Static Checking vs Dynamic Checking

Static checking is the phase after the program is properly parsed, but before it actually runs. The kind of work that is being done during this phase is part of the programming language's definition. The most standard way to perform static checking is via a **type system**. We already saw type systems extensively in our study of OCAML.

The goal of a type system is to *prevent certain programs from running*, by signaling a type error during this static-checking phase. There are many different kinds of erroneous program behavior:

- Behavior that can be prevented during static-checking by preventing the program from running: For example preventing a boolean or a pair from being used when a number is expected.
- Behavior that cannot be prevented during static-checking, but that can be detected as the program runs: Dividing by zero or trying to access an array beyond its bounds are examples of this.
- Behavior that cannot be detected by the interpreter/compiler even when the program runs. These are typically programmer errors like getting the two branches in a conditional mixed up, or using an addition when a multiplication was needed. These can only be prevented/limited via extensive testing.

Type systems

A type system prevents certain programs from running. It also sets out to deal with a specific set of "bad behavior", which we will generically denote by X. There are two important characterizations of type systems:

- A type system is called **sound** if it never accepts a program that would exhibit the behavior X.
- A type system is called **complete** if it never rejects a program that would never exhibit the behavior X.

In other words, soundness prevents false negatives, while completeness prevents false positives.

Soundness is crucial. We absolutely want our type system to be sound, and we would usually provide a formal proof of the fact.

Though we would like our type system to also be complete, it is not possible.

> We cannot have a type system that is both sound and complete, and such that the typechecker always terminates. This is basically the undecidability issues discussed in Theory of Computation.

Let us consider an example of this. Suppose that X is the property that only numbers are used in arithmetic operators. Then consider the following program:

```
(if true then 5 else (2, 1)) + 1
```

This "program" would run just fine: The conditional evaluates to 5, then the addition can be performed. But most type systems will not let this program run, as they would require the two branches of the conditional to evaluate to values of the same type. Such type systems would not be complete.

Of course in this specific case we could notice the test and know which branch will be taken, and accept the program. But it is entirely possible that the test expression is a complicated expression that does in fact always end up evaluating to true but so that this fact is not discernible without actually evaluating the expression. Then there is no way for the type system to know that it should not reject the program.

Strong and Weak Typing

There are two more terms extensively used and often misused, those of weak and strong typing.

We already talked about the possibility of a type system being unsound for certain properties, for example out-of-bounds array accesses. Most languages add dynamic runtime checks to prevent a program from actually reading or writing beyond the array bounds. Others, like C or C++, do not, to avoid the performance hit associated with those tests. These are called **weakly typed** languages, as opposed to **strongly typed** languages like Java or OCAML. A weakly typed system passes all the responsibility for preventing these erroneous behaviors onto the programmer.

Note that both of these terms refer to the language behavior during dynamic checking, not static checking as is usually the case with type system considerations.

Advantages and Disadvantages of Static Checking

We will now discuss various points related to static and dynamic checking, trying to present positives and negatives for both sides.

Convenience

A big attraction of dynamic checking is that it doesn't get in your way as much. For example you can have a list containing both numbers and strings, and you don't need to do anything special to accomplish it. By comparison, in a statically typed language you need to create a new datatype that incorporates both numbers and strings in it as variants, then do some pattern matching or something similar.

On the flip side, dynamic checking has to more often test its input to make sure it is as expected before operating on it. Imagine a simple cubing function. Who will be responsible to make sure that the function is not called on anything that is not a number? The writer of the function can do that by either using a static type system or by adding dynamic checks before doing the multiplications. Or they can leave it up to the user of the function to know how it should be called, not an ideal proposition.

Preventing useful programs

A fundamental aspect of static typing is that it prevents certain perfectly valid programs from running. We already saw some examples earlier when we discussed non-completeness. Of course it is a harder question whether there are any such useful programs that would not type-check.

One common misunderstanding about dynamic typing is the thought that values don't have a type. In fact they do, they all have the same type, but they also have tags that distinguish them. We can do a similar thing in OCAML, creating one universal data type via a long list of variants, with something like:

```
type anyType = Int of int
             | Float of float
             | Pair of anyType * anyType
             | Func of anyType -> anyType
             ....
```

We can then write functions that take almost anything as input, and they use pattern-matching to decide what "tag" the value has and behave accordingly. The different constructors play the role of the dynamic tags.

