FP vs. OOP Decomposition Static vs Dynamic Typing

From Dan Grossman's CSE 341 in Spring 2023

Programming Languages
UW CSE 341 - Spring 2024

Introduction of Guest Lecturers

Youssef Ben Taleb

Stanley Yang

Breaking Things Down

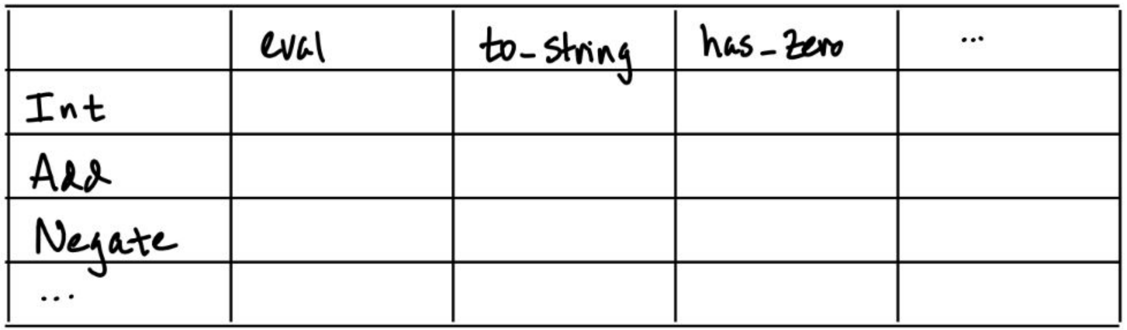
- In Functional Programming (FP), we decompose programs into functions that perform some operation
- In Object-oriented Programming (OOP), we decompose programs into classes that give behavior to some data
- **Key Insight: these are *exactly* opposite**
    - In a technical sense
    - So different they are not different at all...

Summary

- FP and OOP fill out the same “matrix” in different ways:
    - If we put data variants in rows and operations in columns, then:
    - FP organizes programs “by column”
    - OOP organizes programs “by row”
- Which is better? Depends on:
    - What changes/extensions are most likely
    - Personal taste (?) for problem domain

“The Expression Problem”

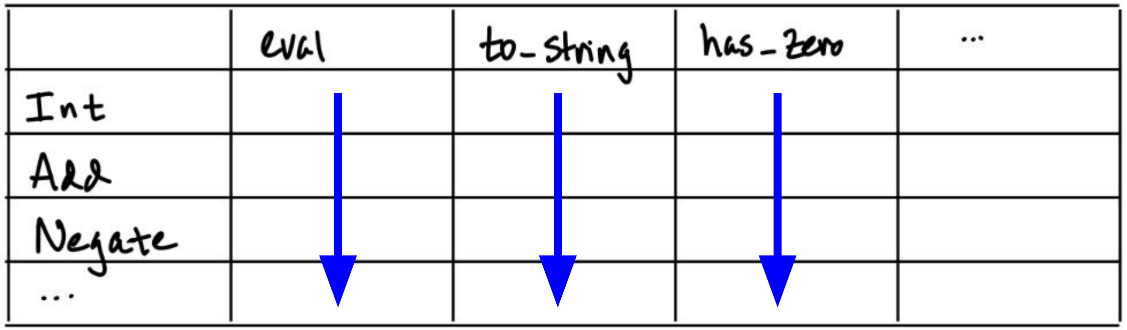
- Famous example in the PL field: expressions for a small language
  - Different expression variants : *int constants*, *additions*, *negations*, ...
  - Different operations to perform : `eval`, `to_string`, `has_zero`, ...
- Leads to a matrix of variant/operation pairs
  - In *any PL*, programmer must define behavior for all matrix positions



	eval	to_string	has_zero	...
Int				
Add				
Negate				
...				

FP Approach = “By Column”

See code!

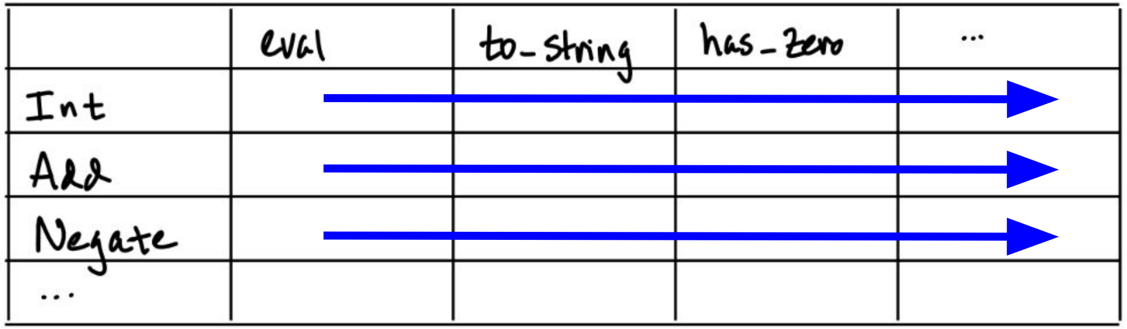


	eval	to-string	has-zero	...
Int				
Add				
Negate				
...				

