Review

• Done with OOP features: interfaces, multiple inheritance, mixins, double dispatch, subtyping

• This week: two big debates
  – Static vs. dynamic typing (today)
  – Strict vs. lazy evaluation (tomorrow)

Static vs. dynamic typing

• A big, juicy, essential, topic about how to think about PLs
  – Conversation usually overrun with half-informed opinions 😕
  – Will consider reasonable arguments "for" and "against" in second half of lecture

• First need to understand:
  – What static checking means
  – What static checking intends to accomplish
  – Why static checking must be approximate
  – The standard features of a type system
  – How eager error detection should be

Static checking

• Static checking is anything done to reject a program after it (successfully) parses but before it runs

• How much and what static checking is done?
  – That’s part of the language definition
  – Third-party tools can do more

• Most common way to define a PL’s static checking is via a type system
  – Approach is to give each variable, expression, etc. a type
  – Purposes include preventing misuse of primitives (e.g., 4/"hi") and avoiding dynamic checking (dynamic means at run-time)

• Dynamically typed PLs (e.g., Ruby) do much less static checking
  – Maybe none, but the line is fuzzy

A question of eagerness

"Catching a bug before it matters" is in inherent tension with "Don’t report a bug that might not matter"

Static checking / dynamic checking are two points on a continuum

Silly example: Suppose we just want to prevent evaluating 3 / 0
  – Keystroke time: disallow it in the editor
  – Compile time: disallow it if seen in code
  – Link time: disallow it if seen in code that may be called to evaluate main
  – Run time: disallow it right when we get to the division
  – Later: Instead of doing the division, return INF instead

• Just like 3.0 / 0.0 does in every (?) PL (it’s useful)
Example: ML, what types do prevent

In ML, type-checking ensures a program (when run) will never:

• Use a primitive operation on a value of the wrong type
  – Use arithmetic on a non-number
  – Have \( e_1 \cdot e_2 \) where \( e_1 \) does not evaluate to a function
  – Have a non-boolean between \( \text{if} \) and \( \text{then} \)
• Use a variable that is not in the environment
• Have a pattern-match with a redundant pattern
• …

The first two are “standard” for type systems.

Example: ML, what types don’t prevent

In ML, type-checking does not prevent any of these errors:

• Exceptions don’t occur (e.g., \( \text{hd} \ [\ ] \) )
• An array-bounds error
• Division-by-zero
  Instead, these are detected at run-time

And in general no type system prevents logic or algorithmic errors:

• Reversing the branches of a conditional
• Using \( + \) instead of \( - \)
  Without a program specification, type checker can’t “read your mind”

Purpose is prevention

What the ML type system does and does not prevent is different from

How it prevent such things

…though we already studied many of ML’s typing rules

Different languages prevent different things:

– Java: prevents casting to types other than supertypes or subtypes,
  prevents missing field and method errors, prevents accessing
  private fields, etc.
– SML: prevents using \( = \) on anything other than equality types. (But
  OCaml lets you use \( = \) on any two types)

Part of language design is deciding what is checked and how

…hard part is making sure the type system does it correctly

True and false positives

<table>
<thead>
<tr>
<th>Test says you don’t have disease</th>
<th>Test says you do have disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>You really don’t have disease</td>
<td>True negative</td>
</tr>
<tr>
<td>You really do have disease</td>
<td>False negative</td>
</tr>
</tbody>
</table>

• Airport security: wristwatch mistaken for weapon …false positive
• Spam filter: desirable message is sent to spam folder …false positive
• Quality control: broken toy ships from factory …false negative

In type systems:

• Test is the type system. Patient is the program. Disease is “doing X.”

“Correctness”

Suppose a type system is supposed to prevent X for some X

• A type system is sound if it never accepts a program that, when run with
  some input, does X
  – No false negatives
• A type system is complete if it never rejects a program that, no matter
  what input it is run with, will not do X
  – No false positives

The goal is usually for a PL type system to be sound (so you can rely on it) but
not complete

– “Fancy features” (e.g., generics) make system “more” complete

True and false positives

<table>
<thead>
<tr>
<th>Type system accepts program</th>
<th>Type system rejects program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program doesn’t do X</td>
<td>True negative</td>
</tr>
<tr>
<td>Program does do X</td>
<td>False negative</td>
</tr>
</tbody>
</table>

