Static vs. Dynamic Typing

From Dan Grossman's CSE 341 in Spring 2023

Programming Languages
UW CSE 413 - Spring 2026

Introduction of Guest Lecturer

Stanley Yang

- Senior in CS+Math
- Upcoming BS/MS student
- Jr. SDE in Amazon
- I loves cats and board games!

Administrative Details

- Your final is on **Monday June 8, 12:30-2:20 pm in CSE2 G01**.
- Final review will be on **Friday June 5** during lecture time. Attendance is highly recommended!!
- **HW 8** is due on **Thursday June 4**. You are allowed at most 4 late days.
- No office hours next week.

OCaml vs. Racket

- A ton in common: closures, first-class functions, mutation-only-as-needed, let expressions, ...
- Some key differences
  - Syntax
  - Scoping rules and semantics of let
  - Pattern-matching vs. struct features
- Biggest difference: OCaml's static types vs. Racket's dynamic types

Much to discuss!

- Key questions
  - What *is* type checking? *Static typing*? *Dynamic typing*?
  - Why is static type checking *necessarily approximate*?
  - What are the *pros and cons of using static type checking*
- But first to gain a useful perspective:
  - How could a Racket programmer describe OCaml?
  - How could an OCaml programmer describe Racket?

OCaml from Racket Perspective

- Ignoring syntax, OCaml is like a well-behaved subset of Racket
- Many programs not in this “OCaml subset” have bugs 
  - e.g., passing a list to **`+`** or a number to **`car`**
- In the “OCaml subset”, no need for type predicates like **`number?`**
  - Answer is *always* known “at compile time” for every expression
- BUT: there are also many useful programs not in the “OCaml subset”
  - e.g. list of alternating strings and numbers
  - returning different types from different branches
  - :-(

Racket from OCaml Perspective

- One view: Racket just uses “one big variant”
  - *All* values built from this variant:
  - ```
    type the_type = Int of int | Pair of the_type * the_type
    | Fun of (the_type -> the_type) | Bool of bool | ...
    ```
- Constructors are applied *implicitly* (values are *tagged*)
  - **`42`** is really **`Int 42`**

- Primitive operations (e.g., **`+`**) check tags and extract data, raising errors for wrong constructors

```
let car v = match v with Pair(a,_) -> a | _ -> failwith …
let pair? v = Bool(match v with Pair _ -> true | _ -> false)
```
```

Static Checking

- Static checking is anything done to possibly reject a program after it (successfully) parses but *before* it runs
- Common way to describe a PL's static checking: type system
  - *Approach*: Give each expression, variable, etc. a type (or error)
  - *Purpose*: Use types to prevent some errors (e.g., adding a number to a list), enforce modularity, etc.
  - Anything made impossible due to type system need not be checked dynamically (i.e., at run-time)

Languages

- Statically typed languages use a type system to prevent various errors statically
- Dynamically typed languages do little to no static checking
- The “line” between the two can be fuzzy but usually clear enough
  - Racket “is” dynamically typed but detects undefined variables
  - OCaml “is” statically typed but allows `List.hd []`

Example: what OCaml's types prevent

- OCaml guarantees that a well-typed program (when run) will never:
  - Use a primitive operator (e.g., `+`) on a value of the wrong type
  - Try to access a variable that doesn't exist
  - Evaluate a `match` with no matching pattern
  - Violate the internal invariants of a module
  - …
- In general, different languages' type systems prevent different things
  - But many commonalities across languages

Example: what OCaml's types allow

- Plenty of other errors are still possible!
  - Calling a function that raises an exception, e.g., `List.hd []`
  - Array out of bounds errors
  - Division by zero
  - …
- There are fancy type systems that catch all of these but trickier to use
- Without a *full specification*, no type system can detect all bugs
  - If you reverse the branches of a conditional or call the wrong library function, how can the type system "know" it's wrong?

Types are designed to “prevent bad things”

- Just discussed: *what* the OCaml type system does / doesn’t prevent
- There’s also the *how*, i.e., what the actual typing rules are
- Type-system design involves:
  - Choosing what to prevent
  - Choosing how to prevent it
  - Difficult: making sure the “how” achieves its goal (the “what”)
    - See upcoming definition of *soundness*

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 if seen in code that may be used to evaluate **`main`**
- Run time: disallow it right when we get to the division
- Later: Return a special **`+inf.0`** (like floating point does!)