- Use a recursive variant type with *1 constructor per variant*
    - Or similar approach with dynamic typing (e.g., Racket structs)
- “Fill out the grid” with one function per column
    - Each function has a branch for each row
    - Can share branches if appropriate (e.g., wildcard patterns)

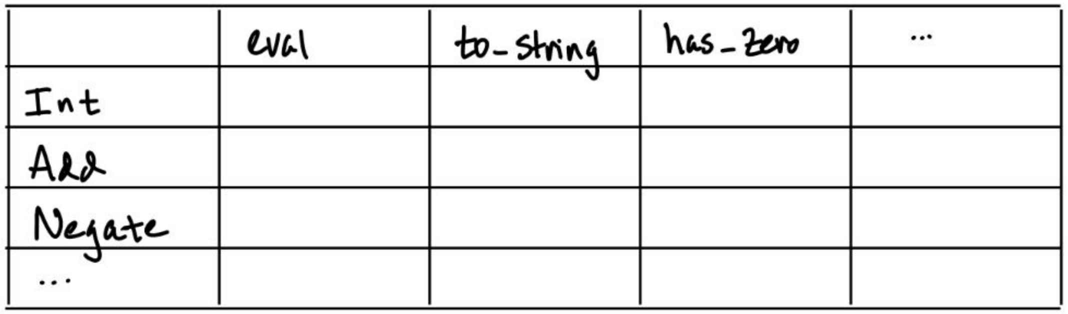
OOP Approach = “By Row”

See code!



- Define a class with *1 abstract method per operation*
  - In dynamic typing, this can be conceptual, not explicit
- “Fill out the grid” with subclass per row
  - Each class has a method for each column
  - Can share methods if appropriate (e.g., method in shared superclass)

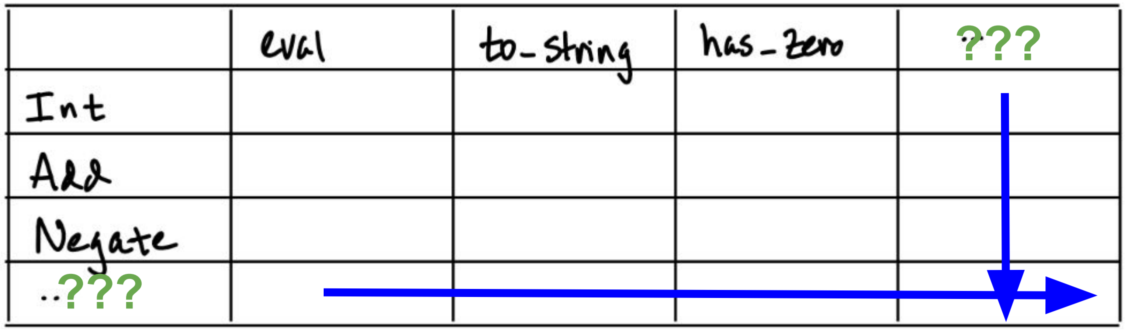
A Big Course Punchline



	eval	to_string	has_zero	...
Int				
Add				
Negate				
...				

- FP and OOP are doing the same thing in *exact* opposite way:
  - Organize the program by-row or by-column
- Which is better depends on application domain and team aesthetics
  - Regardless of program structure, can help to “think both ways”
- Code layout is important, but impossible to meet all needs
  - Tools help present other code views

