Aspects of efficiency in functional programming languages

by

Samuel Valdemar Grange

supervised by

Prof. Kim Skak Larsen

UNIVERSITY OF SOUTHERN DENMARK

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Master's 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 refered to as L.

1.1 Untyped lambda calculus

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

$$x \tag{1.1}$$

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

$$YE \tag{1.3}$$

The first term is that of a *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. A bound variable is one explicitly parameterized in the abstraction. In traditional programming languages bound variables are parameters to a function, whereas *free* variables are variables that exist outside of the function scope. It is easy to see that variables are either free or bound. Free varables are determined by Equation 1.4, Equation 1.5 and Equation 1.6.

$$Free(x) = \{x\} \tag{1.4}$$

$$Free(\lambda x.E) = Free(E) \backslash \{x\} \tag{1.5}$$

$$Free(YE) = Free(Y) \cup Free(E) \tag{1.6}$$

Finally, in Equation 1.3, *application*. An application of two terms can be interpreted as substituting the variable in the left abstraction Y with the right term E. It is also common to introduce the *let binding* to lambda calculus, which will be done when introducing typing in **??**.

Example 1.1.1. Let Y be $\lambda x.T$ and E be z, then YE is $(\lambda x.T)z$. Furtermore, substituteing x for E, such that Y becomes $T[x := E]$. Since $E = z$, substitute E for z, such that $T[x := z]$, read as "Every instance of x in T, should be substituted by z".

Remark 1.1.1. Substituting lambda terms is a popular method of evaluateing lambda calculus programs. Languages like Miranda, Clean and general purpose evaluation programs like the G-machine **??**, implement *graph rewriting*, which will be introduced in **??**

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 realized by modelling numerics, boolean logic and recursion with the *Y-combitator*. Church encoding, is the encoding of numerics, arithmetic expressions and boolean logic [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 1.7 and Listing 1.1, that both add an a and a b.

$$(\lambda add.E)(\lambda a.(\lambda b.a + b)) \tag{1.7}$$

Listing 1.1: Add function

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

Notice that in Equation 1.7 the term E is left undefined, E is "the rest of the program in this scope". If the program was to apply 1 and 2 to add, directly below in the high level representation (Listing 1.2) the lambda calculus equivalent would look like Equation 1.8.

$$(\lambda add.add\ 1\ 2)(\lambda a.(\lambda b.a + b)) \tag{1.8}$$

Listing 1.2: Add function applied

```
fun add a b = a + b;
add 1 2;
```

1.2.1 Scoping

Notice that Equation 1.7, 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 1.3, functions must be explicitly named to translate as in the above example.

Listing 1.3: A traditional program

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

Listing 1.4: 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 1.4 would not translate correct, it would translate to Equation 1.9. The definition of *sub*, or rather, the applied lambda abstraction, is missing a reference to the *add* function.

$$(\lambda sub.(\lambda add.(sub\ (add\ 10\ 20)\ 5))\ (\lambda a.(\lambda b.a+b)))\ (\lambda a.(\lambda b.add\ a(0-b))) \tag{1.9}$$

lambda lifting is a technique where free variables (section 1.1), are explicitly parameterized [6]. This is exactly the problem in Equation 1.9, which has the lambda lifted solution Equation 1.10.

$$(\lambda sub.(\lambda add.(sub\ add\ (add\ 10\ 20)\ 5))\ (\lambda a.(\lambda b.a+b)))\ (\lambda add.(\lambda a.(\lambda b.add\ a(0-b)))) \tag{1.10}$$

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 1.5, yields 12.