Correctness

Notice soundness/completeness is with respect to X

Suppose a type system is supposed to prevent X for some X

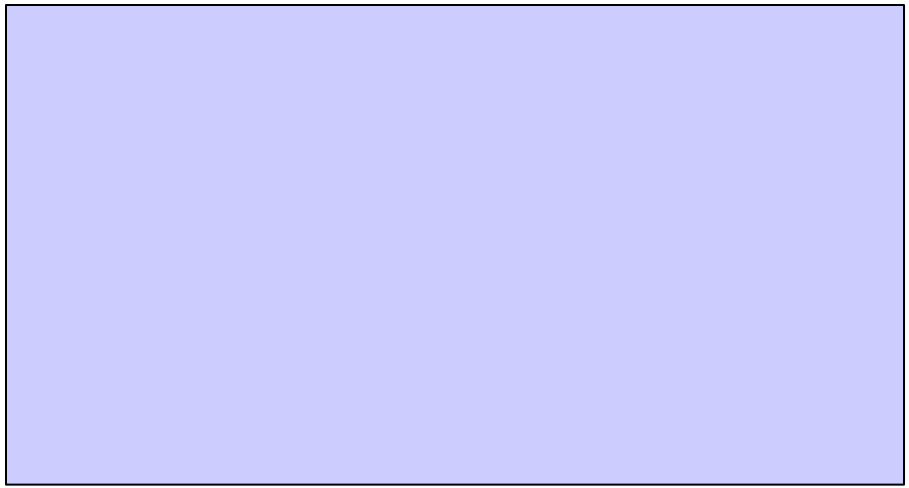
- A type system is *sound* if it never accepts a program that, when run with some input, does X
  - No *false negatives*
- A type system is *complete* if it never rejects a program that, no matter what input it is run with, will not do X
  - No *false positives*

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"

Venn Diagrams

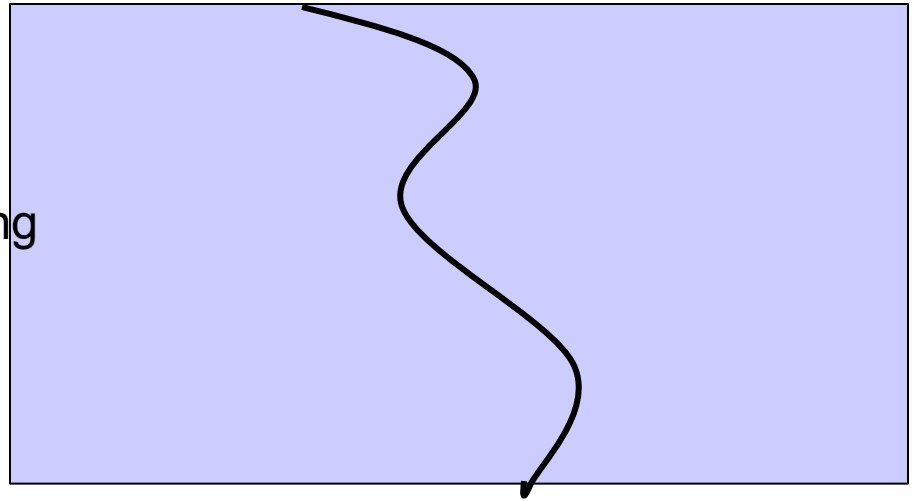
All possible programs
(in syntax of some language)



