Type Systems: Basics of Theory and Practice

Clark Jones

Every programming language in the world has a type system of some sort. Many make this system explicit, and many don't. But what is a type system? Broadly, type systems describe the **types** that **values** in a program can take on during a program's execution.

Why is knowing about type systems useful? Consider the case of refueling a car. Filling most cars with diesel is catastrophic, yet it's easy to mix up diesel and gasoline—type systems can prevent these kinds of mistakes, whether at **compile time** or **run time**.

Understanding this Report

We use multiple, ad-hoc variations of Alonzo Church's lambda calculus in this report. The basic calculus consists of variables such as x, y, z, f, and g; applications $(\mathrm{f}\ \mathrm{x})$, $(\mathrm{f}\ \mathrm{g}\ \mathrm{x}) = ((\mathrm{f}\ \mathrm{g})\ \mathrm{x})$; and abstractions $\lambda\mathrm{x}.\mathrm{x}$. Ignoring name conflicts, evaluation proceeds by repeatedly replacing applications like $((\lambda\mathrm{x}.\mathrm{g}\ \mathrm{x})\ \mathrm{f})$ with $(\mathrm{g}\ \mathrm{f})$. Occasionally, math notation is used in a term. For example, $\lambda\mathrm{x}.\mathrm{x} + 2$ is a function that takes a number and returns that number plus two.

The lambda calculus may seem simplistic, but the untyped variant can compute as much as any traditional programming language, including a compiler for the C programming language, shown in Figure 1.

There are a few typographical conventions used in this report. Text in Latin Modern represents **terms**, typically of the lambda calculus, or types. Terms in **bold** are defined in a list at the end of the report; not every instance of a term will be bold.

Implementation Failures

Most type systems are technically impossible to implement, because they make simplifying assumptions. For example, there are infinite real numbers, so computers can't store them all. However, large, incrementally built type systems often fail and admit **unsound** programs. That is, programs that cause bad behavior at runtime that the language and its type system are supposed to prevent.

https://counterexamples.org is a valuable repository of examples of these kind of failures in popular languages [7].

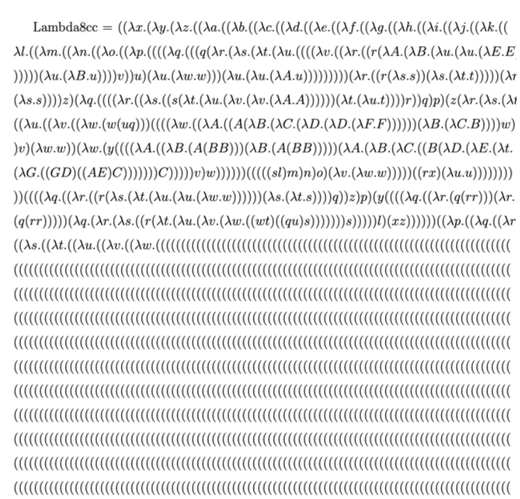

Figure 1: One of the 18,506 pages of lambda-8cc [8].

Theory

But what is a **type**? And how can we determine the types of the **values** in a program before it even runs? Often, beginner programmers are taught **dynamically-typed** languages—they are considered easier to learn because they are more forgiving in what programs they admit. However, we will discuss **static** type systems first.

With static type systems, the types of values are known before the program even runs, and they do not change. To ensure that this is the case, a process known as **type checking** must occur. This process checks that every value conforms to the language's type system. That system may not be formally specified elsewhere, but it always exists.

To understand type checking, let us first consider a simple language that has **functions**, **integers**, and **strings**. If you try to apply a function that takes an integer and returns a string ($\text{integer} \to \text{string}$) to a value that is known to be a string, this would be an error at compile-time. This helps avoid careless mistakes and gives the **compiler** more information to **optimize** a program [1].

However, having to declare the type of every value in our type system would be laborious. It is so simple that every type is obvious just by looking at it. 19 is an integer, “hello” is a string. This is where **type inference** comes in. Type inference allows us to avoid writing **annotations** that tell the compiler the types of the values in our programs. This means that the compiler can automatically figure out that 19 is an integer and “hello” is a string.

Our type system cannot support full inference, though; annotations will sometimes be required. For example, the function $\lambda x.x$, which takes a value and returns it unchanged, can either be of type ($\text{integer} \to \text{integer}$) or ($\text{string} \to \text{string}$). In fact, it can have any number of types, as it could also take functions as well (for example, $(\text{integer} \to \text{integer}) \to (\text{integer} \to \text{integer})$). This is why most static type systems support inference but still require annotations to resolve ambiguity. Later, we will talk about the Hindley-Milner type system, which does support full inference.

Dynamic Type Systems

Not all type systems are static, though. JavaScript, perhaps the most widely used programming language in history, is dynamically typed [2]. However, we can still reason about dynamic type systems in a similar way to static ones. Because a dynamic type system allows values to take on any type at run-time, we can model every value as having a **sum type** [3]; a dynamic version of our simple language would have a **unitype** τ, where $\tau = \text{integer} \mid \text{string} \mid \tau \to \tau$.

Practice

JavaScript and Python are extremely popular programming languages [4]. JavaScript powers complex web applications used by billions daily, and Python is increasingly popular for machine learning applications. Both languages are dynamically typed, which may lead one to believe that static type systems are only for those in the "ivory tower." However, TypeScript, a superset of JavaScript that checks and infers types, is also extremely popular. Python, too, has embraced **gradual typing** with PEP 484 standardizing type hints [5]. It is worth noting that gradual typing is not without its detractors, as shown in Figure 2.

