CMSC 330: Organization of Programming Languages

Type Systems

Types: Recall our Intro to OCaml

- Types classify expressions
  - Characterize the set of possible values an expression could evaluate to
  - We use metavariable ***t*** to designate an arbitrary type
    - Examples include **`int`**, **`bool`**, **`string`**, and more.
- Expression ***e*** has type ***t*** if ***e*** will (always) evaluate to a value of type ***t***
  - { …, **`-1`**, **`0`**, **`1`**, … } are values of type **`int`**
  - **`34+17`** is an expression of type **`int`**, since it evaluates to **`51`**, which has type **`int`**
  - Write ***e : t*** to say ***e*** has type ***t***

Type Systems

- A type system is a series of rules that ascribe types to expressions
  - The rules prove statements ***e : t***
- The process of applying these rules is called type checking
  - Or simply, typing
  - Type checking *aka* the program's static semantics
- Different languages have different type systems

OCaml Type System: Conditionals

- Syntax
  - `if` ***e1*** `then` ***e2*** `else` ***e3***

- Type checking
  - If ***e1*** : `bool` and ***e2*** : ***t*** and ***e3*** : ***t*** then
    `if` ***e1*** `then` ***e2*** `else` ***e3*** : ***t***
  - *More formally:*

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

Type Safety

- Well-typed
  - A well-typed program passes the language's type system
- Going wrong
  - The language definition deems the program nonsensical
    - "Colorless green ideas sleep furiously"
    - If the program were to be run, anything could happen
    - char buf[4]; buf[4] = 'x'; // undefined!
- Type safe = "Well-typed programs never go wrong"
  - Robin Milner, 1978
  - In other words: Well-typed $\Longrightarrow$ well-defined

Type Safe?

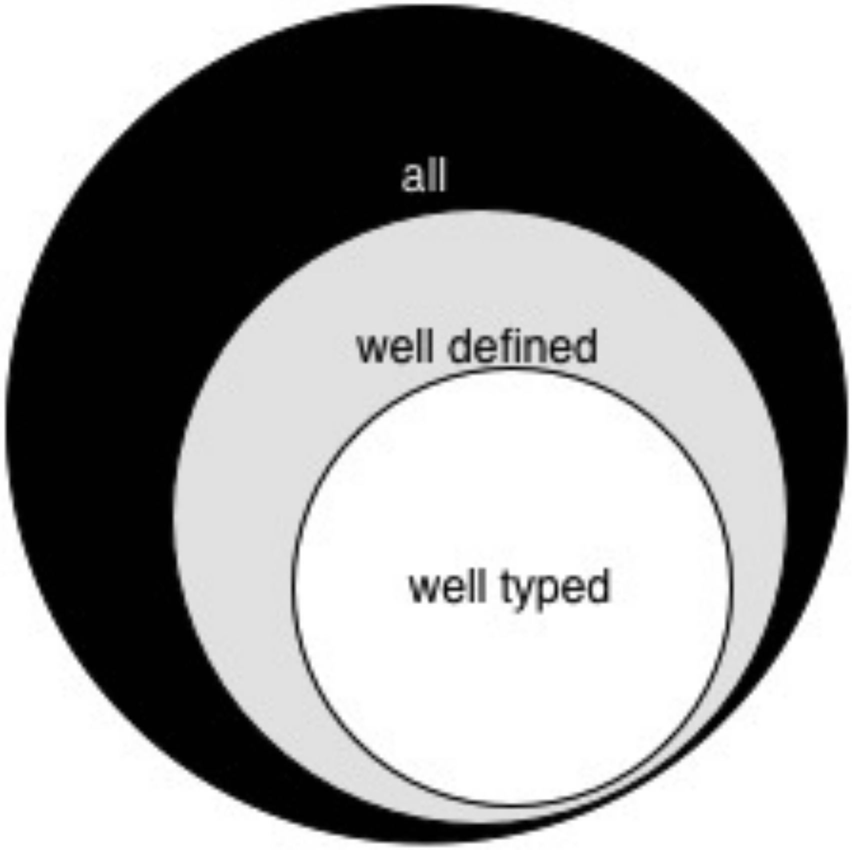
- Java, Haskell, OCaml: **Yes** (arguably).
  - The languages' type systems restrict programs to those that are defined
    - Caveats: Foreign function interfaces to type-unsafe C, bugs in the language design, bugs in the implementation, etc.

- C, C++: **No**.
  - The languages' type systems do not prevent undefined behavior
    - Unsafe casts (int to pointer), out-of-bounds array accesses, dangling pointer dereferences, etc.

