CSE341, Fall 2011, Lecture 18 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

The most important difference between Racket and ML is that ML has a type system that rejects many programs "before they run" meaning they are not ML programs at all. (Ruby and Java have an analogous difference.) 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 (1) the fact that different programming languages check different properties statically and (2) that 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.

What is Static Checking?

What is usually meant by "static checking" is anything done to reject a program *after* it (successfully) parses but *before* it runs. If a program does not parse, we still get an error, but we call such an error a "syntax error" or "parsing error." In contrast, an error from static checking, typically a "type error," would include things like undefined variables or using a number instead of a pair. We do static checking without any input to the program identified — it is "compile-time checking" though it is irrelevant whether the language implementation will use a compiler or an interpreter after static checking succeeds.

What static checking is performed is part of the definition of a programming language. Different languages can do different things; some language do no static checking at all. Given a language with a particular definition, you could also use other tools that do even more static checking to try to find bugs or ensure their absence even though such tools are not part of the language definition.

The most common way to define a language's static checking is via a *type system*. When we studied ML (and when you learned Java), we gave typing rules for each language construct: Each variable had a type, the two branches of a conditional must have the same type, etc. ML's static checking is checking that these rules are followed (and in ML's case, inferring types to do so). But 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 "may try to misuse a language feature."

For example, one purpose of ML's type system is to prevent passing strings to arithmetic primitives like the division operator. In contrast, Racket uses "dynamic checking" (i.e., run-time checking) by tagging each value and having the division operator check that its arguments are numbers. The ML implementation does not have to tag values for this purpose because it can rely on static checking. But as we will discuss below, the trade-off is that the static checker has to reject some programs that would not actually do anything wrong.

What does a type system prevent?

Remember that what a type system prevents depends on the language, but in practice there are several standard things it would be very odd for a type system not to prevent. For example, type systems ensure that when a program runs, it will never use a variable that is not in the current environment or apply a primitive operation (such as a function call) to the wrong kind of value (such as something that is not a function). Additional properties checked statically differ among languages. For example, ML ensures no pattern-match has redundant patterns and that the signatures for modules are respected (e.g., code outside modules cannot directly call functions in the module but not in the signature).

It is just as important to recognize what a type system is *not* checking so that language users can be

more careful (not having a type system to catch their errors) and language implementers can make sure the necessary dynamic checks are performed. Some standard things that most languages, including ML and Java, check dynamically are array-bounds errors and division-by-zero. One could design static checkers that prevent these errors, but they might end up rejecting too many programs that are not actually a problem. Also remember that no static type system can make sure your program is correct unless it had a full specification of what your program was supposed to do. For example, how can a type system "know" that you reversed the branches of your conditional or used addition when you meant subtraction?

We often say a language is "statically typed" (it uses a type system to perform static checking for lots of things) or "dynamically typed" (it checks types for operations at run-time instead). This terminology is a little loose since the line dividing how much a language must check statically to be a statically typed language is not precise, but in practice we usually all agree on a language's categorization.

Correctness: Soundness, Completeness, Undecidability

Intuitively, a static checker is correct if it prevents what it claims to prevent — otherwise, either the language definition or the implementation of static checking needs to be fixed. But we can give a more precise description of correctness by defining the terms *soundness* and *completeness*. For both, the definition is with respect to some action X we wish to prevent, e.g., X could be "looks up a variable that is not in the environment."

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 static 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 hopefully it is rare in practice that a program is rejected unnecessarily and in those cases it is 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 given any program in your language (a) always terminates, (b) is sound, and (c) is complete. Since we have to give up one, (c) seems like the best option (programmers don't like compilers that may not terminate).

The impossibility result is exactly the idea of *undecidability* you study in another course (CSE 311). Knowing what it means that nontrivial properties of programs are undecidable is fundamental to being an educated computer scientist. The fact that undecidability directly implies the inherent approximation (i.e., incompleteness) of static checking is probably the most important ramification of undecidability. We simply cannot write a program that takes as input another program in ML/Racket/Java/etc. that always correctly answers questions such as, "will this program divide-by-zero?" "will this program treat a string as a function?" "will this program terminate?" etc.

Digression: "Strong Typing" vs. "Weak Typing"

Now suppose a type system is unsound for some property X. Then to be safe the language implementation

should still, at least in some cases, perform dynamic checks to prevent X from happening and 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, or set the computer on fire. If a language has programs where a legal implementation is allowed to set the computer on fire (even though it probably wouldn't), we call the language *weakly typed*. Languages where the behavior of buggy programs is more limited are called *strongly typed*. These terms are a bit unfortunate since the correctness of the type system is only part of the issue. After all, Racket is dynamically typed but nonetheless strongly typed. Moreover, a big source of actual undefined and unpredictable behavior in weakly typed languages is array-bounds errors (they need not check the bound, they can just access some other data by mistake), yet few type systems check array bounds.

