

Part I

Compilers and languages

Chapter 1

Programming languages

Computers are devices which read a well-defined, finite sequence of simple instructions and emit a result. In theoretical analysis of computers, models have been developed to understand and prove properties. A finite sequence of instructions fed to a computer is called an *algorithm*, which is the language of high level computation[3]. In modern encodings of algorithms or programs, “high level” languages are used instead of the computational models. Such languages are then translated into instructions, that often are much closer to a computational model. The process of translating programs into computer instructions is called *compiling*, or *transpiling* if the program is first translated into another “high level” language.

For the purpose of this dissertation, a simple programming language has been implemented to illustrate the concepts in detail. The language transpiles to *untyped lambda calculus*. For the remainder, the language will be referred to as *L*.

1.1 Untyped lambda calculus

The *untyped lambda calculus* is a model of a computer, developed by Alonzo Church[1]. The untyped lambda calculus is a simple tangible language, of just three types of terms.

$$x \tag{1.1}$$

$$\lambda x.E \tag{1.2}$$

$$YE \tag{1.3}$$

The first component, is that of the *variable* Equation 1.1. A variable is a reference to another lambda abstraction. Equation 1.2 shows a lambda *abstraction*, which contains a bound variable x and another lambda term E . Finally in Equation 1.3, *application*. Application can be interpreted as substituting the variable in the left *abstraction* Y , with the right term

E . Let Y be $\lambda x.T$ and E be z , then YE can be substituted for $(\lambda x.T)z$. Furthermore, substitute x for E or z , such that Y becomes $T[x := y]$, read as “Every instance of x in T , should be substituted by y ”.

The untyped lambda calculus is in fact, turing complete; any algorithm that can be evaluated by a computer, can be encoded in the untyped lambda calculus. The turing completeness of the untyped lambda calculus, can be derived using Church encoding, which is an encoding of lambda terms that can express boolean and arithmetic expressions[2]. For the remainder of the dissertation, ordinary arithmetic expressions are written in traditional mathematics. The expressiveness and simplicity of lambda calculus, makes it an excellent language to transpile to, which in fact, is a common technique.

1.2 High level programming languages

High level languages associated with lambda calculus are often also very close to it. The L language is very close to the untyped lambda calculus, for instance see two equivalent programs, that both add an a and a b .

$$(\lambda add.E)(\lambda a.(\lambda b.a + b)) \tag{1.4}$$

Listing 1.1: Add function

```
fun add a b = a + b;
```

Notice here that the lambda calculus variant must bind the function name “outside the rest of the program” or more formally in

Bibliography

- [1] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.
- [2] Alonzo Church. *The calculi of lambda-conversion*. Number 6. Princeton University Press, 1985.
- [3] B Jack Copeland. The church-turing thesis. 1997.