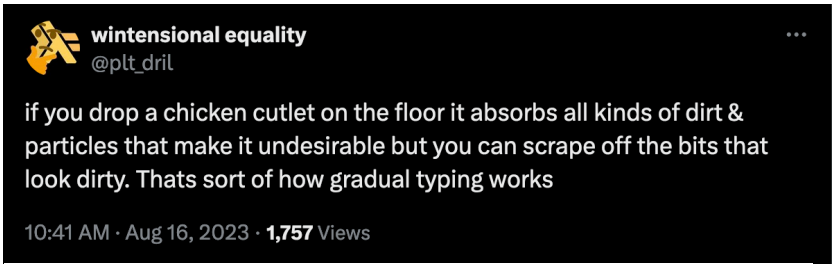

Figure 2: A humorous tweet deriding gradual typing *[9]*.

It is clear from these two mainstream examples that today's developers care more and more about being able to reason about the types in their programs. A significant motivator for this shift is the improved developer tooling made possible by having more explicit types. Catching a wide variety of simple mistakes before a program is even run is valuable, and the continued admiration of the Rust programming language, which has a significantly stronger and more complex type system than TypeScript and Python is more evidence of this [4].

Formal Methods

It makes sense to cover the mainstream first, but no field shows the value of type systems as an analytical tool better than that of formal methods. For most software, failures are a mere annoyance. A laggy or unstable video game is frustrating, but it won't cost lives or enormous investments of time and money.

There are areas, however, where software failures are unacceptable. The most exciting is the National Aeronautics and Space Administration (NASA)'s work. It is not uncommon for NASA missions to cost hundreds of millions of dollars and tens of thousands of person-hours, so they have developed numerous techniques to make more reliable and verifiable software, most famously *The Power of 10* [6].

More grimly, in the medical field, software failures can *kill*. This is not a hypothetical scenario; the software that powered Therac-25, well known in the field of computer science, was error-prone

and delivered horrifically high doses of radiation due to poor design.

Figure 3: The VT100 Terminal that controlled Therac-25 *[10]*

Software is so enormously ubiquitous in today's world that it's easy for most of us to ignore it. However, type systems are the backbone of how we reason about how this software functions. Being able to do this reasoning is useful for developers, sure, but it's important not to lose sight of how incredibly important it is to be able to reason about our software. Imagine if we couldn't understand how other infrastructure, like roads and bridges worked. Software may not be as obviously critical as this physical infrastructure, but in the worst cases, its failures can be just as deadly.

Definitions

- Bold – heavier text that looks **like this**
- Compile time – the time before a program runs
- Run time – while a program is running
- Type – there are many meanings, but it encompasses a possibly infinite set of values, like **integers** or **strings**
- Term – a part of a program's source. Can be literal values like 1 or “dog”, or more complex constructs like $\lambda \text{fx.f x}$.
- Integers – -1, -2, ... and 0, 1, 2 ...
- Strings – a sequence of characters, e.g. “cat”, “orange”, or “”.
- Statically-typed – values have known types at **compile time** that do not change
- Dynamically-typed – values can change at any time in unpredictable ways
- Function – a procedure that turns some number of values into some other number of values. In many type systems, functions are values themselves, and they only take one argument and return one result.
- Optimize – making a program better by some metric; typically speed, but occasionally size or memory usage
- Unsound – wrong in a way that breaks core assumptions in a dangerous way
- Sum type – a type that can be either one type or the other. For example, $\text{a} \mid \text{b}$ can either be an a or a b. It is called a sum because it has as many possible values as its alternatives combined
- Unitype – a single type that all values in a program have

- Gradual typing – a system where **terms** can optionally be annotated with types, allowing a codebase to gradually be converted to use less dynamic type features, which can improve developer tooling

Works Cited

[1] Ben, "Sets, types and type checking," 30 10 2024. [Online]. Available: https://kaleidawave.github.io/posts/sets-types-and-type-checking/. [Accessed 24 11 2024].

[2] MDN, "Grammar and types," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types. [Accessed 7 December 2024].

[3] R. Harper, "Dynamic Languages are Static Languages," 19 March 2011. [Online]. Available: https://existentialtype.wordpress.com/2011/03/19/dynamic-languages-are-static-languages/. [Accessed 7 December 2024].

[4] Stack Exchange, "Technology," 2024. [Online]. Available: https://survey.stackoverflow.co/2024/technology/. [Accessed 7 December 2024].

[5] G. v. Rossum, J. Lehtosalo and Ł. Langa, "PEP 484 – Type Hints," 29 September 2014. [Online]. Available: https://peps.python.org/pep-0484/. [Accessed 7 December 2024].

[6] G. J. Holzmann, "The Power of 10: Rules for Developing Safety-Critical Code," NASA/JPL Laboratory for Reliable Software, June 2006. [Online]. Available: http://web.eecs.umich.edu/~imarkov/10rules.pdf. [Accessed 7 December 2024].

[7] S. Dolan, 8 June 2023. [Online]. Available: https://counterexamples.org/. [Accessed 7 December 2024].

[8] H. Ikuta, "lambda-8cc - An x86 C Compiler Written as an Untyped Lambda Calculus Term," 6 October 2022. [Online]. Available: https://woodrush.github.io/lambda-8cc.pdf. [Accessed 7 December 2024].

[9] @plt_dril, "wintensional equality on X," 16 August 2023. [Online]. Available: https://x.com/plt_dril/status/1691822487409979766. [Accessed 7 December 2024].

[10] ClickRick, "File:Terminal-dec-vt100.jpg," 29 April 2009. [Online]. Available: https://commons.wikimedia.org/wiki/File:Terminal-dec-vt100.jpg. [Accessed 7 December 2024].