C and C++ are the well-known weakly typed languages. Why are they defined this way? In short, because the designers do not want the language definition to force implementations to do all the dynamic checks that would be necessary. While there is a time cost to performing checks, the bigger problem is that the implementation has to keep around extra data (like tags on values) to do the checks and C/C++ are designed as lower-level languages where the programmer can expect extra "hidden fields" are not added.

An older now-much-rarer perspective in favor of weak typing is embodied by the saying "strong types for weak minds." The idea is that any strongly typed language is either rejecting programs statically or performing unnecessary tests dynamically (see undecidability above), so a human should be able to "overrule" the checks in places where he/she knows they are unnecessary. In reality, humans are extremely error-prone and we should welcome automatic checking even if it has to err on the side of caution for us. Moreover, type systems have gotten much more expressive over time (e.g., generics) and language implementations have gotten better at optimizing away unnecessary checks (they will just never get all of them). Meanwhile, software has gotten very large, very complex, and relied upon by all of society. It is deeply problematic that 1 bug in a 30-million-line operating system written in C can make the entire computer subject to security exploits. While this is still a real problem and C the language provides little support, it is increasingly common to use other tools to do static and/or dynamic checking with C code to try to prevent such errors.

Racket is Dynamically Typed

The Racket function `(define (f) (/ 4 "hi"))` is perfectly legal and causes no problem until `f` is called, at which point a dynamic error is raised. While this `f` is not useful, the fact that Racket does not impose a type system is a feature Racket programmers do leverage. It lets us use cons cells to build anything, to treat anything that is not `#f` as true, to pass different kinds of data to/from a function without defining a datatype, etc. The obvious trade-off is that there is no static checker to catch some of our more obvious bugs and to ensure that we use functions with the expected kinds of arguments.

What Types are Allowed for an Operation

Racket performs dynamic checks to make sure primitives are passed arguments of the right types, functions 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 much more lenient view. Maybe calling a function with too many arguments is okay and we should just ignore the extra ones. Maybe an array-bounds error should just grow the size of the array. Maybe passing a string to an arithmetic operation means some sort of string function. These choices involve a trade-off between convenience (maybe it is useful to implicitly grow an array to avoid array-bounds errors) and bug-detection (maybe I want to know right away that my array-index calculation might be wrong). Technically, though, these questions are a separate issue from static vs. dynamic checking.

A Continuum of Eagerness

As ML and Racket demonstrate, the typical points at which to prevent a "bad thing" are "compile-time" and "run-time." However, it's worth realizing that there is really a continuum of eagerness about when we declare something an error. Consider for sake of example something that most type systems do not prevent statically: division-by-zero. If we have some function containing the expression `(/ 3 0)`, when could we cause an error:

- Keystroke-time: Adjust the editor so that one cannot even write down a division with a denominator of 0. This is approximate because maybe we were about to write 0.33, but we were not allowed to write the 0.
- Compile-time: As soon as we see the expression. This is approximate because maybe the context is `(if #f (/ 3 0) 42)`.
- Link-time: Once we see the function containing `(/ 3 0)` might be called from some "main" function. This is less approximate than compile-time since some code might never be used, but we still have to approximate what code may be called.
- Run-time: As soon as we execute the division.
- Even later: Rather than raise an error, we could just return some sort of value indicating division-by-zero and not raise an error until that value was used for something where we needed an actual number, like indexing into an array.

While the "even later" option might seem too permissive at first, it's exactly what floating-point computations do. `(/ 3.0 0.0)` produces `+inf.0`, which can still be computed with but cannot be converted to an exact number. In scientific computing this is very useful to avoid lots of extra cases: maybe we do something like take the tangent of $\pi/2$ but only when this will end up not being used in the final answer.

Advantages and Disadvantages

Now that we know what static and dynamic typing are, let's wade into the decades-old argument about which is better. We know static typing catches many errors for you early, soundness ensures certain kinds of errors do not remain, and incompleteness means some perfectly fine programs are rejected. We won't answer definitively whether static typing is desirable (if nothing else it depends what you are checking), but we will consider seven specific claims and consider for each valid arguments made for and against static typing.

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 dynamic type predicates as necessary. In Racket, we can write:

```
(define (f y) (if (> y 0) (+ y y) "hi"))
(let ([ans (f x)]) (if (number? ans) (number->string ans) ans))
```

In contrast, the analogous ML code needs to use a datatype, with constructors in `f` and pattern-matching to use the result:

```
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 Racket function to ensure some data is, for example, a number, it has to insert an explicit dynamic check in the code, which is more work and harder to read. The corresponding ML code has no such awkwardness.

