CS 3110

Lecture 14: Static vs. dynamic checking

Prof. Clarkson

Fall 2014

Today's music: "Let's talk about sex" (Power Mix Instrumental) by Salt-n-Pepa

SPOILER: We won't.

Review

Course so far:

- Introduction to functional programming
- Modular programming

Next two weeks:
Advanced topics in functional programming

- Dynamically-typed functional languages
- Concurrency (Guest lectures: Prof. Foster)
- Dependent types (Guest lecture: Prof. Constable)

Question #1

How much of PS4 have you finished?

A. None
B. About 25%
C. About 50%
D. About 75%
E. I'm done!!!

PS4

Scheme3110:

- A new language, based on R5RS Scheme, a language from the LISP family
- Like an "untyped" version of OCaml, but don't confuse them

John McCarthy

(1927-2011)

Turing Award Winner (1971)

- Invented the phrase "artificial intelligence"
- Helped design ALGOL, a language that influenced all modern imperative languages, and was the first language with nested functions and lexical scope
- Designed LISP (LISt Processing) language, a language that influenced all modern functional languages; introduced garbage collection

PS4

Scheme3110 interpreter:

- First, learn the language
- Then, design the interpreter based on starter code
- Only then should you start to code
- Problem is structured in stages
  - If you run out of time, complete early stages before later stages
  - Build a test suite that tests each stage; run regression tests

- Start now!

Let's talk about "untyped" languages...

- Better name: "dynamically typed"
- Type checking done dynamically, i.e., at run time
- As opposed to statically, i.e., at compile time

Static vs. dynamic checking

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

- First need to understand the facts:
  - *What* static checking means
  - *When* static checking is done (and *where it* falls on a continuum)
  - *Why* static checking is used
  - *How much* you can expect out of static checking

Question #2

Have you ever been involved in an argument about typed-vs.-untyped (i.e., static vs. dynamically checked) languages?

A. Yes

B. No

FACTS ABOUT
STATIC VS DYNAMIC CHECKING

WHAT: 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 (Lint, FindBugs, etc.) can go beyond that definition

Type checking

Type checking is a kind of static checking

- *Approach* is to give a type to each variable, expression, etc.
- *Purposes* include preventing misuse of primitives (e.g., **`4/"hi"`**) and avoiding run-time checking
- Dynamically-typed PLs (e.g., Python, JavaScript, Scheme) do much less type checking
  - Maybe none, but the line is fuzzy and depends on exactly what one means by "type checking"...

Question #3

Which is an example of static checking?

A. Checking that only 7-bit ASCII characters appear in the source code
B. Checking that every left paren is matched by a right paren
C. Checking that all return values of a function have the same type
D. Checking that all pattern matches are exhaustive
E. Checking that the program never causes a division by zero error

Question #3

Which is an example of static checking?

A. Checking that only 7-bit ASCII characters appear in the source code // done before parsing

B. Checking that every left paren is matched by a right paren // done during parsing

C. Checking that all return values of a function have the same type

D. Checking that all pattern matches are exhaustive

E. Checking that the program never causes a division by zero error // has to be done at run-time

Note: to some extent, depends on definition of "compile time" and "run time"

WHEN: A question of eagerness

- Static checking & dynamic checking are two points (or maybe two intervals) 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 OCaml, and in every PL that implements IEEE floating point standard

"Catching a bug before it matters"

is in inherent tension with

"Don't report a bug that might not matter"

WHY: Purpose is prevention

Different languages prevent different things:

- **Java:** prevents
  - casting to types other than supertypes or subtypes
  - missing field and method errors
  - accessing private fields, etc.
- **OCaml:** prevents
  - inexhaustive pattern matches, 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…

Example: OCaml

OCaml static checking **prevents** these errors from ever occurring at run-time:

- Using arithmetic on a non-number
- Trying to evaluate a function application `e1 e2` where `e1` does not evaluate to a function
- Having a non-Boolean between `if` and `then`
- Using a variable that is not in the environment
- Having a pattern-match with a redundant pattern
- …

These are all standard goals for statically typed (functional) languages

Example: OCaml

OCaml static checking **does not prevent** these errors from ever occurring at run-time:

- Exceptions (e.g., **`hd []`**)
- An array-bounds error
- Division-by-zero

Instead, these are *detected* at run-time

HOW MUCH:
Could all errors be prevented?

E.g., logic or algorithmic errors:

- Reversing the branches of a conditional
- Using **+** instead of **–**

Without a program specification, static checker can’t “read your mind”

- Dependent types prevent more errors (next week)
- Program verification prevents even more errors (next month)
- Provably impossible to prevent all errors (cf. *undecidability* in CS 4810)

...so static checking doesn't attempt to prevent all errors
...what would it mean for static checker to be **correct**?

