Programming Paradigms

Static and dynamic typing

Hans Hüttel

November 19, 2023

In this short note I will highlight the use and usefulness of type systems and discuss the pros and cons of static and dynamic typing. The canonicalk reference to this very large topic is the excellent book by Pierce about type systems in programming language [7].

1 What are type systems?

Most programming languages today incorporate a type system of some kind that defines a way of classifying syntactic entities.

Some languages such as C have very simple type systems, while others, notably functional languages such as the languages of the ML family and Haskell have very expressive type systems. The languages of the Lisp family also have a notion of type, but a very different one.

Some widely used programming languages have no type system at all. A widely known example is Javascript.

Traditionally, type systems have been used to classify programs by means of *type checking*: given an assignment of types to data entities in program P, check that P is *well-typed* , i.e. does not contain type errors.

Implementations of typed programming languages incorporate a type checker. The main difference is if type checking occurs at compile-time or at run-time.

2 Static and dynamic type systems

A static type system carries out type checking based on the program code *before run-time.* Errors are therefore caught at compile time. C, Java, the ML languages, Scala and Haskell all have static type systems.

A dynamic type system catches type errors *at run-time.* Languages such as the Lisp dialects, Python, Ruby and Lua are dynamically typed. If a type error is found at run-time, execution stops.

In some communities, there have been heated discussions as to whether static or dynamic type disciplines are best. A very important argument for having a static type discipline is that, if a static type system can be shown to be *sound*, then certain run-time errors will never occur.

A program has a run-time error if at some point durings its execution, the execution becomes unable to progress. The program is stuck – usually because the program tries to perform an operation that does not make sense.

Let us assume that programs are expressions and that their semantics is given by a small-step transition relation where transitions are of the form $e \to' e$. Such a transition is to be read as saying that e can take a single evaluation step, after which e' remains to be evaluated. We write $e \to$ if e can perform an evaluation step and $e \to^* e'$ if e can evaluate to e' in zero or more steps.

For any programming language and any non-trivial class of errors it is undecidable if a given program is error free.

Theorem 1. *Assume that our programming language is Turing-powerful. Let C be a class of errors and e be an expression. It is undecidable if there exists an e' such that $e \to^* e'$ where e' performs the run-time error C.*

Proof. A reduction from the halting problem for Turing machines. We assume (without loss of generality) that, since our programming language is Turing-powerful, then there exists a translation from pairs $\langle M, w\rangle$ of Turing machines and strings to programs $e_{M,w}$. We also assume that our language has a notion of sequential composition such that we can execute e_1 before executing e_2. We write this as $e_1; e_2$.

We also know that there must be some program e_C that leads to a run-time error from the class of run-time errors C.

Given the description $\langle M, w\rangle$ of a Turing machine M and an input string w, we can compute a program $e_{M,w}$ that simulates the behaviour of M when run with input w.

Now the program $e_{M,w}; e_C$ will result in a run-time error if and only if M halts when given input w. □

3 What a static type system guarantees

The important observation is that while it is undecidable if a program will give rise to a run-time error, for all static type systems in use, it is the case that it is decidable if a program is well-typed. For this reason, one could hope that whatever type system we consider it should be able to give a characterization of some class C of run-time errors. If that is the case, then we know that if a program is well-typed, it will not give rise to a run-time error in C.

Robin Milner coined the slogan that *Well-typed programs cannot go wrong* in an important paper from 1978. This means that if a program is well-typed, then it will never get stuck. This is a property that any static type system should guarantee – but it is not always the case.

In the setting of static types, type safety is defined relative to the semantics of the programming language that we consider.

Let us now assume that type judgements in our type system are of the form $E \vdash e : t$. This is to be read as saying that, given the type environment E that tells us the types of the free variables in e, then the expression e has type t.

Then type safety holds for our type system relative to the semantics if the following two properties hold.

Subject reduction If $E \vdash e : t$ and $e \rightarrow e'$, then we have that $E \vdash e' : t$

Progress If $E \vdash e : t$, then either e is a value or $e \rightarrow$.

The subject reduction property is a preservation property: A well-typed expression will stay well-typed after a computation step.

The progress property guarantees that a well-typed expression cannot go wrong: Either it is fully evaluated or evaluation is able to proceed – it is not stuck.

Taken together, these properties imply a well-typed expression will never get stuck.

4 Type checking and type inference

In the case of static typing, type checking takes place as an early part of the process of parsing and translation. For this reason, a natural requirement of a type system is that type checking is an *effective* notion, which means that there exists an algorithm which can determine whether a program P is well-typed.

