Adding a type system to an untyped language

A journey to a type save environment for developers

Christoph Bühler
OST Eastern Switzerland University of Applied Sciences
MSE Seminar "Programming Languages"
Supervisor: Farhad Mehta
Semester: Autumn/Spring 2020

Abstract

Short abstract about why a developer should understand the lambda calculus. What is is how it works and why it is relevant in software engineering and other fields.

Keywords: Lambda-Calculus, type systems, simple types

1 Introduction

All modern programming languages have type systems. They are either of a more dynamic nature - like JavaScript - or statically typed like C#. Functional languages like Haskell have an even stricter form of a type system. They all have one thing in common: They aid the developer to create programs, without the constant fear about runtime errors.

This paper shall give the reader an idea of the steps that are needed to create a type system and how it is applied to an untyped language. This paper will use JavaScript and TypeScript[1] as examples for certain comparisons. Of course, JavaScript is not an untyped language, but it does not have a strict static type analysis which can lead to runtime errors during interpretation and execution of the code. TypeScript is a superset of the JavaScript language which fills this gap and adds a static type analyzer as well as some sort of a compiler (i.e. transpiler) to the language.

But why type systems are helpful and how they work is not a trivial question to be answered. Consider the following code statement:

```
1 foo = "Hello World"
2 bar = 42
3 foo - bar // NaN (Not a Number)
```

Listing 1. Runtime error

As humans, we immediately understand that this statement is not going to terminate well - assuming we have an untyped language. Strings and numbers are not of the same type and cannot be added together. To determine that this is not going to work, the computer needs to execute the statements one by one and will encounter a wrong state. A type system can prevent such errors and create a human-readable message when compiling such a program.

The reader should have an understanding of programming languages and a brief understanding of the untyped λ-Calculus which is described in the first chapters of "Types and Programming Languages" by Benjamin C. Pierce [Pie02].

The following sections will describe the basics of the λ-Calculus, what "types" and "simple types" are and how they can be applied to the λ-Calculus to achieve the "simply typed λ-Calculus".

2 Lambda-Calculus (λ-Calculus)

A computer program can be described in various ways. One very famous variant is the "turing machine" which was defined in a journal [Tur37] by A. M. Turing in the year 1937. The turing machine is fed with instructions and contains a "memory" band to write down results for further computation.

Another famous - but more abstract - method to describe a computation is the λ-Calculus. This system was specified by Alonzo Church [Chu41]. It is a mathematical model of a computation that only contains three rules of operation [Pie02]. Those three operations can be viewed in the following "grammar":

t	::=		*terms*:
	|	x	*variable*
	|	λx.t	*abstraction*
	|	t t	*application*

The untyped λ-Calculus is turing complete, which means it can compute *any* program and therefore can run infinitely. In an untyped λ system, it is possible to search for the successor of "true", which requires the argument to be a number and therefore results in a stuck state.

Since it is not desirable for computer programs to run to infinity, there has to be a way to split up computer programs into two categories: The "useful" and "useless" ones. Any program that will run forever or that will be stuck in an error state counts towards the "useless" ones. Other programs that have valid input and output will be counted to the "useful" programs.

TODO: reference to simons paper about the untyped calculus

For the further progress of this paper, it is necessary to recall the grammar for arithmetic expressions of the untyped calculus [Pie02]:

[1] https://www.typescriptlang.org/

2.1 Terms

t	::=		*terms*:
	|	`true`	*constant true*
	|	`false`	*constant false*
	|	`if t then t else t`	*conditional*
	|	`0`	*constant zero*
	|	`succ t`	*successor*
	|	`pred t`	*predecessor*
	|	`iszero t`	*zero test*

2.2 Values

v	::=		*values*:
	|	`true`	*true value*
	|	`false`	*false value*
	|	`nv`	*numeric value*

2.3 Numeric Values

nv	::=		*numeric values*:
	|	`0`	*zero value*
	|	`succ nv`	*successor value*

3 Types and Simple Types

To get one step closer towards the goal of having a simply typed λ-Calculus we need to define what a type is.