Analogy: True and false positives

	Test says you don't have disease	Test says you do have disease
You really don't have disease	True negative	**False positive**
You really do have disease	**False negative**	True positive

- **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 static checking:

- *Test* is the static checker.
- *Patient* is the program.
- *Disease* is "doing bad thing X."
- *Correctness* is minimizing false positives and false negatives (CS 4110)

Alternative: Dynamic checking

- If false positives and false negatives are a given, maybe we should give up on static checking?
- Not having to obey OCaml or Java's typing rules can be convenient
  - Maybe arrays can hold anything
  - Maybe everything is `true` except `false` and `[]`
  - Maybe don't need to create a datatype just to pass different types of data to a function
  - ...
- Basis of many modern "scripting" languages, e.g., Python, Ruby, etc.

Example: Scheme3110

In OCaml, we might complain about false positives at compile time.

```
let f = fun y ->
  if true then 0 else (4 + true)
```

In Scheme3110, we might complain about not catching obvious errors at compile time*

```
(define f (lambda (y) (+ 4 #t)))
```

*although, you're building an interpreter, not a compiler

DISCUSSION OF STATIC VS DYNAMIC CHECKING

Static vs. dynamic checking

- We've stated a bunch of facts about static and dynamic checking
- Let's rationally consider arguments about which is better

Remember: it's a spectrum; most languages do some of each

- Examples in OCaml and Scheme3110
  - Extensions to Scheme3110: **`number?`**, **`>`**, **`-`**
  - All Scheme3110 examples here are valid R5RS Scheme

Question #4

Make a quick list of the main languages which you've programmed in, and how much code you've written in them.

A. I've programmed more in statically-typed languages.

B. I've programmed more in dynamically-typed languages.

Claim 1a: Dynamic is more convenient

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

```
(define f (lambda (y)
            (if (> y 0) (+ y y) #t)))
```

```
type t = Int of int | Bool of bool (* "tags" *)

let f y = if y > 0 then Int(y+y) else Bool true
```

Claim 1b: Static is more convenient

Can assume data has the expected type without cluttering code with dynamic checks or having run-time errors (possibly far from the logical mistake)

```
(define cube (lambda (x) (* x x x)))
(cube #t) ; run-time error
```

```
(define cube (lambda (x)
   (if (number? x) (* x x x) #f)))
(cube #t) ; --> #f
```

```
let cube x = x * x * x

cube true (* doesn't type-check *)
```

Claim 2a: Static prevents useful programs

Static type systems forbids programs that do nothing wrong, forcing the programmer to "code around" the limitation

```
let f g = (g 7, g true) (* doesn't type-check *)
f (fun x -> (x,x))
```

```
type tost = (* The One Scheme Type *)
   | Int of int
   | Bool of bool
   | Cons of tost * tost
   | Fun of (tost -> tost)

let f g = Cons(g (Int 7), g (Bool true))
f (fun x -> Cons(x,x)) (* does type-check *)
```

Claim 2b: Dynamic allows useful programs

But at the cost of tagging everything at run-time

```
(define f (lambda (g)
   (cons (g 7) (g #t))))
(f (lambda (x) (cons x x)))
; --> ((7 . 7) #t . #t)

(number? 7)
; --> #t
```

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 unit test for them

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

```
(define pow (lambda (x y)
  (if (equal? y 0) 1
    (* x (pow
      (cons x (- y 1))))))) ; run-time error
```

Claim 3b: Static catches only easy bugs

So you still have to unit test your functions, thus finding "easy" bugs, too

```
(define wrong-pow (lambda (x y)
  (if (equal? y 0) 1
    (+ x (wrong-pow x (- y 1))))))
```

```
let rec wrong_pow x y =
  if y = 0
  then 1
  else x + wrong_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
- Although hard (or 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 that lead to extra tagging

Claim 5a: Code reuse easier with dynamic

By not requiring types, tags, etc., more code can just be reused with data of different types

- e.g., great libraries for working with lists/arrays in languages like Python and Ruby
- whereas Java and OCaml collections libraries are often have very complicated static types

Claim 5b: Code reuse easier with static

- If you use arrays to represent everything, you will
  - confuse abstraction functions and
  - get hard-to-debug errors

- Use separate static types to keep data abstractions separate
  - Static types help avoid library misuse
  - Modern type systems support code reuse with features like generics and subtyping

So far

Considered 5 things you care about when writing code:

1. Convenience
2. Writing 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 fully know
  - what operations you need in data abstractions
  - what constructors you need in datatypes
  - what branches you need in pattern matches, etc.
- So dynamic is a win!
  - Dynamic lets "incomplete" programs run
- Static typing loses because
  - won't let you "try out" code without having all cases
  - you make premature commitments to data structures
  - you write a lot of code to appease the type-checker
    - code that you end up throwing away

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