• Soundness = no false negatives
• Completeness = no false positives

Soundness and completeness are always with respect to X
Incompleteness

A few functions ML rejects even though they do not divide by a string:

fun f1 x = 4 div "hi" (* but f1 never called *)
fun f2 x = if true then 0 else 4 div "hi"
fun f3 x = if x then 0 else 4 div "hi"
val x = f3 true
fun f4 x = if x <= abs x then 0 else 4 div "hi"
fun f5 x = 4 div x
val y = f5 if true then 1 else "hi"

(no examples of unsoundness, because ML type system is sound)

Why incompleteness

• Almost anything you might like to check statically is undecidable:
  – Any static checker that always terminates cannot be sound and complete
  
• Examples:
  – Will this function terminate on some input? On any input?
  – Will this function ever use a variable not in the environment?
  – Will this function treat a string as a function?
  – Will this function divide by zero?

• Undecidability is discussed in CSE 3313
  – The inherent approximation of static checking is probably its most important consequence

What about unsoundness?

Suppose a type system were unsound. What could the PL do?

• Fix the PL by updating the language definition
• Insert dynamic checks as needed to prevent X from happening
• Allow X or anything else to happen, including deleting your files, emailing your credit card number, or setting the computer on fire

PLs where the latter is allowed/expected are called weakly typed as opposed to strongly typed
  – Best-known example: C and C++

Why weak typing

Why define a language where there exist programs that, by definition, must pass static checking but then when run can set the computer on fire?

• Note: Dynamic checking is optional and in practice not done
• Ease of language implementation: Checks left to the programmer
• Performance: Dynamic checks take time
• Lower level: Compiler does not insert information like array sizes, so it cannot do the checks

Weak typing is a poor name: Really about doing neither static nor dynamic checks
  – A big problem is array bounds, which most PLs check dynamically

What weak typing has caused

• Old now-much-rarer saying: “strong types are for weak minds”
  – Idea was humans will always be smarter than a type system (cf. undecidability), so need to let them say “trust me”

• Reality: humans are really bad at avoiding bugs
  – We need all the help we can get!
  – And type systems have gotten much more expressive (less incomplete)

• 1 bug in a 30-million line OS written in C can make the whole OS vulnerable
  – An important bug like this was probably announced this week (because there is one almost every week)

Example: Ruby

• Ruby is not weakly typed
  – It just checks most things dynamically
  – (And is willing to convert just about anything to just about anything, albeit in a well-defined way)
  – Implementation can analyze the code to ascertain whether some checks aren’t needed, then optimize them away

• Not having ML or Java’s rules can be convenient
  – Arrays can hold anything
  – Everything is true except false and nil
  – Don’t need to create a datatype just to pass different types of data to a function
  – ...


**Example: Ruby**

In ML, we might complain about "false positives"

In Ruby, we might complain about not catching obvious errors

```ruby
def m = 4/"hi"
end
```

Much more frustrating when inside a large body of code

- Have to test for many more things without a sound type system to guarantee those things are prevented

**A different issue**

A related issue: what operations are primitives defined on

- Example: Is "foo" + "bar" allowed?
- Example: Is "foo" + 3 allowed?
- Example: Is arr[10] allowed if arr has only 5 elements?

This is not static vs. dynamic checking (sometimes confused with it)

- It is "what is the run-time semantics of the primitive"
- It is related because it also involves trade-offs between catching bugs sooner and sometimes being more convenient

Ruby is very lenient on these matters!

**Now consider costs and benefits**

Having carefully stated facts about static checking, we can now consider arguments about whether it is better or worse than dynamic checking

Remember most languages do some of each

- For example, perhaps types for primitives are checked statically, but array bounds aren't

**Claim 1a: Dynamic is more convenient**

Dynamic typing lets you build a heterogeneous list or return a "number or a string" without getting in your way

```ruby
def f y
  if y>0 then y*y else "hi" end
end
(f x)
```

```ruby
def f x
  if x.is_a? Fixnum
    x*x
  else
    raise "cube expects an integer"
  end
end
```

```ruby
fun f y = if y > 0 then (Int(y*y)) else String "hi" end
fun f x of
  Int i => Int.toString i
| String s => s
```

**Claim 1b: 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? Fixnum
    x*x*x
  else
    raise "cube expects an integer"
  end
end
cube 7
```

```ruby
fun cube x = x * x * x
cube 7
```

**Claim 2a: Static prevents useful programs**

Any sound static type system forbids programs that do nothing wrong, forcing the programmer to code around the limitation

