Perhaps the most important difference between Ruby and Java is that Java has a type system that rejects many “programs” at compile-time, meaning they aren’t really Java programs after all. The purpose of this lecture is to give some perspective on what it means to use a type system for static checking (i.e., static typing) rather than detecting type errors at run-time (i.e., dynamic typing).

Software developers often have vigorous opinions about whether they prefer their languages to have static typing or dynamic typing. We will first discuss what static checking is, focusing on (i) how different programming languages check different properties statically and (ii) how static checking is inherently approximate. This discussion will give us the background to discuss various advantages and disadvantages of static checking with clear facts and arguments rather than subjective preferences.

1 Static checking

Static checking is anything done to reject a program after it (successfully) parses but before it runs. Although it’s a bit difficult to define “before it runs,” one of the key characteristics is that static checking doesn’t get to know what specific inputs are supplied to the program. Static checking is also called compile-time checking, although it’s irrelevant whether the language implementation will use a compiler or an interpreter after static checking succeeds.

If a program is rejected because it fails to parse (e.g., a closing parenthesis is missing), the resulting error is usually called a syntax error. In contrast, an error from static checking (e.g., an undeclared variable, or using a integer where is string is expected), is often called a semantic error.

The definition of a programming language specifies what static checking must be performed. Some languages don’t do any static checking; others have heavyweight static-checking machinery that requires the programmer’s assistance. Beyond the static checking required by the language, additional tools can do more static checking to find bugs or ensure their absence—even if such tools are not part of the language definition.

When to report an error. As ML and Ruby demonstrate, the typical points at which to report errors compile time and run time. However, it’s worth realizing that there is really a spectrum of eagerness about when to report errors. Consider division-by-zero errors, which static checkers rarely prevent. Given a function containing expression 3.0/0.0, an error could be raised at:

- Keystroke-time: Adjust the editor so that users cannot type / followed by 0. This is approximate, because maybe the user was about to type 0.33.
- Compile-time: Make the type-checker reject any program containing an expression with literal division by zero. This is approximate, because maybe the entire expression is if false then 3.0/0.0 else 42.0.
- Link-time: Make the linker reject any code that might be called from a “main” function and that contains literal division by zero. This is less approximate than compile-time (since some code might never be used), but still approximate (for the same reason compiler-time was).
- Run-time: Have the run-time raise an error as soon as the division by zero is attempted.
- Even later: Rather than raise an error, the run-time could return a value indicating “division-by-zero was attempted” and not raise an error until that value was used for something where an actual number is needed, such as indexing an array.

Although the “even later” option might seem too permissive at first, it’s exactly what floating-point computations do. In Ruby, expression 3.0/0.0 produces infinity, which can be used as an object but cannot be
converted to an exact number. In scientific computing this can be very useful, because it avoids having to write lots of special case code.

2 Type systems

The most common way to define a language’s static checking is with a type system. When we studied ML, we gave typing rules for each language construct: each variable has a type, the two branches of a conditional must have the same type, etc. (You also learned such rules, albeit more informally, when you learned Java.) ML’s static checker ensures these rules are followed and, in doing so, does type inference. This is the language’s approach to static checking, which is different from the purpose of static checking. The purpose is to reject programs that “make no sense” or “try to misuse” a language feature.

One goal of most type systems is to prevent application of a primitive operation to a value for which that operation is undefined. For example, one purpose of Java’s type system is to prevent passing strings to arithmetic primitives like the division operator. In contrast, Ruby uses dynamic checking (a.k.a run-time checking) by tagging each value and having the division operator check the run-time type of its argument. The Java implementation does not have to tag values for this purpose, because it can rely on static checking. Another common goal of type systems is to ensure that all variables used actually exist in the environment.

Other type-system goals are language dependent. For example, ML checks that no pattern-match has redundant patterns, and Java checks that all interface methods are implemented (with the right types) by classes.

And there are other goals the type system does not attempt to achieve. Sometimes that means the run-time must perform dynamic checks. For example, most modern languages (including ML and Java) check for array-bounds errors and division-by-zero errors at run-time. But C doesn’t check for array-bound errors (and that leads to security vulnerabilities). Static checking can also prevent these errors, but might reject programs at compile-time that will never exhibit an error at run-time.

The most extreme goal of a static type system might be to guarantee that a program is correct according to some specification—for example, sort an array, compute a fast Fourier transform, etc. Having the specification is essential: how could a type system “know” that you reversed the branches of a conditional or used addition when you meant subtraction?

