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 refeered to as L.

1.1 Untyped lambda calculus

The *untyped lambda calculus* is a model of a computer, developed by Alanzo 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 1.3. A variable is a reference to another lambda abstraction. Equation 1.2 shows a lambda *abstraction*, which is 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 $(\lambda x.T)z$. Furtermore, 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 exellent language to transpile to, which in fact, is a common technique.

1.2 Translation to lambda calculus

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. 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;
```

1.2.1 Scoping

Notice that Equation 1.4, must bind the function name "outside the rest of the program" or more formally in an outer scope. In a traditional program such as Listing 1.2, functions must be explicitly named to translate as in the above example.

Listing 1.2: A traditional program

```
fun add a b = a + b;
fun sub a b = a - b;
sub (add 10 20) 5;
```

Listing 1.3: An order dependant program

```
fun sub a b = add a (0 - b);
fun add a b = a + b;
sub (add 10 20) 5;
```

Notice that there are several problems, such as, the order of which functions are defined may alter whether the program is correct or not. For instance, the program defined in Listing 1.3 would not translate correct, it would

translate to Equation 1.5. The definition of sub, or rather, the applied lambda abstraction, is missing a reference to the *add* function.

$$(\lambda sub.(\lambda add.(sub\ (add\ 10\ 20)\ 5))\ (\lambda a.(\lambda b.a + b)))\ (\lambda a.(\lambda b.add\ a(0 - b))) \tag{1.5}$$

lambda lifting is a technique where *free variables*, are explicitly parameterized. A *free variable*, is a variable in respect to some function f that is referenced from within f, but defined outside. This is exactly the problem in Equation 1.5, which has the lambda lifted solution Equation 1.6.

$$(\lambda sub.(\lambda add.(sub\ add\ (add\ 10\ 20)\ 5))\ (\lambda a.(\lambda b.a + b)))\ (\lambda add.(\lambda a.(\lambda b.add\ a(0 - b)))) \tag{1.6}$$

As it will turn out, this will also enables complicated behaviour, such as *mutual recursion.*

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.