3.1 Types

A type is a classification of a value or multiple values. We typically use mathematical terms to describe a type and the relation of values to their types. When we say "$x : T$", we mean that "x is of type T", or in mathematical terms: "$x \in T$" [Pie02]. This relation makes it possible to determine the type of any computation that correlates with x and can be *statically* analyzed. For example:

$$\begin{cases} \text{if} & x, y \in \mathbb{N} \\ \text{then} & f(x, y) = x + y \in \mathbb{N} \end{cases}$$

We can determine the result type of f without *running* the function. Shown in typescript syntax this could mean:

```
1 const x: number = 1
2 const y: number = 2
3 const f = (x, y) => x + y
4 f(x, y) // result is also of type number
```

Listing 2. TypeScript type inference

With such a possibility, we can rule out stuck or meaningless programs. But how do we get there? We add the typing relation to the grammar and define, that terms and variables must have types:

t	::=		*terms*:
	|	`x : T`	*variable*
	|	`λx : T.t`	*abstraction*
	|	`. . .`	

To express types correctly, we also need new syntactic forms:

T	::=		*types*:
	|	`Bool`	*type of booleans*
	|	`Nat`	*type of natural numbers*

And in addition to the syntactic definition, we need typing rules that define the types of the arithmetic expression on top of the rules that are given by the untyped λ-Calculus.

$$\texttt{true} : \texttt{Bool}$$
$$\texttt{false} : \texttt{Bool}$$
$$\texttt{0} : \texttt{Nat}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : \texttt{T} \quad t_3 : \texttt{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \texttt{T}}$$

$$\frac{t_1 : \texttt{Nat}}{\text{succ } t_1 : \texttt{Nat}}$$

$$\frac{t_1 : \texttt{Nat}}{\text{pred } t_1 : \texttt{Nat}}$$

$$\frac{t_1 : \texttt{Nat}}{\text{iszero } t_1 : \texttt{Bool}}$$

This means, we allocate `true` and `false` to the `Bool` type. Then we assign `0` to the `Nat` type. The derivation rules state that, all `succ` (successors) and `pred` predecessors are of the `Nat` type. The rules for the predecessor and successor define, that the result of the applied `succ` or `pred` function must be of type `Nat`. The rule for the condition defines that the input for the "`if`" must be a boolean value and the result is of type `T`. Both branches of the condition must have the same type.

3.1.1 Typesafety.

> Well-typed programs cannot "go wrong."
>
> – Robin Milner (1978)

The given typing rules give our typing system a pretty important property: *safety* (or *soundness*) [Pie02]. In conjunction with the normalization property [Pie02] [BN98], which eliminates the turing completeness in our system, we can guarantee that our programs that compile successfully with this type system won't ever go wrong. We can call such a program well-typed (i.e. it compiles according to the given typing rules [Car96]).

This *safety* is defined by two theorems [Pie02]:

- *Progress*: Well-typed terms are not stuck. They can take a step in the evaluation rules or are a value.
- *Preservation*: A well-typed term that takes a step in the evaluation rules will yield a result that is also well-typed.

3.2 Intermediate Result

With the given syntax, evaluation rules and type definitions, we would have a type system that could successfully compile the following lines of code (the syntax is inspired by TypeScript for a clear reference to a computer program):

```
const tr: boolean = true
const x: number = 42

if (tr) {
    x
} else {
    x + 1
}
```

3.3 Simple Types

Alonzo Church defined the theory of simple types [Chu40]. In combination with the examples and statements of Benjamin C. Pierce [Pie02], there exists a definition of a simple type. Simple types are a first approach towards a typesafe environment for developers. They contain "base types" (or "value types") like `Bool` and `Nat` (natural numbers) as well as "function types".

Function types are needed to grant the program the possibility to perform computations. Up until now, we can allocate variables to types and can perform an *if* condition.