Languages are often described as being either statically typed (using a compile-time type system) or dynamically typed (checking operations at run-time). In fact, there’s really a spectrum between how much checking is done at compile-time vs. run-time.

Ruby is dynamically typed. The Racket method def m; 4/"hi"; end is perfectly legal and causes no problem until m is called, at which point a dynamic error is raised. Although this m is not useful, the fact that Ruby does not impose a type system is a feature Ruby programmers do leverage. It lets arrays hold anything, any value other then false or nil to be treated as “true,” different kinds of data to be passed to or returned from a function without defining a datatype, etc. The obvious trade-off is that there is no type system to catch some of our more obvious bugs with method calls.

Ruby performs dynamic checks to make sure primitives are passed arguments of the right class, methods are called with the right number of arguments, etc. Implicit here is the definition of “correct” — that it is an error to pass non-numbers to division, to call a function with the wrong number of arguments, or to use a too-large-index to access a vector.

But what arguments are allowed is also part of a language definition. Some languages take a lenient view. Maybe calling a function with too many arguments is okay and the run-time should just ignore the extra arguments (as in C with varargs). Maybe an array-bounds error should just grow the size of the array (as in Ruby). Maybe passing a string to an arithmetic operation means some sort of string function (as in many

\[\text{https://github.com/ruby/ruby/blob/trunk/numeric.c#L835}\]
scripting languages). These choices involve a trade-off between convenience (maybe it’s useful to implicitly grow an array to avoid array-bounds errors) and bug detection (maybe it’s useful to know right away that an array-index calculation is wrong).

3 Correctness of type systems

Intuitively, a type system is “correct” if it achieves its goals and prevents what it claims to prevent. To define that more precisely, we introduce two new terms: soundness and completeness. For both, the definition is with respect to some action \( X \) that should be prevented. For example, \( X \) could be “adds a string to an integer” or “looks up a variable that is not in the environment” or “invokes a method that doesn’t exist.” Whether \( X \) happens or not is a property of a program.

- A type system is **sound** if it never accepts a program that, when run with some input, does \( X \).
- A type system is **complete** if it never rejects a program that, no matter what input it is run with, will not do \( X \).

A good way to understand these definitions is that soundness prevents false negatives and completeness prevents false positives. The terms *false negatives* and *false positives* come from statistics and medicine: Suppose there is a medical test for a disease, but it is not a perfect test. If the test does not detect the disease but the patient actually has the disease, then this is a false negative (the test was negative, but that’s false). If the test detects the disease but the patient actually does not have the disease, then this is a false positive (the test was positive, but that’s false). With type checking, the disease is “performs \( X \) when run with some input” and the test is “does the program type-check.”

In modern languages, type systems are sound (they prevent what they claim to) but not complete (they reject programs they need not reject). Soundness is important because it lets language users and language implementers rely on \( X \) never happening. Completeness would be nice, but it seems rare in practice that a program is rejected unnecessarily. And in those cases it is typically easy for the programmer to modify the program such that it type-checks.

Type systems are not complete because for almost anything you might like to check statically, it is impossible to implement a static checker that (i) always terminates, (ii) is sound, and (iii) is complete. Since we have to give up one, (iii) is the best option. (Programmers don’t like compilers that don’t terminate.)

This impossibility result is exactly the idea of undecidability that you’ll study in another course (CSci 3313). Knowing that nontrivial properties of programs are undecidable is essential to being an educated computer scientist. The fact that undecidability directly implies the inherent approximation (i.e., incompleteness) of static checking is perhaps the most important consequence of undecidability. It’s provably impossible to write a program that takes as input another program in ML, Java, etc. and always correctly outputs answers to questions such as, “will this program divide-by-zero?”, “will this program treat a string as a function?”,” will this program terminate?”, etc.

4 An orthogonal issue: Strong typing vs. weak typing

Suppose a type system is unsound for some property \( X \). Then to be safe the language implementation should perform dynamic checks wherever necessary to prevent \( X \) from happening. The language definition should allow that these checks might fail at run-time.

But an alternative is to say it is the programmer’s fault if \( X \) happens and the language definition does not have to check. In fact, if \( X \) happens, then the running program can do anything: crash, corrupt data, produce the wrong answer, delete files, launch a virus, set the computer on fire, or behave in any arbitrary way. If a language allows implementations to behave arbitrarily when \( X \) occurs, the language is **weakly typed**.
A common source of arbitrary behavior in weakly-typed languages is array-bounds errors. The opposite of “weakly typed” is strongly typed. As usual, these terms are really more a spectrum than a binary decision.