In other words, even in a statically-typed language you can get pretty close to programming in a dynamically-typed way, if you so choose. Although it definitely gets a bit cumbersome.

The other argument in favor of static type systems is that these type systems have evolved to become very expressive, and they continue to evolve. Modern type systems prevent fewer valid programs from running. This is in fact a very active area of research, how to make the type system more expressive and closer to complete, without sacrificing on soundness. Scala and Haskell are examples of modern languages with quite expressive type systems.

Bug-detection

A straightforward argument in favor of static typing is that many programmer errors are caught early on, oftentimes by a smart text editor or IDE that reports to you as you type. A common example of this would be providing a tuple to a function that expects its arguments curried, or vice versa.

The fact that such errors will be caught early allows you to focus more on your function's logic, not worrying about those details.

A counter-argument is that the kinds of errors that static typing catches are the kinds of errors that would be caught very early in the testing process anyway. And testing is needed anyway because of all the kinds of errors that cannot be detected at compile time.

Performance

Another argument in favor of static typing is that it leads to performance gains. Since you have determined at compile time the type of expressions, you do not have to do dynamic checks at runtime to make sure the value you are about to feed to a function is appropriate. *Static typing requires fewer dynamic runtime checks.*

A counterargument is that there are very few instances where optimizing for speed in that manner matters, and plenty of domains where it doesn't. It is also the case that many dynamically-checked systems are able to determine checks that are redundant and optimize them away. Furthermore, oftentimes in static typing programmers and up having to complicate their code, by for instance using type variants. And at that point we are effectively setting up the necessary runtime checks on our own via pattern matching and similar features.

Code reuse

Another key issue is the idea of code reuse. Does one of the two styles of typing make code reuse easier?

Dynamic typing certainly puts fewer constraints on this. You can use a function you wrote on pretty much anything, as long as it responds properly to the features your function uses. In a statically-typed setting you often have to force the system to let you do it. But that also protects you from subtle bugs.

The expressiveness of the system here plays a big role when it comes to code reuse. If the system offers parametric types, or generics, then you can provide container types and libraries that offer linked lists, heaps and queues and all sorts of other things in a very generic way: It's implemented once and you get to reuse it. In C++ you can do the same thing using templates.

By contrast, people working in C often have to reimplement these basic data structures, as C's type system isn't rich enough to offer the necessary capabilities.

Prototyping

A place where dynamic checking shines is prototyping. This is one of the reasons for the popularity of Python, the ease with which you can just "try things out" without having to worry about getting the types to match all the time. It also allows you to try out parts of the program even though other parts don't make sense. You don't need to glue everything together upfront, you can more easily do so incrementally.

The counterargument is that planning your types early on can be beneficial and it's never too early to get that setup right. And adding stubs that raise an exception for unimplemented features isn't all that complicated.

Code Evolution

In many cases programs need to be rewritten and refactors, pieces abstracted away into new functions and so on. Every time you change your code you need to make sure you are not breaking things.

Static typing offers many guarantees in this way. It will not let you compile things until everything has been put back together in an orderly way. It offers you a certain guarantee that you did not break certain behaviors.

On the other side, typing limits the kinds of ways in which you can extend your code. For example if we want to move from the first version of `f` below, that only handles numbers, to the second, that also handles strings, we can do it easily in Racket:

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

No previous user of the function can detect that we changed anything; we didn't break any existing code. But we did allow our function in the future to handle strings.

In a statically typed language like OCAML this would be a lot harder to do, as the type of `double` now has to go from `num->num` to some sort of datatype that puts together numbers and strings. And the corresponding users of the function have to wrap their numbers in a constructor to make that work. In other words, it's not easy to do.

Bottom line is that static typing makes it harder to effect changes that actually change the program's types. It can help you track the changes appropriately, but it does expect you to change possibly many places.

Of course before any refactoring effort you have to make sure that you have extensive unit tests in place, to make sure you don't break your program's logic. Many would argue that those tests are more than enough and that the static typing gets in the way.