Extensibility

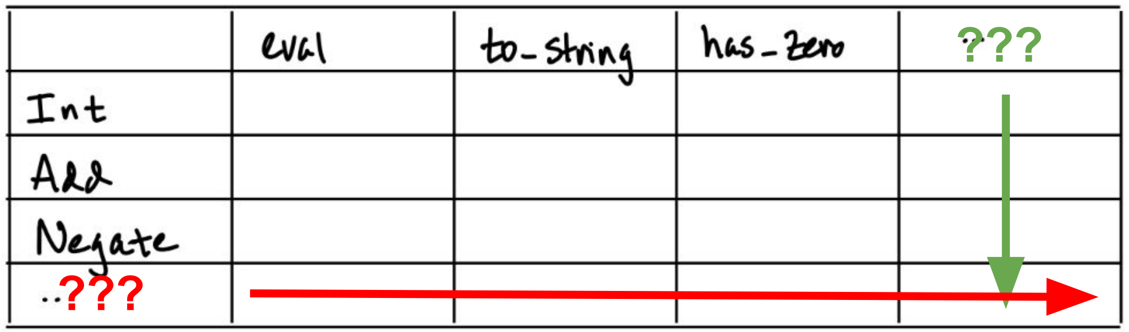


	eval	to_string	has_zero	???
Int				
Add				
Negate				
..???				

- Designing programs for *extensibility* is difficult
  - The future is often hard to predict
- What if we need to add a new kind of expression variant?
  - Annoying in FP and easy in OOP
- What if we need to add a new operation?
  - Annoying in OOP and easy in FP

FP Extensibility

See code!

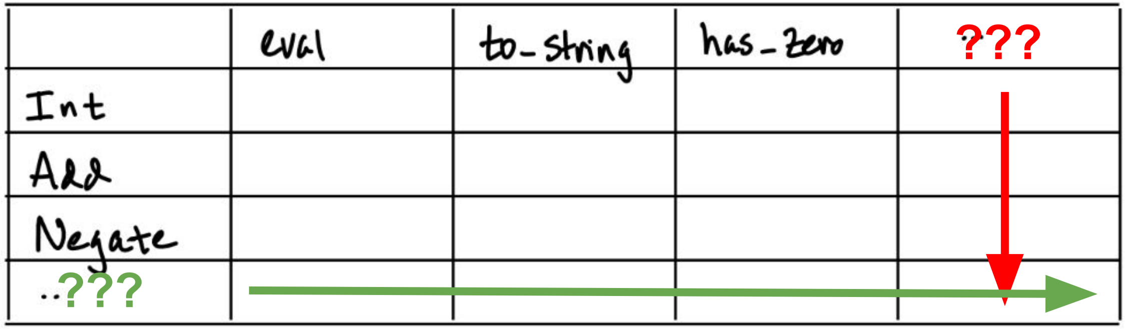


	eval	to_string	has_zero	???
Int				
Add				
Negate				
..???				

- FP: easy to add new operation, e.g., **`no-neg-constants`**
- FP: annoying to add new expression variant, e.g., **`Mult`**
    - Have to revisit all old operations implementations to add a case
    - OCaml type-checker does help by giving a to-do list (if avoided wildcard patterns)

OOP Extensibility

See code!



	eval	to_string	has_zero	???
Int				
Add				
Negate				
..???				

- OOP: easy to add new variant, e.g., `Mult`
- OOP: annoying to add new operation,e.g., `no-neg-constants`
  - Have to revisit all old classes to add a method
  - Java type-checker does help by giving a to-do list (if avoided default implementations in superclass)

Extensibility is Difficult

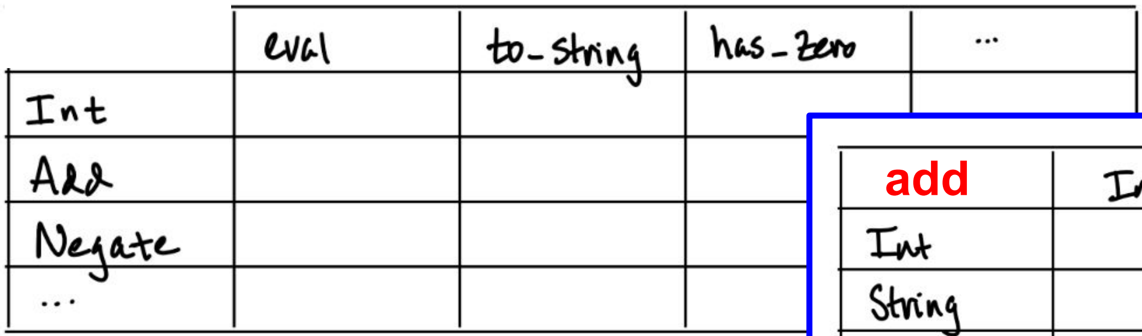
- If you know new operations are coming, use FP
  - In OOP, can plan ahead with advanced techniques like the Visitor Pattern
- If you know new variants are coming, use OOP