Listing 1.5: Scoping rules in programming languages

```
let x = 22;
let a = 10;
fun f =
  let x = 2;
  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 1.6: Infinite program

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

Listing 1.6 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 1.7: Recursive program

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

Listing 1.7 defines a recursive variant of f, it is a product of the reduction in Equation 1.11.

$$n + (n-1) \cdots + 0 = \sum_{k=0}^{n} k \tag{1.11}$$

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 in expressiveness.

$$(\lambda f.E)(\lambda n.\mathtt{if}(n == 0)(n)(n + (f(n-1)))) \tag{1.12}$$

The naive implementation of a recursive variant, will yield an unsolvable problem, in fact, an infinite problem. In Equation 1.12, 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 1.13, will yield the same problem again, but at one level deeper.

$$(\lambda f.E)(\lambda n.\mathtt{if}(n == 0)(n)(n + ((\lambda n.\mathtt{if}(n == 0)(n)(n + (f(n-1))))(n-1)))) \tag{1.13}$$

One could say, that the problem is now recursive. Recall that lambda lifting (subsection 1.2.1), 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 1.8, translating to Equation 1.14.

Listing 1.8: Explicitly passing recursive function

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

$$(\lambda f.E)(\lambda f.(\lambda n.\mathtt{if}(n == 0)(n)(n + (f\ f\ (n-1))))) \tag{1.14}$$

The initial invocation of f, must involve f, such that it becomes $f\ f\ n$. The *Y-combinator*, an implementation of a fixed-point combinator, in Equation 1.15 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.

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{1.15}$$

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

Listing 1.9: Mutual recursion

```
fun g = f;
fun f = g;
```

Notice in Listing 1.9 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" (**??**).

1.3 High level abstractions

The lambda calculus is a powerful language that can express any algorithm. Expressiveness does not necessarily imply ergonomics or elegance, in fact encoding moderately complicated algorithms in lambda calculus becomes quite messy. Many high level techniques exist to model abstractions in tangible concepts.

1.3.1 Algebraic data types

Algebraic data types are essentially a combination of disjoint unions, tuples and records. Algebraic data types are closely related to types thus require some type theory to fully grasp. Types are explored more in depth in **??**.

Listing 1.10: List algebraic data type

```
type List a =
    | Nil
    | Cons a (List a)
;
```

Listing 1.10 is an implementation of a linked list. The list value can either take the type of `Nil` indicating an empty list, or it can take the type of `Cons` indicating a pair of type a and another list. The list implementation has two type constructors and one type parameter. The type parameter a of the list algebraic data type defines a *polymorphic type*; a can agree on any type, it is universally quantified $\forall a.$ `Cons` a `(List` a`)`. The two type constructors `Nil` and `Cons` both create a value of type `List a` once instatiated.

Listing 1.11: List instance and match

```
let l = Cons 1 (Cons 2 (Cons 3 Nil));

match l
    | Nil -> 0;
    | Cons n _ -> n;
;
```

Once a value is embedded into an algebraic data type such as a list it must be extractable to be of any use. Values of algebraic data types are extracted and analysed with *pattern matching*. Pattern match comes in may forms, notably it allows one to define a computation based on the type an algebraic data type instance realizes (Listing 1.11).

Scott encoding

Pattern matching strays far from the simple untyped lambda calculus, but can in fact be encoded into it. The *scott encoding* (Equation 1.16) is a technique that describes a general purpose framework to encode algebraic data types into lambda calculus [8]. Considering an algebraic data type instance as a function which accepts a set of "handlers" allows the encoding into lambda calculus. The scott encoding specifies that type constructors should now be functions that are each parameterized by the type constructor parameters $x_1 \dots x_{A_i}$ where A_i is the arity of the type constructor i. Additionally each of the type constructor functions return a n arity function, where n is the cardinality of the set of type constructors. Of the n functions, the type constructor parameters $x_1 \dots x_{A_i}$ are applied to the i'th "handler" c_i. These encoding rules ensure that the "handler" functions are provided uniformly to all instances of the algebraic data type.

$$\lambda x_1 \dots x_{A_i}.\lambda c_1 \dots c_n.c_i x_1 \dots x_{A_i} \tag{1.16}$$

Example 1.3.1. The `List` algebraic data type in Listing 1.10 has two type constructors, `Nil` with the type constructor type Equation 1.17 and `Cons` with the type constructor type Equation 1.18. Equation 1.17 is in fact also the type of `List` once instantiated, effetively treating partially applied functions as data.

$$b \to (a \to \texttt{List}\ a \to b) \to b \tag{1.17}$$

$$(a \to \texttt{List}\ a \to b) \to b \to (a \to \texttt{List}\ a \to b) \to b \tag{1.18}$$

Listing 1.12: List algebraic data type implementation

