Java, Ocaml)
- ▶ (b) dynamic type systems (e.g., Ruby, Python)

Static vs. Dynamic: Age-old Debate

- ▶ Static vs. dynamic typing is too coarse a question
 - Better question: *What should we enforce statically?*
 - E.g., OCaml checks array bounds, division-by-zero, at run-time
 - Legitimate trade-offs
- ▶ Idea: Flexible languages allowing *best-of-both-worlds?*
 - Use static types in some parts of the program, but dynamic checking in other parts?
 - Called *gradual typing*: an idea still under active research
 - Would programmers use such flexibility well? Who decides?