- Some languages try to support both well, but there are trade-offs in approaches
  - See Scala as an example

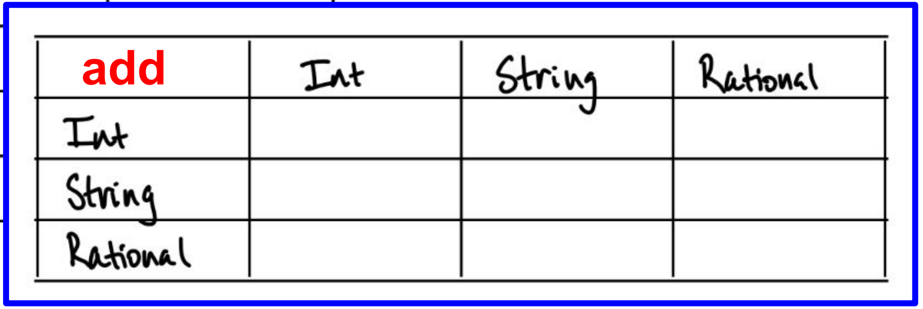
Thoughts on extensibility

- Reality: we often don't know how we or others will want to extend/change program behavior in the future
- Extensibility has downsides
  - Extensible code is harder to understand and maintain
    - Must always think about the future
    - Must always reason about correctness with respect to unknown extensions
  - Thus we often end up with features to **restrict** extensibility, e.g., **`final`** classes in Java

Binary Operations and OOP's "Double Dispatch"



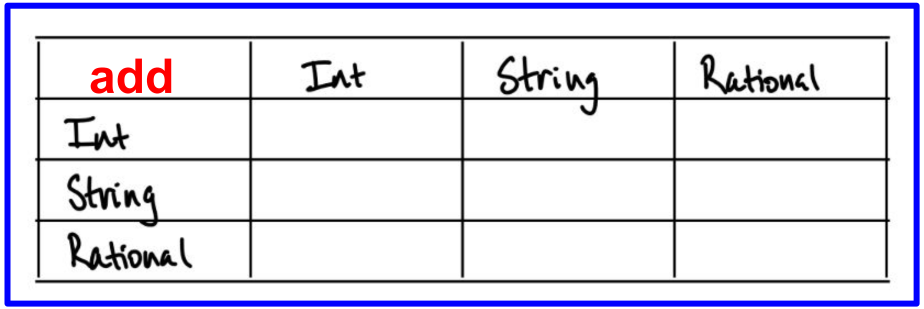
	eval	to_string	has_zero	...
Int				
Add				
Negate				
...				



add	Int	String	Rational
Int			
String			
Rational			

- What if we have a binary operation that depends on >1 variants?
  - A *different* matrix: pair of variants, *not* variants and operations
- Easy in FP: just pattern match on pair of arguments
- More complicated in OOP...

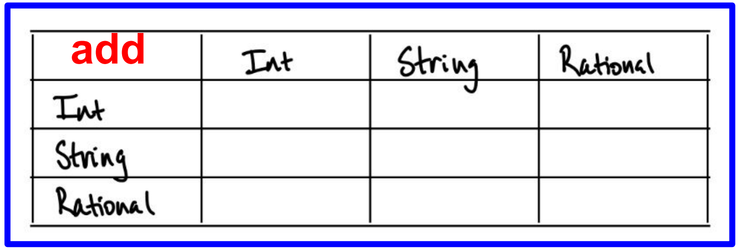
Example



add	Int	String	Rational
Int			
String			
Rational			

- Enrich our expression language with **`String`** and **`Rational`**
- Change the **`Add`** operation to allow recursive results to be any combination of **`Int`**, **`String`**, and **`Rational`**
  - If >= 1 argument is a string, then string concatenation, else math
- Now the "**`eval`** for **`Add`**" square of our first matrix requires implementing another 3x3 matrix

Binary Operations Easy in FP



add	Int	String	Rational
Int			
String			
Rational			

- Just pattern match on the pair of values
    - 9 cases in 9 branches

See code!

