CMSC 330: Organization of Programming Languages

Type Checking
Implementing an Interpreter

```
let x = 3 in x + 1
```

Parsing:
```
Let ("x", false, 
    Int 3, 
    Binop (Add, ID "x", Int 1))
```

Evaluation:
```
Int 4
```

Pretty Printing:
```
4
```
Implementing an Interpreter: type error

let x = true in x + 1

Parsing

Let ("x", false,
    Bool true,
    Binop (Add, ID "x", Int 1))

Evaluation
Error
Type Checking

let x = 3 in x + 1

Parsing

Let ("x", false, Int 3, Binop (Add, ID "x", Int 1))

Eval

Pretty Printing

4

Int 4

Type Checking

Int
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
      - $\text{false } 0$ // 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:

\[
\begin{align*}
A; e_1 \Rightarrow v_1 & \quad A, x: v_1; e_2 \Rightarrow v_2 \\
\hline
A; \text{let } x = e_1 \text{ in } e_2 & \Rightarrow v_2
\end{align*}
\]

- 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.
  
  \[ G(x) \rightarrow \text{look up } x's \text{ type in } G \]
  \[ G,x:t \rightarrow \text{extend } G \text{ so that } x \text{ maps to } t \]

- Example context: \( x:\text{int}, y:\text{bool}, z:\text{int} \)
- 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 $e$.

$$e = \text{Binop} \ (\text{Add}, \text{ID} \ "x" , \ \text{Binop} (\text{Add}, \text{ID} \ "y" , \text{ID} \ "z" ))$$

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>Int</td>
</tr>
<tr>
<td>$y$</td>
<td>Int</td>
</tr>
<tr>
<td>$z$</td>
<td>Int</td>
</tr>
</tbody>
</table>
Type Checking Rules

**Goal:** Give rules that define the relation "$G \vdash e : t$".

- We give one rule for every sort of expression.

```olang
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**

| G ⊢ true : bool | G ⊢ false : 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

Where:

- `optype (+, -, *, /) = (Int, Int, Int)`
- `optype (=, !=) = ('a, 'a, Bool)`
- `optype (<, >, <=, >=) = (int, int, bool)`
- `optype (&&, ||) = (Bool, Bool, Bool)`

- `e1 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**

- Variable $x$ has the type given by the context $\Gamma \vdash x : G(x)$
Type Checking Rules: Conditionals

- Eq0:

\[
\begin{align*}
G \vdash e : \text{int} \\
G \vdash \text{eq0 e} : \text{bool}
\end{align*}
\]

- If

\[
\begin{align*}
G \vdash e1 : \text{bool} & \quad G \vdash e2 : t \quad G \vdash e3 : t \\
G \vdash \text{if e1 then e2 else e3} : t
\end{align*}
\]

- If \(e1\) has type \text{bool}, and \(e2\) has type \(t\), and \(e3\) has (the same) type \(t\) then \(\text{if e1 then e2 else e3}\) has type \(t\)
Type Checking Rules: Let

- If \( e_1 \) has type \( t_1 \) and \( G \) extended with \( x : t_1 \) proves \( e_2 \) has type \( t_2 \) then \( \text{let } x = e_1 \text{ in } e_2 \) has type \( t_2 \)
Type Checking Rules: Functions

- if $G$ extended with $x : t_1$ proves $e$ has type $t_2$ then $\text{fun } x : t_1 \rightarrow e : t_1 \rightarrow t_2$ has type $t_1 \rightarrow t_2$
Type Checking Rules: Function Call

- If $e_1$ has type $t_1 \rightarrow t_2$ and $e_2$ has type $t_1$ then $e_1 \ e_2$ has type $t_2$
Type Checking Rules: Record

- **Record:**

  \[
  \frac{G \vdash e_1 : t_1 \ldots G \vdash e_n : t_n}{G \vdash \{l_1 = e_1 \ldots l_n = e_n\} : l_1 : t_1 \ldots l_n : t_n}
  \]

- **Select**

  \[
  \frac{G \vdash e_1 : t_1 \ldots G \vdash e_n : t_n,}{G \vdash \{l_1 = e_1 \ldots l_n = e_n\} : l_1 : t_1 \ldots l_n : t_n}{G \vdash e.l_i : t_i}
  \]
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.

\[
\begin{align*}
G, x: \text{int} \vdash x: \text{int} & \quad G, x: \text{int} \vdash 2: \text{int} \\
\hline
G, x: \text{int} \vdash x + 2: \text{int} \\
\hline
G \vdash \text{fun } x: \text{int} \rightarrow (x + 2): \text{int} \rightarrow \text{int}
\end{align*}
\]
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 $\implies$ well-defined.

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 $\text{Dyn}$
    - 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
    - $\text{if true then 0 else 4+"hi"}$

- **Dynamic type systems are often complete**
  - *All* expressions are well defined and (statically) type check
  - $4+"\text{hi}"$ 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?

- 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)

Functions as arguments cannot be used polymorphically

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:

```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:

```ruby
def cube(x)
  if x.is_a?(Numeric)
    x * x * x
  else
    "Bad argument"
  end
end
```

OCaml:

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

OCaml:
```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)

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

```ocaml
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:

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

OCaml:

```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?