Following the pioneering work by Milner, the notion of *type inference* has become increasingly important in this setting . Type inference can be seen as the inverse problem to type checking: given a program P, can we construct a type assignment that will make P well-typed? Implementations of many functional languages, including the ML family and Haskell, incorporate type inference as a central aspect.

If the type system of a programming language supports type inference, then we can allow *implicit typing*, that is, not annotating typable entities in the program with information if we do not want. If we require such type information to always be present, we speak of *explicit typing*. Haskell is an example of a language that allows for both approaches – thanks to type inference.

We would like type checking to be decidable, that is, we want there to exist an algoritm that, when given an E, an expression e and a type t can tell us if $E \vdash e : t$.

If type checking is decidable and our static type system is sound, then we have the situation shown in Figure 1. We are then able to tell if a program is well-typed, but since it is undecidable if a program will get stuck wrt. the property that our type system is supposed to prevent, then there must be programs that are error-free but not well-typed. We call this "unknown zone" the *slack* of the type system. There are always infinitely many programs in the slack (why?).

On the other hand, a sound static type system can change the program development process: We need to spend less time debugging and testing, since we know that some run-time errors cannot happen if our program is well-typed. Instead we must aim to create a well-typed program.

Example 1. In Haskell an example of an expression that is found in the slack of the type system is

```
if 2 + 2 == 4 then "mango" else (not 484000)
```

The two branches of the conditional expression have different types but the first branch is always chosen, since the condition is always true – no run-time error would occur, if Haskell allowed this to evaluate.

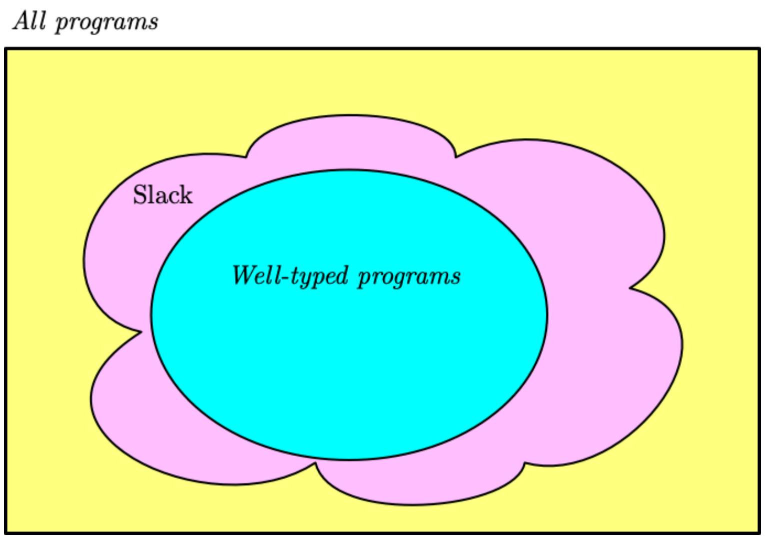

Figure 1: The slack of a type system

5 Pros and cons

I have already outlined the important advantage that sound static type systems will have. But there is still no general agreement as to how one should approach type disciplines. Some people argue that a lack of types has advantages – in that one simply does not need to think about types at all.

Here are some main arguments for static, respectively dynamic typing/no typing.

5.1 Arguments for static typing

Expressiveness Programs that are not well-typed and are rejected by the type-checker tend to be meaningless.

Early warning A static analysis can prevent us from running programs that contain serious errors and help us debug them.

Types as specifications Types describe the behaviour of the program and we can therefore use them as specifications when we develop a program. Some type systems are very expressive and can capture a large class of run-time errors,

Readability A program with type information can be more readable because we make the classification of values explicit.

Homogeneous data structures Data structures must be homogeneous that is, all elements of a data structures must have the same type. This guarantees that operations on data structures cannot go wrong.

Efficiency If one uses a static type system, then type checking also needs to happen once, namely at compile-time, whereas a dynamic type discipline requires the run-time system to carry out type checking every type a program is run.

5.2 Arguments for dynamic typing or no typing

Late warning is better Advocates of dynamic typing often say that the run-time errors that really matter are not type errors at all, so it makes more sense to deal with them at run-time. According to them, it is more useful to have a good programming environment for debugging and testing.

Productivity If one does not need to worry about type errors at compile-time one will spend less time developing a program that can be executed.

Evolution In a dynamically typed setting, it is easy to change changing the type of an entity x. But in a statically typed language, changing the type of the entity $x$may require the programmer to update type annotations for many other entities in the program that depend on x.

Heterogeneous data structures It becomes possible to have heterogeneous data structures, that is, data structures in which not all elements have the same type. This allows one to write simpler code for processing data structures.

Readability Advocates of dynamic typing often say that type information is cumbersome and actually detracts from the readability of a program. The program code gets longer because we have to annotate it with types.