Now try OOP...

add	Int	String	Rational
Int			
String			
Rational			

- Starts okay: Each value class should have an **`add-value`** method the **`eval`** method of can **`add%`** can use
    - Each class handles 3 of 9 cases, where **`this`** is the first argument

```
(define add% ...
   (define/public (eval)
      (send (send _e1 eval) add-value (send _e2 eval)))
(define int% ...
   (define/public (add-value other) ...)
(define string% ...
   (define/public (add-value other) ...)
(define rational% ...
   (define/public (add-value other) ...)
```

add	Int	String	Rational
Int			
String			
Rational			

Uh-oh

- Each **`add-value`** implementation knows variant of **`this`** but not the variant of **`other`**
  - Needs to know, right?
  - Tempting/works to use **`is-a?`** but reverting to half-FP style
    - NOT allowed on Homework 7

```
(define int% …
   (define/public (add-value other)
      (cond [(is-a? other int%) … ]  ; get other’s i and +
            [(is-a? other string%) …] ; get other’s s and concat
            [(is-a? other rational%) …] ; get other’s n,d and
math
            [#t (error …)]))))
```

Another way

- OOP has a consistent answer for whenever you *think* you need a **`cond`** with **`is-a?`** branches:
    - Instead, each class should define a method to do the computation
    - You should call that method, relying on dynamic dispatch
- But here that computation needs will need to have the other add argument and know its variant
    - In **`add-value`**, that other argument is **`this`**
    - And each class knows its variant
- *double-dispatch* trick/idiom gets the callee the information it needs

Double-dispatch

See code!!

- **`int%`**, **`string%`**, and **`rational%`** each define **`add-int`**, **`add-string`**, **`add-rational`**
  - 9 total methods, 1 for each case of adding values
  - **`add-X`** takes an **`X`** that is the left-argument to addition
  - **`this`** is the right-argument
- First dispatch: **`add%`**'s **`eval`** calls **`add-value`** of left argument
- Second dispatch: **`add-value`** implementation calls correct method of right argument with caller's **`this`** for callee's **`other`**

Why am I teaching you this?

- Honestly, partly to belittle full commitment to OOP
    - Full commitment to any approach leads to unnecessary wizardry
- To show a sophisticated idiom that requires *really* understanding dynamic dispatch
- Implementing The Visitor Pattern, which helps do FP-style decomposition in OOP languages, uses double dispatch
- To contrast with *multimethods* (optional material ahead)

Works in Java too

See code!

In a statically typed OOP language, double-dispatch works without problem:

```
abstract class Value extends Exp {
   abstract Value addValue(Value other);
   abstract Value addInt(Int other);
   abstract Value addString(MyString other);
   abstract Value addRational(Rational other);
}
class Int extends Value { … }
class Strng extends Value { … }
class Rational extends Value { … }
```

[optional material ahead]

[We won’t test you on what remains in this lecture and it’s not on the homework]

But this will improve your understanding of:

- Java
- Static overloading
- Dynamic dispatch
- Double dispatch

So don’t be surprised if you find yourself looking for this material some day :-)

Java, in more common practice

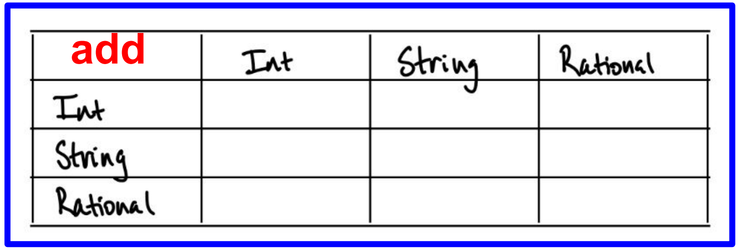
See code!

Because Java has *static overloading*, no need to give different names to methods that take different types of arguments

```
abstract class Value extends Exp {
   abstract Value add(Value other);
   abstract Value add(Int other);
   abstract Value add(MyString other);
   abstract Value add(Rational other);
}
```

Can make double dispatch seem even more magical/confusing, but same thing:

- First dispatch “only knows” its argument has type **`Value`**
- Second dispatch passes **`this`** which has a more specific type like **`Rational`**
- Static overloading *is* weird: **`this`** has a different *static* type in *inherited* code



add	Int	String	Rational
Int			
String			
Rational			

What if instead...

What if we could implement 9 `add-value` methods and have:

```
(define add%
   …
   (define/public (eval)
      (send (send _e1 eval) add-value (send _e2 eval)))
```

Pick the right 1 of the 9 with *one* dispatch?

This would require dynamic dispatch on the receiver *and* the argument

- Racket and Java do dynamic dispatch *only on the receiver*
- This semantics is called *multimethods*; has support in some languages
- Multimethods have pluses and minuses; are *not* static overloading

Static vs. Dynamic Typing

From Dan Grossman's CSE 341 in Spring 2023

Programming Languages
UW CSE 341 - Spring 2024

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!