C and C++ are weakly-typed languages. The language designers did not want the language definition to force implementations to do all the dynamic checks that would be necessary for strong typing. Those checks would incur performance costs. Moreover, they would require the implementation to keep “hidden” runtime data (e.g., tags on values) to do the checks. In C and C++, programmers expect complete control over memory layout and usage; extra “hidden fields” are not desirable.

An older (and now much rarer) perspective is embodied by the saying “strong types are for weak minds.” The idea behind this is that any strongly-typed language either unnecessarily rejects programs statically, or unnecessarily performs dynamic tests—so a human should be able to “overrule” the checks in places where he or she knows they are unnecessary.

In reality, humans are extremely error-prone. We should welcome automatic checking even if it has to err on the side of caution. Type systems continue to become more expressive, and language implementations are getting better at “optimizing away” unnecessary checks.

Meanwhile, software has become very large, very complex, and very relied upon by all of society. It is deeply problematic that one bug in a 30-million-line operating system written in C can make the entire computer vulnerable to security exploits. Since the C language provides little support for finding bugs, it is increasingly common to use other third-party tools with C code to do extended static or dynamic checking.

5 Static typing vs. dynamic typing

Let’s wade into the decades-old argument about which is better, static or dynamic typing. We won’t answer definitively whether static typing is desirable. But we will consider seven specific questions, and discuss valid arguments made for and against static typing.

5.1 Is static or dynamic typing more convenient?

The argument that dynamic typing is more convenient stems from being able to mix-and-match different kinds of data such as numbers, strings, and pairs without having to declare new type definitions or “clutter” code with pattern-matching. For example, if we want a function that returns either a number or string, we can just return a number or a string, and callers can use the result as they wish. In Ruby, we can write:

```ruby
def f y; if y>0 then y+y else "hi" end; end
(f x).to_s
```

In contrast, the analogous ML code needs to use a custom datatype and pattern-matching:

```ml
datatype t = Int of int | String of string
fun f y = if y > 0 then Int(y+y) else String "hi"
val _ = case f x of Int i => Int.toString i | String s => s
```

On the other hand, static typing makes it more convenient to assume data has a certain type, knowing that this assumption cannot be violated, which would lead to errors later. For a Ruby method to ensure some data is (e.g.) a number, it has to insert an explicit dynamic check in the code. That’s more work and makes code harder to read. The corresponding ML code has no such awkwardness:

```ml
def cube x
  if x.is_a? Fixnum
    x*x*x
```

4
fun cube x = x * x * x
val _ = cube 7

Without the \texttt{is_a?} check in the Ruby code, the actual error would arise in the body of the multiplication, which could confuse callers that did not know \texttt{cube} was implemented using multiplication.

5.2 Does static typing prevent useful programs?

Dynamic typing permits programs that static checking forbids. For example, the Ruby code below binds \([-7, 7], [\text{true}, \text{true}]\) to \texttt{pair_of_pairs} without any problem, but the corresponding ML code does not type-check, because there is no type the ML type system can give to \texttt{g}.

```ml
fun f g = (case g of Fun g' => Cons(g' (Int 7), g' (Bool true)))
```

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

5.3 Is static typing's early bug detection important?

Static typing finds errors earlier than dynamic typing. A well-known truism of software engineering is that the earlier bugs are discovered, the cheaper they are to fix. Consider this program:

```ml
datatype tort = Int of int
               | String of string
               | Cons of tort * tort
               | Fun of tort -> tort
               | Bool of bool
               | Real of real
               | ...

fun f g = (case g of Fun g' => Cons(g' (Int 7), g' (Bool true)))
```

```
val pair_of_pairs = f (Fun (fn x => Cons(x,x)))
```
def pow(x,y)
    if y==0
        1
    else
        x* pow[x,y-1]
    end
end

Although the algorithm looks correct, this program has a bug: pow expects two arguments, but the recursive call passes pow one argument as an array. This bug is not discovered until testing pow with a y not equal to 0. The equivalent ML program simply does not type-check:

fun pow x y =
    if y = 0
        then 1
    else x * pow (x,y-1) (* type error *)

A dynamic-typing proponent would argue that static checking usually catches only bugs you would catch with testing anyway. Since you must test programs regardless of whether typing is static or dynamic, the additional value of catching some bugs before testing is marginal. Consider these programs:

def wrong_pow(x,y)
    if y==0
        1
    else
        x* pow[x,y-1]
    end
end

fun wrong_pow x y =
    if y = 0
        then 1
    else x + pow x (y - 1)

Neither is a correct exponentiation function, but the error won’t be caught by ML’s type system. Testing, on the other hand, will reveal the bug here, and it would also reveal the currying bug above.