Venn Diagrams

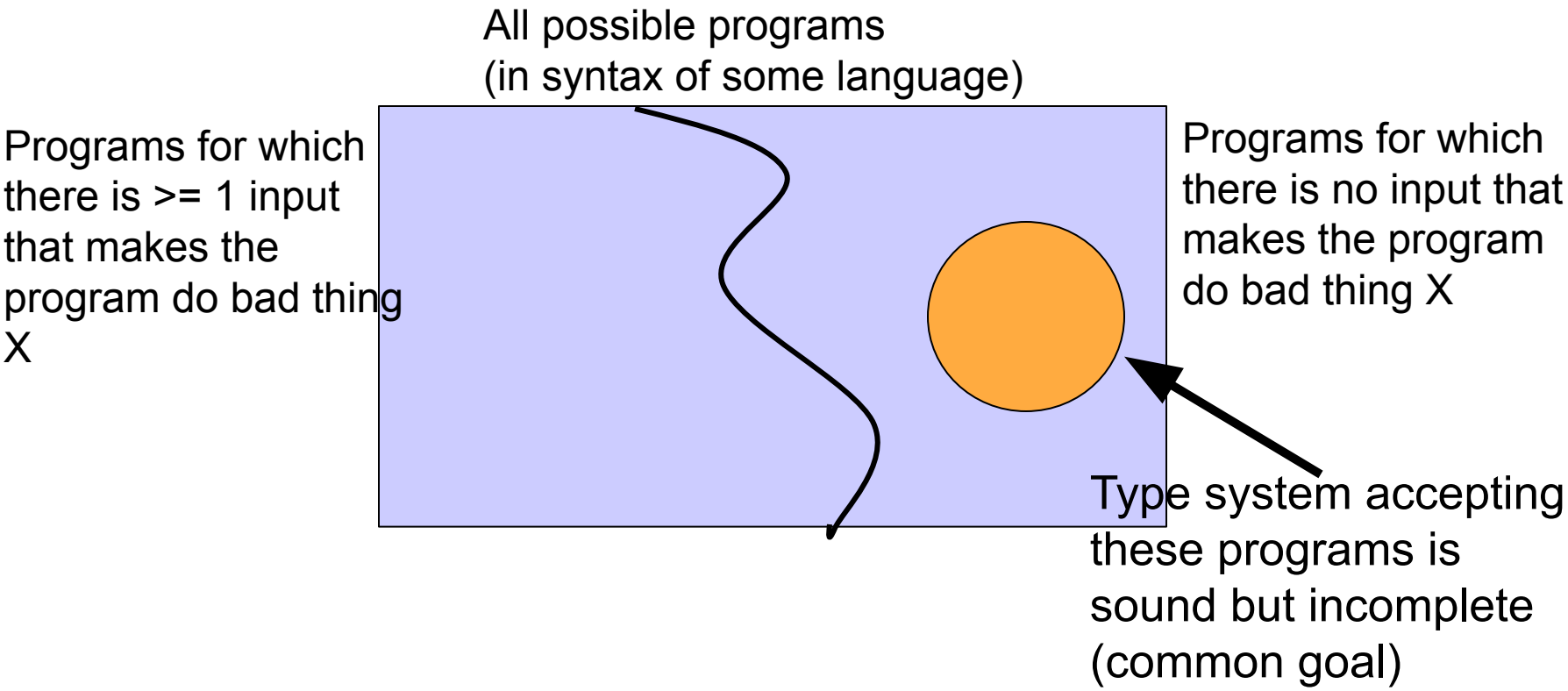
All possible programs
(in syntax of some language)

Programs for which there is >= 1 input that makes the program do bad thing X



Programs for which there is no input that makes the program do bad thing X

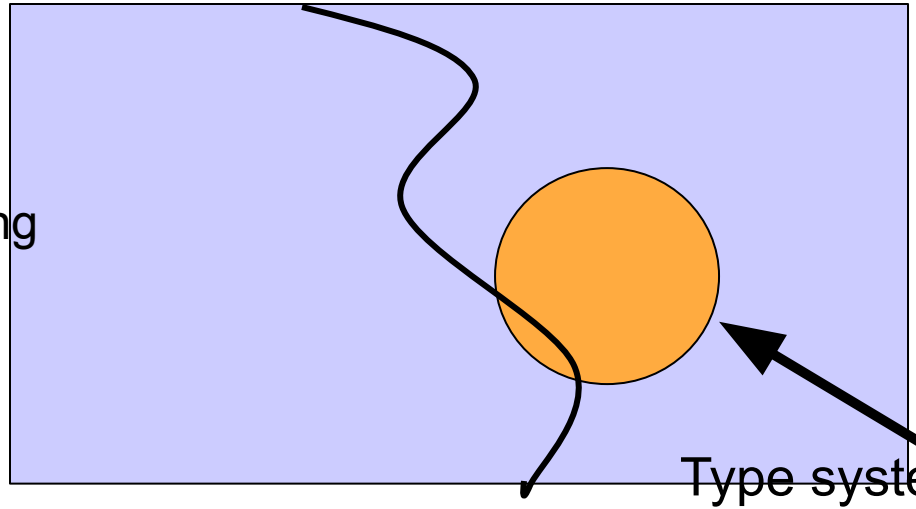
Venn Diagrams



Venn Diagrams

All possible programs
(in syntax of some language)