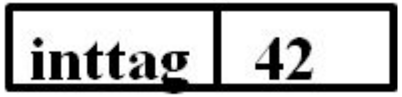
- Key questions
  - What *is* type checking? *Static typing*? *Dynamic typing*?
  - Why is static type checking *necessarily approximate*?
  - What are the *pros and cons of using static type checking*
- But first to gain a useful perspective:
  - How could a Racket programmer describe OCaml?
  - How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

- Each **struct** declaration adds a new constructor to **the_type**

Static Checking

- Static checking is anything done to possibly reject a program after it (successfully) parses but *before* it runs
- Exactly what static checks are performed is part of a PL's definition
  - A "helpful tool" could do more checking
- 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 can 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

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

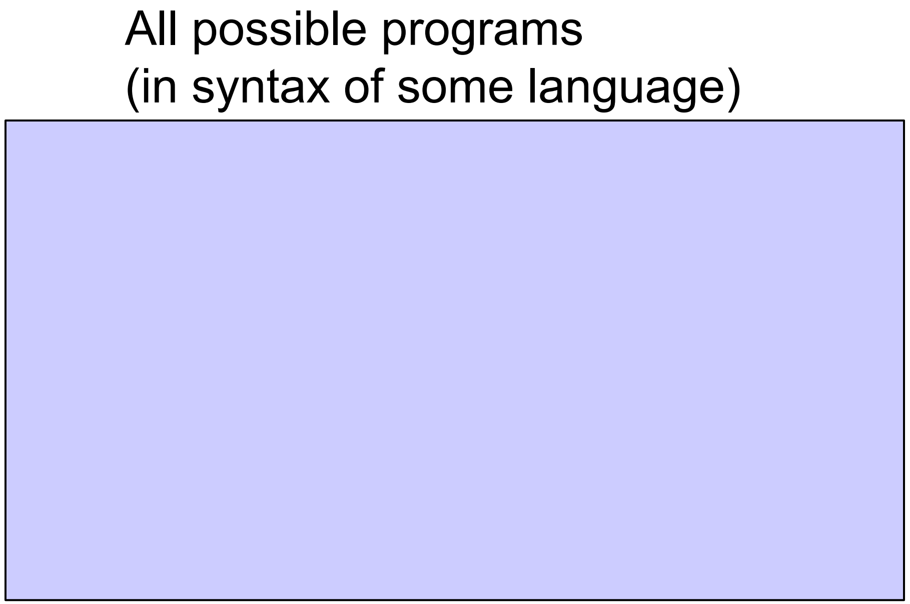
- A type system is *sound* if it never accepts a program that, when run with some input, does X
  - No *false negatives*

- 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”

