CSE 341: Programming Languages

Hal Perkins

Spring 2011

Lecture 17— Static vs. dynamic typing

Today

Consider one of the biggest differences between Scheme and ML:

- ML is statically typed (many errors when compiled)
- Scheme is dynamically typed (many errors when run)

More generally:

- Why is static typing good/bad?
- How do you judge a type system?

Strong typing vs. Weak typing

In languages with weak typing, there exist programs that implementations *must* accept at compile-time, but at run-time the program can do *anything*, including blow-up your computer.

- Examples: C, C++

Old wisdom: Strong types for weak minds

New wisdom: Weak typing endangers society & costs > $1e10/year

Why weak typing? For efficiency and low-level implementation (important for small parts of low-level systems)

Editorial opinion: Programming is hard enough without implementation-defined behavior. This has little to do with types:

- ML, Scheme, Java, Ruby all "strongly typed" in this sense

Static Typing vs. Dynamic Typing

In ML and Scheme `"hi" - "mom"` or `(- "hi" "mom")` are errors.

- In ML it's "at compile-time" (static)
- In Scheme it's "at run-time" (dynamic)

`(define (f) (- "hi" "mom"))` fine *until you call it*, but never type-checks in ML.

This also never type-checks in ML, but may never fail *if called appropriately*:

```
(define (f g x y)
  (if (g x)
      (string-length y)
      (+ y 1)))
```

```
fun f (g,x,y) =
  if g x
  then String.size y
  else y + 1 (* type-error! *)
```

Basic benefits/limitations

Indisputable facts:

- A language with static checks catches certain bugs without testing (earlier in the software-development cycle)
- It's impossible to catch exactly the buggy programs at compile-time
  - *Impossible* (undecidable) to know what code will execute in what environments, so may give *false positives*
  - *Impossible* to know exactly what types a function argument might have without running the program, so may give *false positives*
  - Algorithm bugs remain (e.g., using + where you meant −)

Static Checking

Key questions for a compile-time check (e.g., ML type-checking):

1. What is it checking? Examples (and not):
   - Yes: Primitives (e.g., +) aren't applied to inappropriate values
   - Yes: Module interfaces are respected
     (e.g., don't use private functions)
   - Yes: Patterns are not redundant
   - No: `hd` is never applied to the empty list
   - No: Array indices are in bounds

   Knowing what is caught for me affects how I program.

2. Is it *sound*? (Does it ever accept a program that at run-time does what we claimed it could not? "false negative")
3. Is it *complete*? (Does it ever reject a program that could not do the "bad thing" at run-time? "false positive")

Unfortunately...

All non-trivial static analyses are either unsound or incomplete.

- Direct corollary to CSE 311 concept of undecidability

Good design leads to "useful subsets" of all programs, typically (but not always) ensuring soundness and sacrificing completeness.

- Forbid all programs that do some "bad" things
  (like pass a function to +)
- Also forbid some programs that don't do the bad things because we can't tell

To judge a type system:

- Is it sound (or is it "broken")?
- Is it "expressive enough" (is the incompleteness palatable)?

A Question of Eagerness

Again, every static type system provides certain guarantees. Some things we might want to check statically (soundly but incompletely), but ML and Java's type system don't: no null-pointer exceptions, no division-by-zero, no data races, ...

There is also more than "compile-time" or "run-time".

Consider `3 / 0`

- Compile-time: reject if code is "reachable" (maybe dead branch)
- Link-time: reject if code is "reachable" (maybe unused function)
- Run-time: reject if code executes (maybe branch never taken)
- Even later: maybe delay error until "bad number" is used to index into an array or something.
  - Crazy? Floating-point allows 3.0 / 0.0; gives you `+inf.0`.

Exploring Some Arguments

1a. Dynamic typing is more convenient

```
(define (f x) (if (> x 0) (* 2 x) #f))
(let ([ans (f y)]) (if (number? ans) e1 e2))
```

```
datatype intOrBool = Int of int | Bool of bool
fun f x = if x > 0 then Int (2*x) else Bool false
case f y of
  Int ans => e1
| Bool _ => e2
```

Just return what you want; no need to define datatypes (use the-one-big-datatype)

Exploring Some Arguments

1b. Static typing is more convenient

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


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

With dynamic-typing, assuming things about arguments can lead to errors far from the logical mistake
("expected foo got bar" deep in some library)

Exploring Some Arguments

2. Static typing prevents / doesn't prevent useful programs

- Overly restrictive type systems certainly can (e.g., without polymorphism a new list library for each list-element type)
- `datatype` gives you as much or as little flexibility as you want – can embed Scheme in ML:

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

```
if e1
then Fun (fn x => case x of Int i => Int (i * i * i))
else Cons (Int 7, String "hi")
```

Viewed this way, Scheme is "unityped" with "implicit tag-checking" which is "just" a matter of convenience.

Exploring Some Arguments

3. Static/dynamic typing better for code evolution

Change:

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

to:

```
datatype t = I of int
           | S of string
fun f x =                       (define (f x)
  case x of                       (if (number? x)
    I i => I (i * 2)                  (* x 2)
  | S s => S (s ^ s)                  (string-append x x)))
```

- Good example for dynamic: In ML, all callers must change
- But: If we change the return type of `f`, ML type-checker will give us a full to-do list of what to change.

Another evolution example

Suppose I add a new constructor to an ML datatype
(like a `Mult` for arithmetic expressions)

- Most existing patterns over the type will now give a warning
  - Good reason not to use _ patterns
- But if I "know" some expressions will not be multiplies, then these warnings are false positives

Exploring Some Arguments

4. Types make code reuse harder/easier

- Dynamic:
  - Sound types means you'll always restrict how code is used in some way that you need not
  - By using cons cells for everything, you can reuse lots of libraries
- Static:
  - Using separate types catches bugs and enforces abstractions (don't accidentally confuse two different uses of cons cells)
  - We can provide enough flexibility in practice (e.g., with polymorphism)

Design issue: Whether to build a new data structure or encode with existing ones (for libraries) is an important consideration

Exploring Some Arguments

5. Types make programs faster/slower.

- Dynamic: Don't have to code around the type system or duplicate code; optimizer can remove provably unnecessary tag-tests
- Static: Programmer controls where tag-tests occur (in patterns) and knows that compiler need not have unnecessary tests (is argument to + a number).

Summary

There are real trade-offs here; you must know them.

We can have rational discussions about them, informed by facts.

Almost every language checks some things statically and other things dynamically.

- It's really a question of *what* you check statically, but we have an informal sense of what type-checking "normally checks for"