3.3.1 Base Types. Base types represent unstructured values in a programming language. An incomplete list of such base types one will encounter is:

- Numbers (`Integers` and `Float`)
- Booleans
- Strings (list of `Characters`)

Since base types are unspectacular and are used to calculate other types, literature often substitutes them with a letter for all *unknown* base types [Pie02]. Often, constructs like $\mathcal{A}$ or other letters are seen. For this paper we will establish the following syntactic form from Benjamin C. Pierce [Pie02]:

$$\begin{array}{lcll} \mathsf{T} & ::= & & \textit{types}: \\ & | & \ldots & \\ & | & \mathsf{A} & \textit{base type} \end{array}$$

3.3.2 Function Types. From a theoretical perspective, this language is quite interesting, but to be used as a programming language, it lacks some needed features. For example, we need the possibility to apply a function to some input to generate some output. Otherwise this programming language will be quite boring. To add functions to our typing syntax we add the following line:

$$\begin{array}{lcll} \mathsf{T} & ::= & & \textit{types}: \\ & | & \ldots & \\ & | & \mathsf{T} \rightarrow \mathsf{T} & \textit{type of functions} \end{array}$$

A *type environment* (Γ) is introduced here. This environment (or sometimes called *type context*) is a set with variables and their types [Pie02]. When our type-checker stars, the starting type environment equals "$\varnothing$". With each evaluation step this set will grow and contain the specified values.

Now we have the typing syntax for functions, but we need some additional typing rules to ensure the types of functions can be calculated statically:

$$\frac{\Gamma, x : \mathsf{T}_1 \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \lambda x : \mathsf{T}_1 . t_2 : \mathsf{T}_1 \rightarrow \mathsf{T}_2} \qquad \text{(Abstraction)}$$

This typing rule for the general abstraction evaluation rule of the λ-Calculus TODO: ref to simons paper adds a premise to our system that translates to "if x is of type T_1 and is in our typing context Γ and the term t_2 is of type T_2, then the abstraction $\lambda x : T_1 . t_2$ has the type $T_1 \rightarrow T_2$".

Also an additional typing rule for variables and applications are needed:

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}} \qquad \text{(Variable)}$$

"The type that is assumed for x is in the set of Γ".

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_{11} \rightarrow \mathsf{T}_{12} \quad \Gamma \vdash t_2 : \mathsf{T}_{11}}{\Gamma \vdash t_1 t_2 : \mathsf{T}_{12}} \qquad \text{(Application)}$$

"If t_1 is a function that takes a T_{11} and returns a T_{12} and the term t_2 is a value of type T_{12} then the result of the application of t_1 to t_2 will be of type T_{12}".

Translated into a programming language:

```
// number (Variable)
const x = 1337

// number => string (Abstraction)
const f = nr => nr.toString()

// string (Application)
const r = f(x)
```

4 Application of simple types on the λ-Calculus

Describe which steps are needed to apply the concept of simple types on the lambda calculus.

The result is "simply typed lambda calculus".

4.1 Function Types
4.2 Base Types
4.3 Unit Type
4.4 Pairs, Tuples, Records
4.5 References
4.6 Exceptions

With those possibilities, a variety of concepts can be created. While there exists the concept of "exceptions" for example, it does not support subtyping of any kind. Polymorphism is part of a higher version of a type system, for example System F.

TODO: reference to marcs paper about systemF

5 Usage and Relevancy

- describe the usefulness of the calculus
- describe where it is used (ref to dev ops / kubernetes maybe?)
- why it is relevant (for system F / system T among others)

6 Conclusion

Give the conclusion over the paper. What was shown, why it was shown.

What should the reader know about the calculus now and how the reader can translate that into typed languages

References

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Car96] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, March 1996.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6).* Princeton University Press, 1941.

[Ham18] Gary Hammock. Latex listings - javascript & es6. https://github.com/ghammock/LaTeX_Listings_JavaScript_ES6, 2018. Accessed: 2020-10-10.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, MA, USA, 2002.

[Tur37] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.