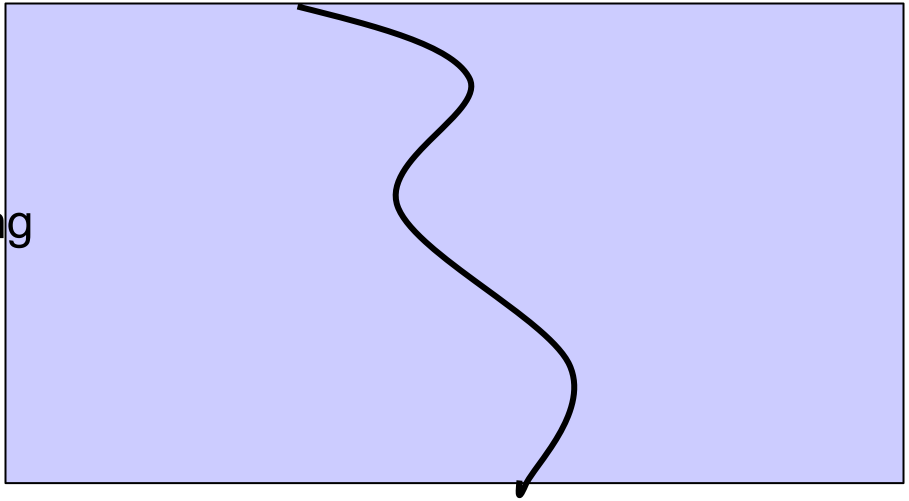
Venn Diagrams



Venn Diagrams

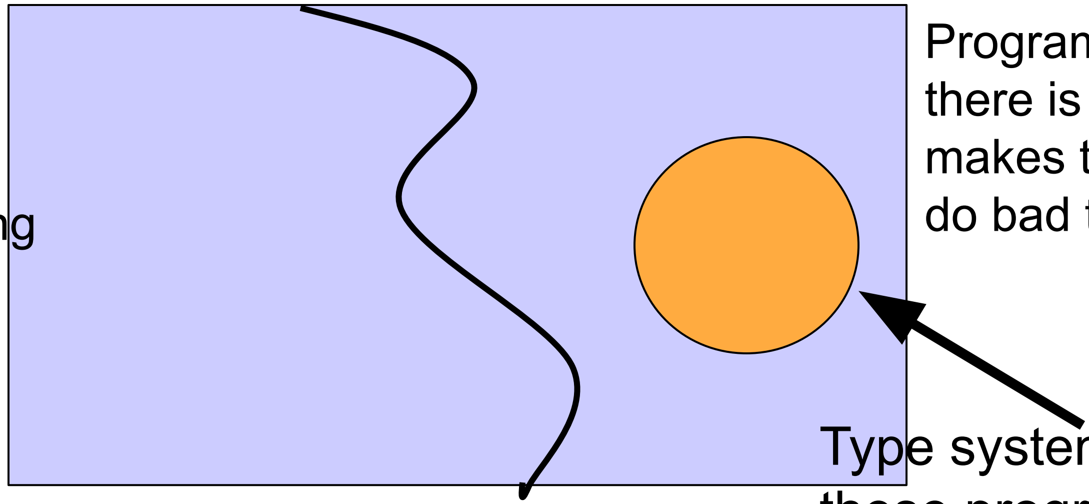
All possible programs
(in syntax of some language)

