CSE341: Programming Languages

Lecture 18
Static vs. Dynamic Typing

Dan Grossman
Spring 2013

# *Key differences*

- Racket and ML have *much* in common
- Key differences
  - Syntax
  - Pattern-matching vs. struct-tests and accessor-functions
  - Semantics of various let-expressions
  - …
- *Biggest* difference: ML's type system and Racket's lack thereof *

* There is Typed Racket, which interacts well with Racket so you can have typed and untyped modules, but we won't study it, and it differs in interesting ways from ML



# *Upcoming topics*

Coming soon:
- What is type-checking? Static typing? Dynamic typing? Etc.
- Why is type-checking approximate?
- What are the advantages and disadvantages of type-checking?

But first to better appreciate ML and Racket:
- How could a Racket programmer describe ML?
- How could an ML programmer describe Racket?



# *ML from a Racket perspective*

- Syntax, etc. aside, ML is like a well-defined subset of Racket
- Many of the programs it disallows have bugs ☺

```
(define (g x) (+ x x)) ; ok
(define (f y) (+ y (car y)))
(define (h z) (g (cons z 2)))
```

  - In fact, in what ML allows, I never need primitives like **`number?`**
- But other programs it disallows I may actually want to write ☹

```
(define (f x) (if (> x 0) #t (list 1 2)))
(define xs (list 1 #t "hi"))
(define y (f (car xs)))
```



# *Racket from an ML Perspective*

One way to describe Racket is that it has "one big datatype"
- *All* values have this type

```
datatype theType = Int of int  | String of string
                 | Pair of theType * theType
                 | Fun of theType -> theType
                 | …
```

- Constructors are applied implicitly (values are *tagged*)
  - **`42`** is really like **`Int 42`** | inttag | 42 |
- Primitives implicitly *check tags and extract data*, raising errors for wrong constructors

```
fun car v = case v of Pair(a,b) => a | _  => raise …
fun pair? v = case v of Pair _ => true | _  => false
```



# *More on The One Type*

- Built-in constructors for "theType": numbers, strings, booleans, pairs, symbols, procedures, etc.
- Each struct-definition creates a *new constructor*, dynamically adding to "theType"

## *Static checking*

- *Static checking* is anything done to reject a program *after* it (successfully) parses but *before* it runs
- ***Part of a PL's definition: what static checking is performed***
  – A "helpful tool" could do more checking
- 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"`), enforcing abstraction, and avoiding dynamic checking
    - Dynamic means at run-time
- Dynamically-typed languages do (almost) no static checking
  – Line is not absolute



## *Example: ML, what types prevent*

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

- A primitive operation used on a value of the wrong type
  – Arithmetic on a non-number
  – `e1 e2` where `e1` does not evaluate to a function
  – A non-boolean between `if` and `then`
- A variable not defined in the environment
- A pattern-match with a redundant pattern
- Code outside a module call a function not in the module's signature
- …

(First two are "standard" for type systems, but different languages' type systems ensure different things)



## *Example: ML, what types allow*

In ML, type-checking does ***not*** prevent any of these errors
  – Instead, detected at run-time

- Calling functions such that exceptions occur, e.g., `hd []`
- An array-bounds error
- Division-by-zero

In general, no type system prevents logic / algorithmic errors:
- Reversing the branches of a conditional
- Calling `f` instead of `g`

(Without a program specification, type-checker can't "read minds")



## *Purpose is to prevent something*

Have discussed facts about *what* the ML type system does and does not prevent
  – Separate from *how* (e.g., one type for each variable) though previously studied many of ML's typing rules

Language design includes deciding *what* is checked and *how*

Hard part is making sure the type system "achieves its purpose"
  – That "the how" accomplishes "the what"
  – More precise definition next



## *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.0` instead
    - Just like `3.0 / 0.0` does in every (?) PL (it's useful!)



## *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" like generics aimed at "fewer false positives"

Notice soundness/completeness is 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")
```



# Why incompleteness

- Almost anything you might like to check statically is undecidable:
  - Any static checker *cannot* do all of: (1) always terminate, (2) be sound, (3) be complete
  - This is a mathematical theorem!
- Examples:
  - Will this function terminate on some 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 an essential concept at the core of computing
  - The inherent approximation of static checking is probably its most important ramification



# What about unsoundness?

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

- Fix it with an updated language definition?
- Insert dynamic checks as needed to prevent X from happening?
- Just allow X to happen even if "tried to stop it"?
- Worse: Allow not just X, but *anything* to happen if "programmer gets something wrong"
  - Will discuss C and C++ next…



# Why weak typing (C/C++)

Weak typing: There exist programs that, by definition, *must* pass static checking but then when run can "set the computer on fire"?

- Dynamic checking is optional and in practice not done
- Why might anything happen?

- 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 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 (fewer false positives)
- 1 bug in a 30-million line operating system written in C can make an entire computer vulnerable
  - An important bug like this was probably announced this week (because there is one almost every week)



# Example: Racket

- Racket is ***not*** weakly typed
  - It just checks most things dynamically*
  - Dynamic checking is the *definition* – if the *implementation* can analyze the code to ensure some checks are not needed, then it can *optimize them away*
- Not having ML or Java's rules can be convenient
  - Cons cells can build anything
  - Anything except `#f` is true
  - …

  This is nothing like the "catch-fire semantics" of weak typing

*Checks macro usage and undefined-variables in modules statically

## Another misconception

What operations are primitives defined on and when an error?

- Example: Is `"foo" + "bar"` allowed?
- Example: Is `"foo" + 3` allowed?
- Example: Is `arr[10]` allowed if `arr` has only 5 elements?
- Example: Can you call a function with too few or too many arguments?

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 versus maybe being more convenient

Racket generally less lenient on these things than, e.g., Ruby



## Now can argue…

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

Remember most languages do some of each

- For example, perhaps types for primitives are checked statically, but array bounds are not



## Claim 1a: Dynamic is more convenient

Dynamic typing lets you build a heterogeneous list or return a "number or a string" without workarounds

```
(define (f y)
  (if (> y 0) (+ y y) "hi"))

(let ([ans (f x)])
  (if (number? ans) (number->string ans) ans))
```

```
datatype t = Int of int | String of string
fun f y = if y > 0 then Int(y+y) else String "hi"

case 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

```
(define (cube x)
  (if (not (number? x))
      (error "bad arguments")
      (* x x x)))

(cube 7)
```

```
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 programmers to code around limitations

```
(define (f g)
  (cons (g 7) (g #t)))

(define pair_of_pairs
  (f (lambda (x) (cons x x))))
```

```
fun f g = (g 7, g true) (* does not type-check *)

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



## Claim 2b: Static lets you tag as needed

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

In the extreme, can use "TheOneRacketType" in ML

- Extreme rarely needed in practice

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

if e1
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 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

```
(define (pow x) ; curried
  (lambda (y)
    (if (= y 0)
        1
        (* x (pow x (- y 1)))))) ; oops
```

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



## Claim 3b: 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

```
(define (pow x) ; curried
  (lambda (y)
    (if (= y 0)
        1
        (+ x ((pow x) (- y 1)))))) ; oops
```

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



## 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?