Aspects of efficiency in functional programming languages

by

Samuel Valdemar Grange

supervised by

Prof. Kim Skak Larsen

UNIVERSITY OF SOUTHERN DENMARK
INSTITUTE OF MATHEMATICS AND COMPUTER SCIENCE

Masters thesis in Computer Science

Contents

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 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 by enriching the language, with properties such as recursion with the Y-combitator and 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, Equation 1.4 and Listing 1.1, 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

```
1 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

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

Listing 1.3: An order dependant program

```
1 fun sub a b = add a (0 - b);
2 fun add a b = a + b;
3 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[4]. 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*.

Moreover, lambda lifting also conforms to "traditional" scoping rules. *Variable shadowing* occurs when there exists $1 <$ reachable variables of the same name, but the "nearest", in regard to scope distance is chosen. Effectively, other variables than the one chosen, are *shadowed*. Variable shadowing is an implied side-effect of using using lambda calculus. Convince yourself that the function f in Listing 1.4, yields 12.

Listing 1.4: Scoping rules in programming languages

```
1 let x = 22;
2 let a = 10;
3 fun f =
4   let x = 2;
5   a + x;
```

1.2.2 Recursion

Complexity - "The state or quality of being intricate or complicated"

Reductions in mathematics and computer science are one of the princiral methods used developing beautiful equations and algorithms.

Listing 1.5: Infinite program

```
1 fun f n =
2   if (n == 0) n
3   else if (n == 1) n + (n - 1)
4   else if (n == 2) n + ((n - 1) + (n - 2))
5   ...
```

Listing 1.5 defines a function f, that in fact is infinite. Looking at the untyped lambda calculus, there are not any of the three term types that define infinite functions or abstractions, at first glance. Instead of writing an infinite function, the question is rather, how can a reduction be peformed on this function, such that it can evaluate *any* case of n?

Listing 1.6: Recursive program

```
1 fun f n =
2     if (n == 0) n
3     else n + (f (n - 1))
```

Listing 1.6 defines a recursive variant of f, it is a product of the reduction in Equation 1.7.

$$n + (n-1) \cdots + 0 = \sum_{k=0}^{n} k \tag{1.7}$$

But since the untyped lambda calculus is turing complete, or rather, if one were to show it were, it must also realize algorithms that are recursive or include loops, the two of which are equivalent.

$$(\lambda f.E)(\lambda n.\texttt{if}(n == 0)(n)(n + (f(n-1)))) \tag{1.8}$$

The naive implementation of a recursive variant, will yield an unsolvable problem, in fact, an infinite problem. In Equation 1.8, when f is applied recursively, it must be referenced. How can f be referenced, if it is "being constructed"? Substituting f with its implementation in Equation 1.9, will yield the same problem again, but at one level deeper.

$$(\lambda f.E)(\lambda n.\texttt{if}(n == 0)(n)(n + ((\lambda n.\texttt{if}(n == 0)(n)(n + (f(n-1))))(n-1)))) \tag{1.9}$$

One could say, that the problem is now recursive. Recall that lambda lifting, is the technique of explicitly parameterizing outside references. Convince yourself that f lives in the scope above its own body, such that, when referenceing f from within f, f should be parameterized as in Listing 1.7, translating to Equation 1.10.

Listing 1.7: Explicitly passing recursive function

```
1 fun f f n =
2     if (n == 0) n
3     else n + (f f (n - 1))
```

$$(\lambda f.E)(\lambda f.(\lambda n.\texttt{if}(n == 0)(n)(n + (f\ f\ (n-1))))) \tag{1.10}$$

The initial invocation of f, must involve f, such that it becomes $f\ f\ n$. The *Y-combinator* in Equation 1.11, is the key to realize that the untyped

lambda calculus can implement recursion. Languages with functions and support binding functions to parameters, can implement recursion with the Y-combinator.

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{1.11}$$

Implementing mutual recursion is an interesting case of lambda lifting and recursion in untyped lambda calculus.

Listing 1.8: Mutual recursion

```
1 fun g = f;
2 fun f = g;
```

Notice in Listing 1.8 that g requires f to be lifted and f requires g to be lifted. If a translation "pessimistically" lifts all definitions from the above scope, then all required references exist in lexical scope.

Languages have diffirent methods of introducing recursion, some of which have very different implications, especially when considering types. For instance, OCaml has the `let rec` binding, to introduce recursive definitions. The `rec` keyword indicates to the compiler that the binding should be able to "see itself",

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.

[4] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.