```ruby
def f
  [yield(7), yield(true)]
end
```

```ruby
def f g = (g 7, g true) (* doesn’t type-check *)
```

```ruby
val pair_of_pairs = f (fn x => (x,x))
```

```ruby
```

```ruby
```
Claim 2b: Static lets you tag as needed

Rather than pay the time, space, and late-errors costs of tagging everything, statically typed languages let the programmer what is tagged (e.g., with datatypes).

In the extreme, we can use "OneRubyType" in ML:

```
datatype tort = Int of int |
              String of string |
              Cons of tort * tort |
              Fun of tort -> tort |
              ..
```

if el
then Fun (fn x => case x of Int i => Int (i*i*i))
else Cons (Int 7, String "hi")

Claim 3a: Static catches bugs earlier

Static typing catches tons of simple bugs as soon as you compile
- Since you know they are prevented, no need to test for them

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

fun pow x y = (* curried *)
if y = 0
then 1
else x * pow (x,y-1) (* does not type-check *)

Claim 3b: Static catches only easy bugs

But it usually catches only the "easier" bugs, so you still have to test your functions, which should find the "easier" bugs too

```
fun wrong_pow x y = (* curried *)
  if y == 0
    1
  else
    x + pow(x,y-1) (* oops *)
```

Claim 4a: Static typing is faster

The language implementation:
- Does not need to store tags (space, time)
- Does not need to check tags (time)

Your code:
- Does not need to check arguments and results
- Put tag tests just where needed

Claim 4b: Dynamic typing is faster

The language implementation:
- Can use optimization to remove some unnecessary tags and tests
- While that is hard (impossible) in general, it is often easier for the performance-critical parts of a program

Your code:
- Do not need to "code around" the type-system limitations, which can lead to extra functions, tags, etc.

Claim 5a: Code reuse easier with dynamic

By not requiring types, tags, etc., more code can just be reused with data of different types
- If you use arrays for everything, libraries that work on arrays are available
- Collections libraries are amazingly useful but often have very complicated static types
- Etc.

Claim 5b: Code reuse easier with dynamic

By not requiring types, tags, etc., more code can just be reused with data of different types
Claim 5b: Code reuse easier with static

- Modern type systems should support reasonable code reuse with features like generics and subtyping
- If you use arrays for everything, you will confuse what represents what and get hard-to-debug errors
  - Use separate static types to keep ideas separate
  - Static types help avoid library misuse

So far

Considered 5 things you care about when writing code:
1. Convenience
2. Not preventing useful programs
3. Finding bugs early
4. Performance
5. Code reuse

But we took the naïve view that software is developed by taking an existing spec, coding it up, testing it, and declaring victory. Reality:
- Often do a lot of prototyping before you have a stable spec
- Often do a lot of maintenance/evolution after version 1.0

Claim 6a: Dynamic better for prototyping

Early on, you don’t know what cases you need in your datatypes and your code
- But static typing won’t let you try code without having all cases; dynamic lets incomplete programs run
- So you make premature commitments to data structures
- And end up writing a lot of code to appease the type-checker that you are going to end up throwing away

When prototyping, conciseness matters more (?)

Claim 6b: Static better for prototyping

What better way to document your evolving decisions on data structures and code-cases than with the type system?

Easy to put in temporary stubs as necessary, such as
\[
| _ \Rightarrow \text{raise Unimplemented}
\]

Claim 7a: Dynamic better for evolution

Can change code to be more permissive without affecting old callers
- Example: Take an `int` or a `string` instead of an `int`
- All ML callers must now use a constructor on arguments and pattern-match on results
- Existing Ruby callers can be oblivious

```
def double x
    2 * x
end
```

```fun f x = 2 * x
```

Claim 7b: Static better for evolution

When we change type of data or code, the type-checker gives us a “to-do” list of everything that must change
- Avoids introducing bugs
- The more of your spec that is in your types, the more the type-checker lists what to change when your spec changes

Example: Changing the return type of a function

Example: Adding a new constructor to a datatype
- Good reason not to use wildcard patterns

Counter-argument: The to-do list is mandatory, which makes evolution in pieces a pain can’t “test what I’ve changed so far”
Coda

- Static vs. dynamic typing is too coarse a question
  - What should we enforce statically makes more sense

- There are real trade-offs here you should know
  - Allows for rational discussion informed by facts

- Ideally would have flexible languages that allow best-of-both-worlds
  - Still an open and active area of research