Programs for which there is >= 1 input that makes the program do bad thing X



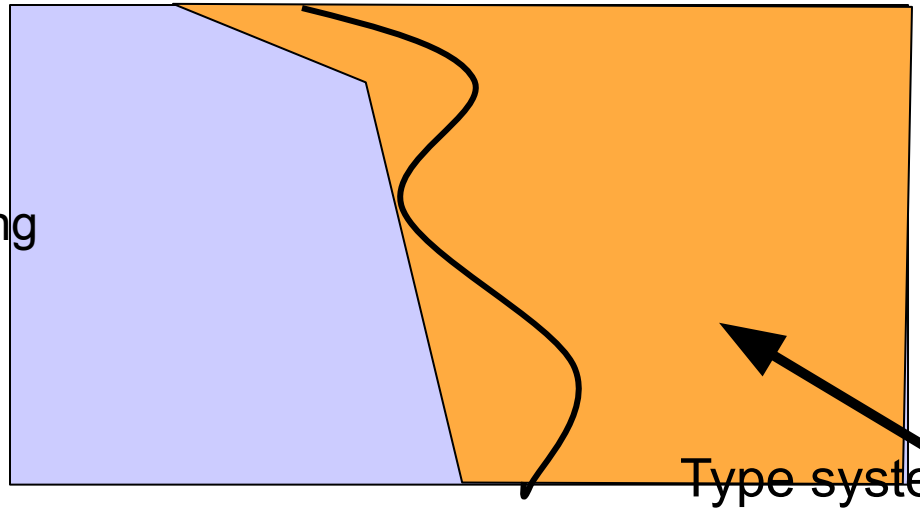
Programs for which there is no input that makes the program do bad thing X

Type system accepting these programs is unsound and incomplete

Venn Diagrams

All possible programs
(in syntax of some language)

Programs for which there is >= 1 input that makes the program do bad thing X



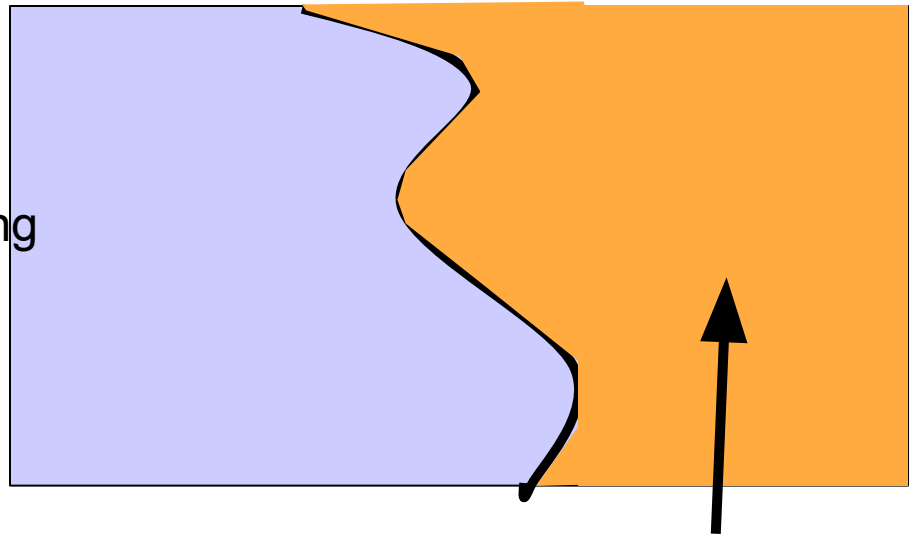
Programs for which there is no input that makes the program do bad thing X

Type system accepting these programs is unsound and is complete

Venn Diagrams

All possible programs
(in syntax of some language)

Programs for which there is >= 1 input that makes the program do bad thing X



Programs for which there is no input that makes the program do bad thing X

Type system accepting these programs is sound and complete (impossible if type-checker always terminates)

Incompleteness

OCaml rejects these even though they never misuse division:

```
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"

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

Why incompleteness

- Almost any X 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! (See CSE 311, 431, ...)
- 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

Undecidability is an essential concept at the core of computing

- The inherent approximation of static checking is probably its most important ramification

Most common example: The Halting Problem

- For many other things, “if you could solve them, then you could solve the halting problem”
  - Does `e; 4 / "hi"` divide by a string?

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 checks like array bounds

Weak typing is a poor name: Really about doing *neither* static nor dynamic checks

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 50-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 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 OCaml's or Java's rules can be convenient
  - Cons cells can build anything
  - Anything except `#f` is true
  - …

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

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))
```

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

match f x with
 | Int i -> string_of_int 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)
```

```
let 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))))
```

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

let pair_of_pairs = f (fun 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 variants)

In the extreme, always possible to do enough tagging

- See earlier discussion of `the_type`

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
```

```
let rec 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
```

```
let rec 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

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:

6. Often a lot of prototyping *before* a spec is stable
7. 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 variants 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* OCaml 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)))
```

```
let f x = 2 * x
```

```
let f x =
  match x with
  | 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 variant

- 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

Conclusion

- 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?
  - “Gradual typing”: a great idea still under active research
  - For example, see Typed Racket