What's Bad about Being Undefined?

- Well, undefined behavior is unconstrained
  - Depends on the compiler/interpreter's treatment
- Undefined behavior in C/C++ is traditionally a source of severe security vulnerabilities
  - These are bugs that have security consequences
- Stack smashing exploits out-of-bounds array accesses to inject code into a running program
  - Write outside the bounds of an array (undefined!)
  - thereby corrupting the return address
  - to point to code the attacker provides
  - to gain control of the attacked machine

Type Safety is Often Conservative



I.e., some well-defined programs are *not* well typed

Well-defined but not Well-typed

- In OCaml, the expression `4+"hi"` is undefined
  - Ocaml's type system does not typecheck this expression, ensuring it is never executed
    - Good!

- But the following expressions are well-defined, but still rejected
  - `if true then 0 else 4+"hi"`
    - Always evaluates to 0
  - `let f4 x = if x <= abs x then 0 else 4+"hi"`
    - `f4 e` evaluates to `0` for all (`e : int`)

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 all type check
  - **`4+"hi"`** well-defined: it gives a run-time exception

Static vs. Dynamic Type Systems

- OCaml, Java, Haskell, etc. are statically typed
  - Expressions are given one of various different types at compile time, e.g., **`int`**, **`float`**, **`bool`**, etc.
    - Or else they are rejected
- Ruby, Python, etc. are dynamically typed
  - Can view all expressions as having a single 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
    - Nevertheless, such behavior is well defined

Dynamic Type Checking

- The run-time checks performed by dynamic languages often called dynamic type checking
- The type of an expression checked when 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 that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

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 that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

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 that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

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 that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

Devil's Bargain?

- Dynamic typing is sound and complete
  - That seems good …
- But it trades compile-time errors for (well-defined) run-time exceptions!
- Can't we build a better static type system?
  - I.e., that that aims to eliminate all language-level run-time errors and is also complete?
- Yes, we can build more precise static type systems, but never a perfect one
  - To do so would be undecidable!

Fancy Types

- Lots of ideas over the last few decades aimed at improving the precision of type systems
  - So they can rule out more run-time errors
- Generic types (parametric polymorphism)
  - for containers and generic operations on them
- Subtyping
  - for interchanging objects with related shapes
- Dependent types can include *data in types*
  - Instead of **`int list`**, we could have **`int`** ***n*** **`list`** for a list of ***n*** elements. Hence **`hd`** has type **`int`** ***n*** **`list`** where ***n***>0.

Type Systems with Fancy Types

- OCaml's type system has types for
  - generics (polymorphism), objects, curried functions, …
  - all unsupported by C
- Haskell's type system has types for
  - Type classes (qualified types), effect-isolating monads, higher-rank polymorphism, …
  - All unsupported by OCaml
- More precision ensures more run-time errors prevented, with less contorted programs: Good!
  - But now the programmer must understand (and sometimes do) more ..

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*

Type Checking and Type Inference

- Type inference is a part of (static) type checking
  - Reduces the programmer's effort
- Static types are explicit (*aka* manifest) or inferred
  - Manifest – specified in text (at variable declaration)
    - C, C++, Java, C#
  - Inferred – compiler determines type based on usage
    - OCaml, C# and Go (limited)
- Fancier type systems may require explicit types
  - Haskell considers adding a type signature your function to be good style, even when not required

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

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

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?

Weak vs. Strong Typing

- Weak typing
  - Allows one type to be treated as another or provides (many) implicit casts
  - Example (int treated as bool)
    - C
      ```
      int i = 1 ;
      if (i)                    // checks for 0
          printf("%d", i);
      ```
    - Ruby
      ```
      i = 1
      if i                      // checks for nil
          puts i
      end;
      ```
  - Example languages
    - C, C++, Ruby, Perl, Javascript

Weak vs. Strong Typing (cont.)

- Strong typing
  - Prevents one type from being treated as another, implicitly
  - Example (int not treated as bool)
    - Java

```
int i = 1 ;
if (i)                    // error, not bool
  System.out.println(i);
```

    - OCaml

```
let i = 1 in
  if i then               // error, not bool
    print_int i
```

  - Example languages
    - Java (rare exceptions), OCaml

Terms: Strong vs. Weak Typing

- These terms are not illuminating, or even agreed upon
  - “strong typing” is often confused with “type safety” or “static typing”
  - Supporting implicit casts, or not, is not particularly interesting as a language feature
    - And is confused with features like subtyping
- Other terms we’ve discussed are more well understood