Simply typed Lambda-Calculus for developers

An introduction to the topic

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

1 Introduction

All modern programming languages have type systems. They are either of a more dynamic nature - like JavaScript - or statically typed like C-Sharp. Even if we talk about functional languages like Haskell, they have a type system that helps the developer to create programs.

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

```
foo = "Hello World"
bar = 42
foo + bar //?
```

Listing 1. Untyped language

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. To understand a type system, it is necessary to use maths to explain how we can check and derive types. Such a mathematical system is the λ-Calculus.

The reader should have an understanding in programming languages and a brief understanding of the untyped λ-Calculus which is described in the first chapters of [Pie02].

The following chapters will define some knowledge about the untyped λ-Calculus and then introduce the reader to the simply typed λ-Calculus.

2 Lambda Calculus

A computer program can be described in multiple ways. A. M. Turing defined the turing machine in [Tur37] in the year 1937. This machine was an abstract model of computation. In contrast to a "machine", the mathematical model of a computer program was defined by Alonzo Church [Chu41]. It uses mathematical logic to run calculations and all elements can be described in mathematical terms.

The untyped λ-Calculus is turing complete, which means it can compute *any* program and therefore can run infinitely. This is a contrast to the simply typed calculus, which limits the executable terms in the way that they never reach an erroneous state. 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, some basic method has been introduced [Chu40] to check for typing errors and restrict malformed input from beeing inserted into a function during compile time.

TODO: reference to simons paper

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

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*

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

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

3 Simply Typed Lambda-Calculus

The mathematical abstraction of the λ-Calculus is a topic that is not easy to understand if encountered for the first time. The purpose of the present paper is to describe the nature of the typed calculus with examples from programming languages so that a developer can understand the concepts.

3.1 Key Difference

The simply typed calculus relies upon static type analysis. This means, to determine if a program is *well-typed* (i.e. it compiles according to the given typing rules [Car96]), no machine needs to run the program.

The limitation of the input can eliminate stuck states and in conjunction with the normalization property [Pie02] [BN98] therefore eliminates turing completeness.

A program that successfully compiled in a simply typed lambda system is guaranteed to run to completion and not reaching any stuck states.

3.2 Types

Types in the mathematical language are not the same as a developer might expect. A developer sees a type as *bool* or *string*, while a mathematician defines types as set of elements.

3.3 Types in the λ-Calculus

To enhance the given grammar with types so that our program cannot take expressions like *if succ(0) then true else false*, some typing restriction must be in place. As seen above,

```
if <something boolean-ish> then
    // then statement
else
    // else statement
end if
```

Listing 2. If Condition

To describe this mathematically, we need to define what "boolean-ish" is.

T	::=		*types*:
	|	Bool	*type of booleans*

Our typing context ("T") now contains the *Bool* type and the list of values ("v") contains the values for the *true* and *false* constants. We need to add typing rules to be able to parse this boolean expression:

true : Bool
false: Bool
t1: bool t2: T t3: T
——————————
if t1 then t2 else t3 : T

This means that the condition after the *if* must be a Bool and the two branches of the condition must have the same type.

To make this system more useful, we add natural numbers as well with their corresponding type "Nat":

T	::=		*types*:
	|	...	
	|	Nat	*type of natural numbers*

And of course we need some additional typing rules:

0 : Nat
t1 : Nat
———
succ t1: Nat
t1 : Nat
———
pred t1: Nat
t1 : Nat
———
iszero t1: Bool

Describe the different types and their context. What are function types and how are they used. What are the properties of typing.

Ref to Curry/Church

typescript example of church notation with typings.

3.4 Extending types

- What hides behind "base types"
- the unit type (void)
- other typing constructs for the simple calculus

3.5 References

- What is a reference type

3.6 Error states

- What exactly is an exception
- How to throw / handle them

4 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)

5 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.

[lat15] Tex stack exchange: Derivation tree for type evaluations. https://tex.stackexchange.com/questions/279021/derivation-tree-for-type-evaluations/279295#279295, 2015. Accessed: 2020-09-29.

[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.