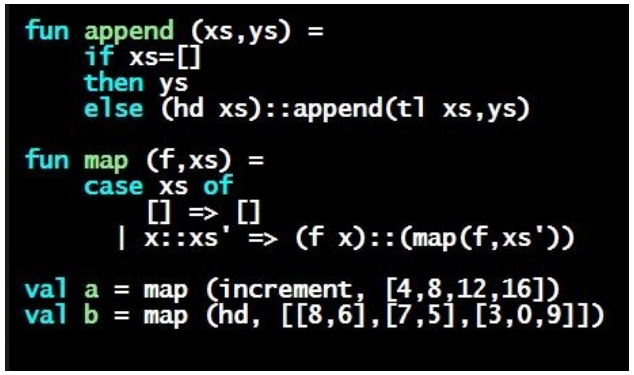

Programming Languages

Dan Grossman

2013

Static Versus Dynamic Typing, Part Two

Claim 4a: Static typing is faster

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

Claim 4b: Dynamic typing is faster

Language implementation:

- Can use optimization to remove some unnecessary tags and tests
  - Example: `(let ([x (+ y y)]) (* x 4))`
- 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" type-system limitations with extra tags, functions etc.

Claim 5a: 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
- Etc.

Claim 5b: Code reuse easier with static

- Modern type systems should support reasonable code reuse with features like generics and subtyping

- If you use cons cells 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 important when writing code:

1. Convenience
2. Not preventing useful programs
3. Catching bugs early
4. Performance
5. Code reuse

But took the naive view that software is developed by taking an existing spec, coding it up, testing it, and declaring victory.

Reality:

- Often a lot of prototyping *before* a spec is stable
- Often a lot of maintenance / evolution *after* version 1.0

Claim 6a: Dynamic better for prototyping

Early on, you may not know what cases you need in datatypes and functions

- But static typing disallows code without having all cases; dynamic lets incomplete programs run
- So you make premature commitments to data structures
- And end up writing code to appease the type-checker that you later throw away
  - Particularly frustrating while prototyping

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?

- New, evolving code most likely to make inconsistent assumptions

Easy to put in temporary stubs as necessary, such as

```
| _ => 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 Racket callers can be *oblivious*

```
(define (f x) (* 2 x))
```

```
(define (f x)
  (if (number? x)
      (* 2 x)
      (string-append x x)))
```

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

```
fun f x =
  case f x of
     Int i    => Int (2 * i)
   | String s => String(s ^
   s)
```

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: cannot test part-way through

Coda

- Static vs. dynamic typing is too coarse a question
  - Better question: *What* should we enforce statically?

- Legitimate trade-offs you should know
  - Rational discussion informed by facts!

- Ideal (?): Flexible languages allowing best-of-both-worlds?
  - Would programmers use such flexibility well? Who decides?