ML vs. Racket and Static vs. Dynamic Type-Checking

Examples adapted from Dan Grossman

ML vs. Racket

Key differences

- syntax
  - datatypes/pattern-matching vs. features not studied
  - let, let*, letrec
  - eval
    - ... static type system vs. dynamic contracts*

* Typed Racket supports typed modules, interesting differences with ML

ML from a Racket perspective

A well-defined subset of Racket

Many Racket programs rejected by ML have bugs.

```lisp
(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, never need primitives like `number`?

Other Racket programs rejected by ML would work.

```lisp
(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

Racket has "one big datatype" for all values.

```lisp
datatype theType = Int of int | String of string | Cons of theType * theType | Func of theType -> theType | ...
```

Constructors applied implicitly (values are tagged)

42 is really like `Int 42`

```lisp
fun car v = case v of
  Pair(s,b) => a
| _ => raise TypeError
fun pair? v = case v of Pair _ => true | _ => false
```
Static checking

May reject a program after parsing, before running.

Part of a PL definition: what static checking is performed?

Common form: static type system

Approach: give each variable, expression, ..., a type

Purposes:
- Prevent misuse of primitives (4/"hi")
- Enforce abstraction
- Avoid cost of dynamic (run-time) checks
- Document intent

Dynamically-typed languages = little/no static checking

Example: ML type-checking

Catches at compile-time: ...
- Operation used on a value of wrong type
- Variable not defined in the environment
- Pattern-match with a redundant pattern

Catches only at run-time: ...
- Array-bounds errors, Division-by-zero, explicit exceptions zip ((1,2), ["a"])
- Logic / algorithmic errors:
  - Reversing the branches of a conditional
  - Calling f instead of g
  (Type-checker can’t “read minds”)

Purpose: prevent certain kinds of bugs. But when / how well?

“Catch a bug before it matters.”

vs.

“Don’t report a (non-)bug that might not matter”

Prevent evaluating 3 / 0

- Keystroke time: disallow it in the editor
- Compile time: disallow it if seen in code
- Link time: disallow it in code attached to main
- Run time: disallow it right when we get to the division
- Later: Instead of doing the division, return +inf.0
  - Just like 3.0 / 0.0 does in every (?) PL (it’s useful)

Correctness

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 / no missed X bugs

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 / no false X bugs

Usual goal: sound (can rely on it) but not complete (why not?)

“Fancy features” like generics aimed at “fewer false positives”

Notice soundness/completeness is with respect to X.
Incompleteness

ML rejects these functions even though they never divide by a string.

```plaintext
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 y = f3 true
fun f4 x = if x <= abs x then 0 else 4 div "hi"
fun f5 x = 4 div x
val z = f5 (if true then 1 else "hi")
```

What if it's unsound?

- **Oops**: fix the language definition.
- **Hybrid checking**: add dynamic checks to catch X at run time.
- **Weak typing**: "best" effort, but X could still happen.
- **Catch-fire semantics**: allow anything (not just X) to happen if program could do X.
  - Simplify implementer's job at cost of programmability.
  - Assume correctness, avoid costs of checking, optimize.

Weak typing -> weak software

- An outdated sentiment: "strong types for weak minds"
  - "Humans will always be smarter than a type system (cf. undecidability), so need to let them say trust me."
- Closer to reality: "strong types amplify/protect strong minds"?
  - Humans really bad at avoiding bugs, need all the help we can get!
    - Type systems have gotten much more expressive (fewer false positives)
- 1 bug in 30-million line OS in C makes entire computer vulnerable.
  - Bug like this was announced this week (every week)

Racket: dynamic, not weak!

- Dynamic checking is the definition
  - If implementation proves some checks unneeded, it may optimize them away
- Convenient
  - Cons cells can build anything
  - Anything except #f is true
  - Nothing like the "catch-fire semantics" of weak typing
Don't confuse semantic choices and checking.

- Is this allowed? What does it mean?
  - "foo" + "bar"
  - "foo" + 3
  - array[10] when array has only 5 elements
  - Call a function with missing/extra arguments

Not an issue of static vs. dynamic vs. weak checking.
- But does involve trade off convenience vs. catching bugs early.

Racket generally less lenient than, JavaScript, Ruby, ...


Most languages do some of each
- Common: types for primitives checked statically; array bounds are not.

Consider:
- Flexibility
- Convenience
- Catch bugs
- Speed (run-time, programming-time, debugging-time, fixing-time)
- Reuse
- Documentation value
- Prototyping
- Evolution/maintenance
- Cognitive load (satisfying compiler, debugging at run-time)
- ...