```
(define (cube x)
  (if (not (number? x))
      (error "cube expects a number")
      (* x x x)))
(cube 7)

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

Notice that without the check in the Racket code, the actual error would arise in the body of the multiplication, which could confuse callers that did not know `cube` was implemented using multiplication.

2. Does Static Typing Prevent Useful Programs?

Dynamic typing does not reject programs that make perfect sense. For example, the Racket code below binds `'((7 . 7) . (#t . #t))` to `pair_of_pairs` without problem, but the corresponding ML code does not type-check since there is no type the ML type system can give to `g`.

```
(define (f g) (cons (g 7) (g #t)))
(define pair_of_pairs (f (lambda (x) (cons x x))))

fun f g = (g 7, g true) (* does not type-check *)
val pair_of_pairs = f (fn x => (x,x))
```

Of course we can write an ML program that produces `((7,7),(true,true))`, but we may have to "work around the type-system" rather than do it the way we want.

On the other hand, dynamic typing derives its flexibility from putting a tag on every value. In ML and other statically typed languages, we can do the same thing *when we want to* by using datatypes and explicit tags. In the extreme, if you want to program like Racket in ML, you can use a datatype to represent "The One Racket Type" and insert explicit tags and pattern-matching everywhere. While this programming style would be painful to use everywhere, it proves the point that there is nothing we can do in Racket that we cannot do in 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)))
```

Perhaps an even simpler argument in favor of static typing is that modern type systems are expressive enough that they rarely get in your way. How often do you try to write a function like `f` that does not type-check in ML?

3. Is Static Typing's Early Bug-Detection Important?

A clear argument in favor of static typing is that it catches bugs earlier, as soon you statically check (informally, "compile") the code. A well-known truism of software development is that bugs are easier to fix if discovered sooner, while the developer is still thinking about the code. Consider this Racket program:

```
(define (pow x)
   (lambda (y)
      (if (= y 0)
          1
          (* x (pow x (- y 1))))))
```

While the algorithm looks correct, this program has a bug: `pow` expects curried arguments, but the recursive call passes `pow` two arguments, not via currying. 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) (* does not type-check *)
```

Because static checkers catch known kinds of errors, expert programmers can use this knowledge to focus attention elsewhere. A programmer might be quite sloppy about tupling versus currying when writing down most code, knowing that the type-checker will later give a list of errors that can be quickly corrected. This could free up mental energy to focus on other tasks, like array-bounds reasoning or higher-level algorithm issues.

A dynamic-typing proponent would argue that static checking usually catches only bugs you would catch with testing anyway. Since you still need to test your program, the additional value of catching some bugs before you run the tests is reduced. After all, the programs below do not work as exponentiation functions (they use the wrong arithmetic), ML's type system will not detect this, and testing catches this bug and would also catch the currying bug above.

```
(define (pow x) ; wrong algorithm
  (lambda (y)
     (if (= y 0)
         1
         (+ x ((pow x) (- y 1))))))

fun pow x y = (* wrong algorithm *)
   if y = 0
   then 1
   else x + pow x (y - 1)
```

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 type tests it can tell are unnecessary. For example, in `(let ([x (+ y y)]) (* x 4))`, the multiplication does not need to check that `x` and `4` are numbers and the addition only needs to check `y` once. While no optimizer can remove all unnecessary tests from every program (undecidability strikes again), it may 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. 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 data out of cons cells, you can just keep using `car`, `cdr`, `cadr`, etc. to get the pieces out rather than defining lots of different getter functions 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 `cadr` 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.

This is really an interesting design issue more general than just static vs. dynamic typing. Often it is good to reuse a library or data structure you already have especially since you get to reuse all the functions available for it. Other times it makes it too difficult to separate things that are really different conceptually so it is better to define a new type. That way the static type-checker or a dynamic type-test can catch when you put the wrong thing in the wrong place.

6. Is Static or Dynamic Typing Better for Prototyping?

Early in a software project, you are developing a prototype, often at the same time you are changing your views on what the software will do and how the implementation will approach doing 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 (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 may 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 often easy and documents what parts of the program are known not to work.

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

7. Is Static or Dynamic Typing Better for Code Evolution?

A lot of effort in software engineering is spent maintaining working programs, by fixing bugs, adding new features, and in general evolving the code to make some change.

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:

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

to this version, which can process numbers or strings:

```
(define (f x)
  (if (number? x)
      (* 2 x)
      (string-append x x)))
```

No existing caller, which presumably uses `f` 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 f x = 2 * x

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

On the other hand, static type-checking is very useful when evolving code to catch bugs that the 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. By this argument, the safest way to evolve code is to change the types of any functions whose specification is changing, which is an argument for capturing as much of your specification as you can in the types.

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.

As valuable as the "to-do list from the type-checker" is, it can be frustrating that the program will not run until all items on the list are addressed or, as discussed under the previous claim, you use comments or stubs to remove the parts not yet evolved.