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

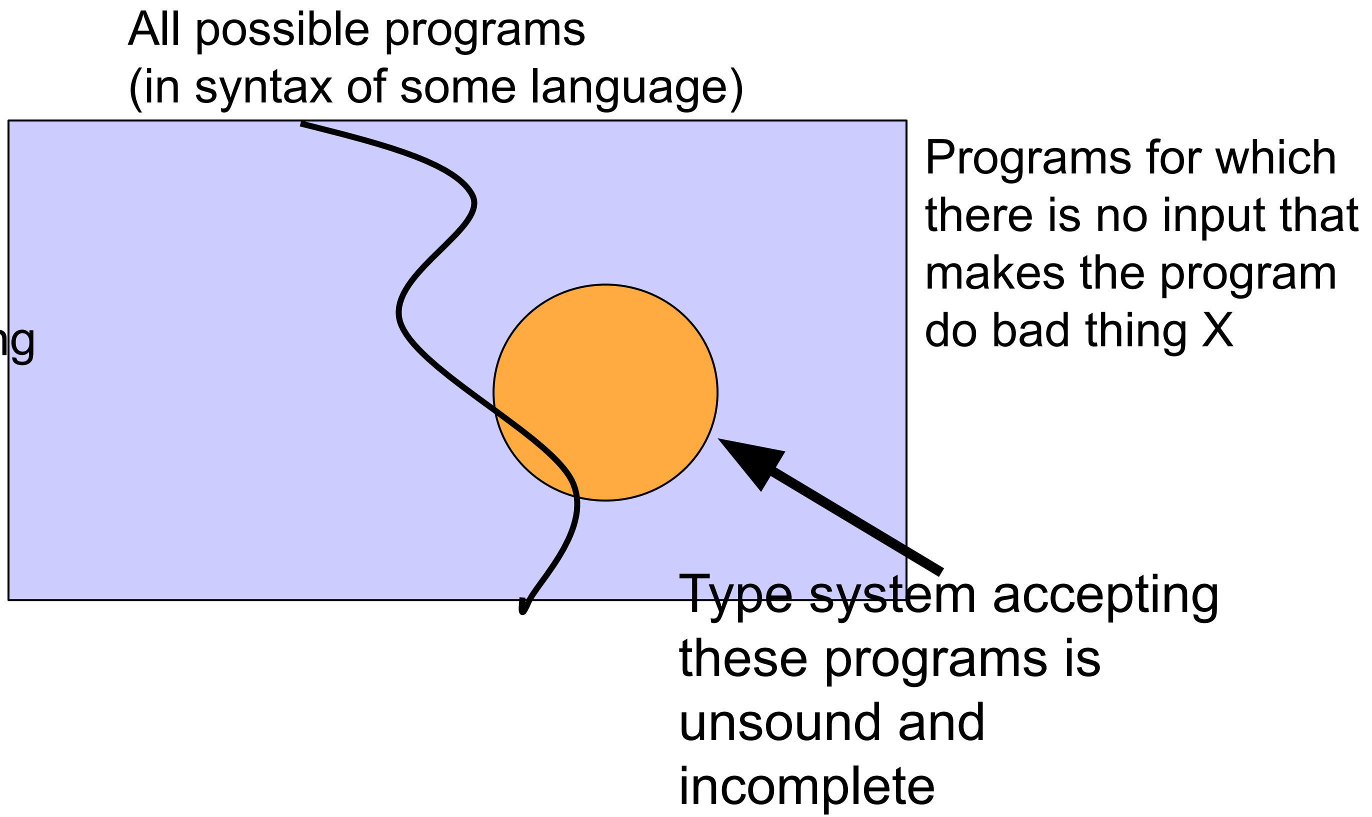


Programs for which there is no input that makes the program do bad thing X

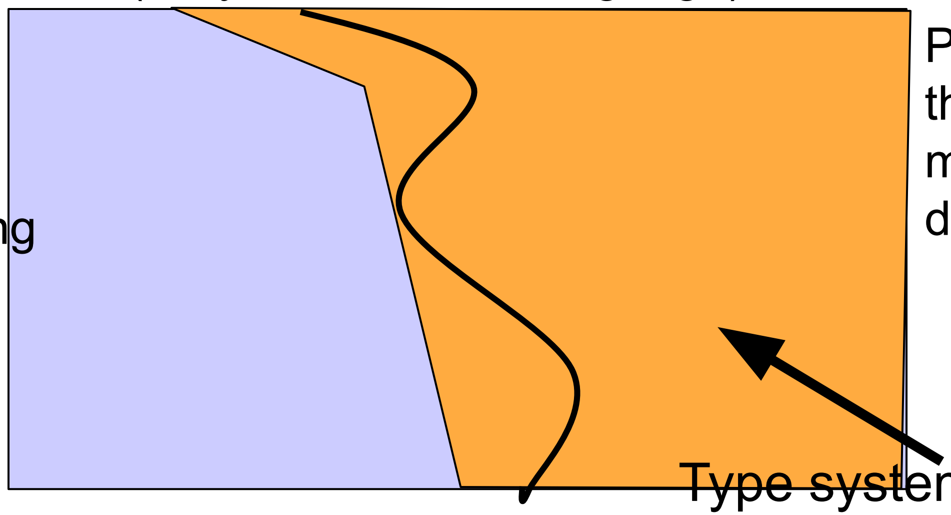
Venn Diagrams


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 but incomplete (common goal)

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

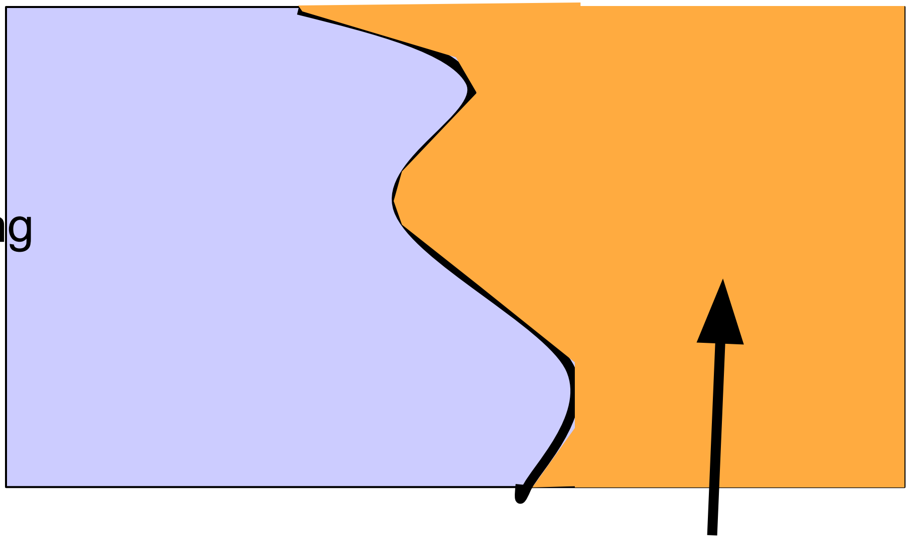
Programs for which there is no input that makes the program do bad thing X

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

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