5.4 Does static or dynamic typing lead to better performance?

Static typing can lead to faster code since it does not need to perform type tests at run time. In fact, much of the performance advantage may result from not storing the type tags in the first place, which takes more space and slows down constructors. In ML, there are run-time tags only where the programmer uses datatypes rather than everywhere.

Dynamic typing has three reasonable counterarguments. First, this sort of low-level performance does not matter in most software. Second, implementations of dynamically typed language can and do try to optimize away unnecessary type tests. No optimizer can remove all unnecessary tests from every program (undecidability strikes again), but it can be easy enough in practice for the parts of programs where performance matters. Third, if programmers in statically typed languages have to work around type-system limitations, then those workarounds can erode the supposed performance advantages. After all, ML programs that use datatypes have tags too.
5.5 Does static or dynamic typing make code reuse easier?

Dynamic typing arguably makes it easier to reuse library functions. After all, if you build lots of different kinds of Ruby data out of nested arrays, you can reuse all the array library methods rather than defining your own custom methods for each data structure. On the other hand, this can mask bugs. For example, suppose you accidentally pass a list to a function that expects a tree. If `inject` (Ruby’s fold function) works on both of them, you might just get the wrong answer or cause a mysterious error later, whereas using different types for lists and trees could catch the error sooner.

Whether to re-use an existing data structure or implement your own is really an design issue that’s orthogonal to static vs. dynamic typing. It’s good to reuse code, but it’s also good to use types to separate conceptually different concepts. That way the static type-checker or a dynamic type-test can catch when you put the wrong thing in the wrong place.

5.6 Is static or dynamic typing better for prototyping?

Early in a software project, when you are developing a prototype, you frequently will change your mind about what the software should do and how the implementation will do it.

Dynamic typing is often considered better for prototyping since you do not need to expend energy defining the types of variables, functions, and data structures when those decisions are in flux. Moreover, you may know that part of your program does not yet make sense (i.e., it would not type-check in a statically typed language), but you want to run the rest of your program anyway (e.g., to test the parts you just wrote). Static typing proponents would counter that it is never too early to document the types in your software design even if (perhaps especially if) they are unclear and changing. Moreover, commenting out code or adding stubs like pattern-match branches of the form `_ => raise Unimplemented` is easy and documents which parts of the program are known not to work.

When prototyping, conciseness can matter even more than usual. Dynamic typing can be more concise, but type inference can help statically typed languages be concise too.

5.7 Is static or dynamic typing better for code evolution?

A lot of effort in software engineering is spent maintaining working programs: fixing bugs, adding new features, and evolving code to make changes.

Dynamic typing is sometimes more convenient for code evolution because we can change code to be more permissive (accept arguments of more types) without having to change any of the pre-existing clients of the code. For example, consider changing this simple function:

```ruby
def double x
  2*x
end
```

to this version, which can process numbers or strings (or anything that support the `+` message):

```ruby
def double x
  x+x
end
```

No existing caller, which presumably uses `double` with numbers, can tell this change was made, but new callers can pass in strings or even values where they do not know if the value is a number or a string. If we make the analogous change in ML, no existing callers will type-check since they all must wrap their arguments in the `Int` constructor and use pattern-matching on the function result:
fun double x = 2 * x

datatype t = Int of int | String of string
fun double x =
  case f x of
    Int i => Int (2 * i)
  | String s => String (s ^ s)

On the other hand, static type-checking is very good at catching bugs that evolution introduces. When we change the type of a function, all callers no longer type-check, which means the type-checker gives us an invaluable “to-do list” of all the call-sites that need to change. A particularly good example in ML is when you need to add a new constructor to a datatype. If you did not use wildcard patterns, then you will get a warning for all the case-expressions that use the datatype. Of course, as valuable as the to-do list is, it can be frustrating that the program will not run until all items on the list are addressed.