5.3 Dynamic typing is a boring form of typing (??)

Robert Harper, who is one of the designers of Standard ML, is known for saying a dynamically typed language is a statically typed language with only one static type!

He writes [3]

> The characteristics of a dynamic language are the same, values are classified by into a variety of forms that can be distinguished at run-time. A collection of values can be of a variety of classes, and we can sort out at run-time which is which and how to handle each form of value. Since every value in a dynamic language is classified in this manner, what we are doing is agglomerating all of the values of the language into a single, gigantic (perhaps even extensible) type. To borrow an apt description from Dana Scott, the so-called untyped (that is "dynamically typed") languages are, in fact, unityped. Rather than have a variety of types from which to choose, there is but one!

Suppose we had a dynamically typed language where the only values were truth-values and integers. We could then define a type ValueType as

```
data ValueType = TV Bool | IV Integer
```

If an expression in our dynamically typed language is well-behaved, it will always have type ValueType ! In some sense this is the same observation as the idea of having a special type called Dynamic, that we mention in the following section.

6 Combining the static and the dynamic

There are approaches that lie somewhere between traditional static typing and dynamic typin.. One of them is "soft typing", where the idea is to use a static type system to provide advice which the programmer can then choose to ignore.

In such a system, the programmer writes her code as if she were using a dynamically typed language and then uses type inference to try to infer types for the program and detect type errors. If and when such an error is found, the programmer can choose to fix or to let it be and run the program anyway. So here type errors are not errors but simply warnings.

Not everyone holds this approach in high regard. Krishnamurti wrote

> Every few years, yet another scripting language's community seems to discover soft typing. Maybe because I'm the PLT loudmouth, they inevitably end up in a long thread with me. I've been through this with Guile, Python and Tcl. Every time, some smart, bright-eyed and bushy-tailed person sets out to build a soft typer for their language. Every time, a few weeks in, they finally realize why this is such a hideously difficult undertaking, and give up.

A more successful approach is that of *gradual typing* as developed by Siek and Taha [9]. Here one introduces an "unknown" type ? that can be used for annotating entities in a program whose type is not yet known. This form of type system allows the programmer to introduce types into the code gradually, the more she figures out about the intended behaviour of the program that she is writing. If everything is typed with known types, the program will have the same soundness guarantees as a normal static type system would. TypeScript is a gradually-typed version of JavaScript.

7 Dynamic typing in Haskell?

Haskell uses static typing, but there are extensions of Haskell that allow for dynamic typing. The idea of having a special type called Dynamic in an otherwise statically typed language dates back to the work by Abadi et al. [1].

The Dynamic interface provides basic support for dynamic types. Using this library one can inject values of arbitrary type into a dynamically typed value, Dynamic, as well as operations for converting dynamic values into a concrete (non-polymorphic) type.

Peterson describes an approach to dynamic typing in Haskell [6] and Shields et al. [8] describe a single type-inference algorithm that can execute partly at compile time and partly at run-time. This allows Haskell to accommodate types that are only known at run-time.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 237–268. `https://doi.org/10.1145/103135.103138`

[2] Data.Dynamic. base-4.19.0.0: Basic libraries. `https://hackage.haskell.org/package/base-4.19.0.0/docs/Data-Dynamic.html`

[3] Robert Harper. Dynamic Languages are Static Languages. March 2011. `https://existentialtype.wordpress.com/2011/03/19/dynamic-languages-are-static-languages/`

[4] Shriram Krishnamurti. RE: bytecode unification for scripting (and other!) languages. The PLT Scheme mailing list. 15 August 2001, `https://www-old.cs.utah.edu/plt/mailarch/plt-scheme-2001/msg00788.html`

[5] Robin Milner. A theory of type polymorphism in programming, *Journal of Computer and System Sciences*, Volume 17, Issue 3, 1978, Pages 348-375, ISSN 0022-0000, `https://doi.org/10.1016/0022-0000(78)90014-4.`

[6] John Peterson. Dynamic Typing in Haskell. Research Report YALEU/DCS/RR-1022. Yaale University, 2003. `http://www.cs.fsu.edu/~langley/COP4020/2019-Fall/Lectures/tr1022.pdf`

[7] Benjamin Pierce. Types and Programming Languages, MIT Press 2000.

[8] Mark Shields, Tim Sheard, and Simon Peyton Jones. 1998. Dynamic typing as staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98)*. Association for Computing Machinery, New York, NY, USA, 289–302. `https://doi.org/10.1145/268946.268970`

[9] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proceedings of Scheme and Functional Programming Workshop*. ACM, pp. 81–92, 2006.