```
| _ => 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 function instead of an **`int`**

```
let twice x = 2 * x
```

```
(* evolved *)
let twice x =
  match x with
     Int i > Int (2 * i)
   | Fun f -> Fun(fun y -> f (f y))
```

Claim 7a: Dynamic better for evolution

- Can change code to be more permissive without affecting old callers
  - Example: Take an `int` or a function instead of an `int`

```
(define twice
  (lambda (x) (* 2 x)))
```

```
; evolved
(define twice (lambda (x)
  (if (number? x) (* 2 x)
    (lambda (y) (x (x y))))))
```

Claim 7a: Dynamic better for evolution

Can change code to be more permissive without affecting old callers

- Example: Take an `int` or a function instead of an `int`
- All OCaml clients must now use a constructor on arguments and pattern-match on results
- Existing Scheme3110 callers can be *oblivious*

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

Examples:

- Changing the return type of a function
- Adding a new constructor to a data type
  - 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"

Conclusion:
Static vs. dynamic checking

- **Controversial topic!**
- There are real trade-offs here you should know
  - Enables rational discussion informed by facts
- Simply debating "static vs. dynamic typing" isn't useful
  - It's a continuum; most languages have examples of both ends
  - "What should we enforce statically" makes more sense to debate
- You get to experience examples of each:
  - **Imperative:** Python vs. Java
  - **Functional:** OCaml vs Scheme3110
- Ideally would have flexible languages that allow best-of-both-worlds
  - Still an open and active area of research!

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Prelim 1

- If you have questions about what you missed, talk to consultants or TAs
- If **after that** you want to talk, come to my office hours
- I will hold an extra office hour today to accommodate extra demand
  - Office hours today: 2-4 pm

Upcoming events

- **PS4 is out, due in one week**

This is controversial.

THIS IS 3110

APPENDIX: SOUNDNESS AND COMPLETENESS

Correctness

Suppose a static checker is supposed to prevent X

- A static checker is sound if it never accepts a program that, when run with some input, does X
  - No false negatives: test never ignores disease
- A static checker is complete if it never rejects a program that, no matter what input it is run with, will not do X
  - No false positives: test never needlessly scares you

Usual goal in designing static checker is to be:

- sound (so you can rely on it) but
- not complete

True and false positives

	Type system accepts program	**Type system rejects program**
Program doesn't do X	True negative	**False positive**
Program does do X	**False negative**	True positive

- Soundness = no false negatives
- Completeness = no false positives

Soundness and completeness are always with respect to a bad thing X

Incompleteness

OCaml rejects even though never divide by a string:

```
let f1 x = 4 / "hi"  (* but f1 never called *)

let f2 x = if true then 0 else 4 / "hi"

let f3 x = if x then 0 else 4 / "hi"
let x = f3 true

let f4 x = if x <= abs x then 0 else 4 / "hi"

let f5 x = 4 / x
let y = f5 (if true then 1 else "hi")
```

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

Why incompleteness?

- Almost anything "interesting" you might like to check statically is undecidable: can't write an algorithm that always gives correct answer
  - Will this function terminate on some input? On any input?
  - Will this function ever read or write a variable not in the environment?
  - Will this function divide an integer by a string?
  - Will this function divide by zero?

- Undecidability is discussed in CS 4810
  - The inherent approximation of static checking is probably its most important consequence
  - Can never build a static checker that always terminates, never has false positives, and never has false negatives.
  - Must give up on one of those three

Why not give up on unsoundness?

We could! Static checker would have false negatives.

- Program really does bad thing X but static checker says it doesn't

Two further choices:

1. Make behavior unspecified when X happens?
   - including deleting your files, emailing your credit card number, setting the computer on fire...
   - Languages where this is the norm are called *weakly typed* (as opposed to *strongly typed*)
   - Weak typing is a poor name: Really about doing neither static *nor* dynamic checks
   - Best-known example of weak typing: C and C++
2. Insert run-time checks as needed to prevent X from happening?
   - Many statically-typed languages do this when X is hard or impossible to detect statically
     - E.g., Java ClassCastException, ArrayStoreException, etc.
   - So many static checkers are neither sound nor complete (but they're still useful)
   - Or, could just give up on static checking and test everything dynamically, i.e., dynamic checking

Choice 1: 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?

- 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 leads to insecurity

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

Choice 2: Dynamic checking

Not having OCaml or Java's rules can be convenient

- Maybe arrays can hold anything
- Maybe everything is **`true`** except **`false`** and **`[]`**
- Maybe don't need to create a datatype just to pass different types of data to a function
- ...

Basis of many modern "scripting" languages, e.g., Python, Ruby, etc.

Implementation can analyze the code to ascertain whether some checks aren't needed, then *optimize them away*

A different issue: Operators

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

This is not "static vs. dynamic checking."

It is "what is the run-time semantics of the primitive."

- Regardless of choice as to whether each is allowed, could build either static or dynamic checker for it