Type Theory, Type Systems & Type Rules

MMA130 - Mathematical logic for computer science

Sebastian Lindgren

February 17, 2016

Abstract

This work will provide some of the history of type systems. It will show how to use a technique similar to natural deduction to check the type of an expression. By using this technique one can formally prove if a program has the correct types at the proper places. These is used by compilers to check programming languages. It is an important skill to learn for computer scientists.

1 Introduction

Type theory is a fascinating subject with it's roots in formal logic. It started in 1870 with the origins of formal logic and has been evolving through the ages. In the late 1950s type checkers mostly checked if the type was a integer or a floating value. As time passed more complex checks were introduced, checking data types and polymorphic values etc. [1]

1.1 Type theory - formal logic to type checkers

In mathematic logic type theory is a class of formal systems. In it each term has a type and operations are restricted to the type. Type theory is closely related to type systems. These type systems can be used to formally prove that a program will use the right types. [1]

2 Different type systems

To formally check if the program use the correct types one can use natural deduction. By formulating the functions in that form one can follow the rules to see if a variable is of the correct type. [5]

There are several different kinds of type checkers, those that check statically and those that do it dynamically. The statical checks are made during the compilation, for example in programming languages as C, C++, Java, etc. There are those that do it dynamically during runtime. This leads to slower run times but is safer if a type error occurs. The program can then exit cleanly with an exception. [1]

Modern languages, for example Java, use a variety of dynamic typing called, soft typing. This means that it can cast the type to whatever fits the best depending on the runtime information. Other languages with this power is Clojure, Common Lisp and Cython. These languages use a optional statical syntax with which the programmer can decide which type is wanted with additional annotations. [4]

3 How to formally check type correctness

In type theory a type rule is a inference rule. This works much like in natural deduction. [2] These rules may be applied to show that a program is well typed.

3.1 The notation:

The general form is stated as:

$$\frac{\Gamma 1 \vdash e : \tau 1 \ \ldots \ \Gamma n \vdash e : \tau n}{\Gamma \vdash e : \tau}$$

Here "e" is the expression. τ is the type and Γ the typing environment. [3]

3.2 Using the natural style type rules in a small language

If we have a language containing

e ::= n $\in \mathbb{N}$ | b $\in$ {true | false} | e + e | e || e | e == e

Then we can create rules defining different functions that can be used on the language.

Normally there is a typing environment Γ with all the defined variables, in this minimalistic language there is no need for it. There is however a type t which can be int or bool. The rules below are named after their ordering.

1. The int function: $\frac{}{n:int}$ meaning that n has type int
2. The bool function $\frac{}{b:bool}$ meaning that b has type bool
3. equal to function $\frac{e1:bool\ e2:bool}{e1||e2:bool}$
   Meaning that if e1 and e2 has type bool then the function e1||e2 has type int.
4. equal to function $\frac{e1:int\ e2:int}{e1+e2:int}$
   Meaning that if e1 and e2 has type int then the function e1+e2 has type int.
5. equal to function $\frac{e1:t\ e2:t}{e1==e2:bool}$
   Meaning that if e1 and e2 has type t then the function e1==e2 has type bool.

With these rules we can for example evaluate simple type derivation trees such as that for: 1+2+3.

$$\frac{}{(1+2)+3}$$

Here we can see that to get to know which type this has we have to use rule 4 on both sides.

$$\frac{\frac{}{1+2:int} \quad \frac{}{3:int}}{(1+2)+3}$$

We can then use rule 4 on the upper left and rule 1 on the upper right.

$$\frac{\frac{\frac{}{1:int} \quad \frac{}{2:int}}{1+2:int} \quad \frac{}{3:int}}{(1+2)+3}$$

We can then use rule 1 on both sides on the upper left and we see that all types are the correct ones. We conclude that we have type int on the bottom expression.

$$\frac{\frac{\frac{}{1:int} \quad \frac{}{2:int}}{1+2:int} \quad \frac{}{3:int}}{(1+2)+3:int}$$

This concludes the demonstration.

4 Discussion

In this work we have seen the usefulness of type checking with the aid of natural deduction style type rules. We have also seen have to formally prove if a program has the correct types. This can be used by programmers to improve their programs. It can also be used when creating own languages or designing programs.

For these reasons type theory and learning to read type rules are important. To improve the understanding of ones programs, increase efficiency, detect errors and provide language safety.

5 Referenses

1 - Types and programming languages, Benjamin C. Pierce, The MIT Press, London, England, 2002.
2 - https://en.wikipedia.org/wiki/Natural_deduction
3 - https://en.wikipedia.org/wiki/Type_rule
4 - https://en.wikipedia.org/wiki/Type_theory
5 - Twenty-five Years of Constructive Type Theory Proceedings of a Congress Held in Venice, October 1995