Convenience: Dynamic is more convenient

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

```scheme
(define (f y)
  (if (> y 0) (+ y y) "hi"))
(let ([ans (f x)])
  (if (number? ans) (number->string ans) ans))
```

Convenience: 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

```scheme
(define (cube x)
  (if (not (number? x))
    (error "bad arguments")
    (* x x x)))(cube 7)
```

```scheme
fun cube x = x * x * x
```

cube 7
Expressiveness: Static prevents useful programs

Any sound static type system forbids programs that do nothing wrong, possibly 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
\[
f \ g = (g \ 7, \ g \ true) \quad (* \text{might not type-check} *)
\]

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

Expressiveness: Static lets you tag as needed

Pay costs of tagging (time, space, late errors) only where needed, rather than on everything, everywhere, all the time.

Common: a few cases needed in a few spots.
Extreme: "TheOneRacketType" in ML, everything everywhere.

```
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")
```

Bugs: Static catches bugs earlier

Lean on type-checker for compile-time bug-catching, do less testing.

```
(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)
```

Bugs: 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 *)
```
Efficiency: Static typing is faster

Language implementation:
• Need not store tags (space, time)
• Need not check tags (time)

Your code:
• Need not check argument and result types.
  (Convenience, Expressiveness, Bugs)

Your effort:
• Need not spend time writing checks or debugging type issues later.
  (Bugs)

Efficiency: Dynamic typing is faster

Language implementation:
• May optimize to remove some unnecessary tags and tests
  • Example: `(let ([x (+ y y)] (* x 4)))`
• Hard (impossible) in general
• Often easier for performance-critical parts of program
• Can be surprisingly effective

Your code:
• Need not "code around" type-system limits with extra tags, functions
  (Convenience, Expressiveness)

Your effort:
• Need not spend time satisfying type checker now.
  (Convenience, Expressiveness)

Reuse: Code reuse easier with dynamic

Reuse code on different data flexibly without restrictive type system.

• If you use cons cells for everything, libraries that work on cons cells are useful

• Collections libraries are amazingly useful, may have complicated static types

• Use code based on what it actually does, not just what it says it can do, for flexible code reuse.

Reuse: Code reuse easier with static

• Modern type systems 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

• Enforce clean abstractions and invariants for safe/reliable code reuse.
  • Also possible with dynamic types, less common, often involves at least a small static component.
But software evolves.

Considered 5 things important when writing code:
1. Convenience
2. Not preventing useful programs
3. Catching bugs early
4. Performance
5. Code reuse

What about:
• Prototyping before a spec is stable
• Maintenance / evolution after initial release

Prototyping: Dynamic better for prototyping

Early on, may not know what cases needed in datatypes and functions
• Static typing disallows code without having all cases
• Dynamic lets incomplete programs run
• Static forces premature commitments to data structures
• Waste time appeasing the type-checker when you will just change it/throw it away soon anyway

Prototyping: 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

Temporary stubs as necessary, such as
   \_ \_ \_=> raise Unimplemented
   but don’t forget them!

Evolution: 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 now use constructor on arguments, pattern-match results.
• Existing Racket callers can be oblivious

Counter-argument: Quick patches and hacks leave bloated, confusing code. Easy to make deeper change that accidentally breaks callers.
Evolution: Static better for evolution

When changing types of data or code, type-checker errors provide a to-do list of necessary changes.
- 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

Examples:
- Change the return type of a function
- Add a new constructor to a datatype

Counter-argument:
- The to-do list is mandatory, so evolution in pieces is a pain.
- Cannot test part-way through.

Resolved?

Static vs. dynamic typing is too coarse a question.
- Better: What should we enforce statically? Dynamically?
- My research area:Concurrency/parallelism need more of both!

Legitimate trade-offs, not all-or-nothing.

Beyond...

More expressive static type systems that allow more safe behaviors (without more unsafe behaviors).
- **Dependent typing** (long-running, active research field)
- Starting to see wider adoption
- Concurrency, network activity, security, data privacy
- Strong, fine-grain guarantees

SML type checker: pattern-matching inexhaustive.

```
fun nth 0 (x::xs) = x
   | nth n (x::xs) = nth (n-1) xs
```

```
nth : int -> 'a list -> 'a
```

Dependent types would allow:

```
nth : (n:int, n>=0) -> (xs:'a list, length xs >= n) -> 'a
```

Or maybe even:

```
x = (ys @ (r::zs)), length ys = n
```

Beyond...

Types are much more.
Curry-Howard correspondence: **Proofs are Programs!**

<table>
<thead>
<tr>
<th>Logic</th>
<th>Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositions</td>
<td>Types</td>
</tr>
<tr>
<td>Proposition $P \rightarrow Q$</td>
<td>Type $P \rightarrow Q$</td>
</tr>
<tr>
<td>Proposition $P \land Q$</td>
<td>Type $P \land Q$</td>
</tr>
<tr>
<td>Proof of proposition $P$</td>
<td>Expression $e : P$</td>
</tr>
<tr>
<td>Proposition $P$ is provable</td>
<td>$\exists$ expression $e : P$</td>
</tr>
</tbody>
</table>

What then is 'a in logic?

Table adapted from Pierce, Types and Programming Languages, an excellent read if this direction inspires you.