```
fun cons x xs =
    fun c _ onCons = onCons x xs;
    c;

fun nil =
    fun c onNil _ = onNil;
    c;
```

Encoding the constructors in L yields the functions defined in Listing 1.12. Pattern matching is but a matter of applying the appropriate handlers. In Listing 1.13.

Listing 1.13: Example of scott encoded list algebraic data type

```
let l = cons 1 (cons 2 (cons 3 nil));

fun head x _ =
    x;

l head 0;
```

Efficiency can be a bit tricky in lambda calculus as it is at the mercy of implementation. A common method of considering efficiency is counting *β-reduction* since they evaluate to function invocations. The β-reduction is a substitution which substitutes an application where the left side is an abstraction in witch the bound variable is substituted with the right side term (Equation 1.19).

$$\beta_{red}((\lambda x.T)E) = T[x := E] \tag{1.19}$$

It should be clear that invoking a n arity function will take n applications. In the case of a scott encoded algebraic data types the largest term in regard to complexity is either the size of the set of "handler" functions or the "handler" function with most parameters. The time to evaluate pattern match is thus $O(\max_i(c_i + A_i))$.

Chapter 2

Typing and validation

Automatic validation is one of many reasons to use computers for solving various tasks including writing new computer programs. Spellchecking is a very common and trivial instance of an input validation algorithm. The equivalent for computer programs could be type checking; the problem of validating a programmers intuition of a program's intent.

2.1 Types and validation

The spellchecking equivalent for computer programs could be type checking; the problem of validating a programmers intuition of a program's intent. Types also have other properties than simply validating they can in fact be related to thorems to which an implementation is the proof [5].

Listing 2.1: Head implementation

```
fun head l: List a -> a =
    match l
        | Cons x _ -> x;
        | Nil -> ?;
    ;
```

For instance consider the implementation of type `List a -> a` in Listing 2.1, a total implementation of the function cannot exist.

The type scheme for the L language will be the Hindley-Milner type system [4, 7].

2.1.1 The language of types

Before delving into types the lambda calculus defined in section 1.1 must be agumented with the *let expression* (Equation 2.1).

$$\texttt{let } x = Y \texttt{ in } E \tag{2.1}$$

It should be noted that the let binding can be expressed by abstraction and application (Equation 2.2).

$$(\lambda x.E)(Y) \tag{2.2}$$

The let expression has a nice property that will become apparent later when typing rules are introduced. The untyped lambda calculus exists at the "value level" while the now introduced syntax exists at the "type level". There are two vairants of types in Hindley-Milner, the *monotype* and the *polytype*. A monotype is either a variable, an abstraction of two monotypes or an application (Equation 2.3).

$$mono\ \tau = a \,|\, \tau \rightarrow \tau \,|\, C\tau_1 \ldots \tau_n \tag{2.3}$$

The application term of the monotype is dependant on the primitive types of the programming language. In L the set of monotypes are $\{\texttt{Int}, \texttt{Bool}, \texttt{ADT}\}$. The types $\tau_1 \ldots \tau_n$ are the typevariables required to construct some atomic type. An instance of which could be a `List a` wich one type parameter `a`. A polytype is a polymorphic type (Equation 2.4).

$$poly\ \sigma = \tau \,|\, \forall a.\sigma \tag{2.4}$$

A central component of typing in Hindley-Milner is the *environment*. The environment list Γ is a list of pairs of variable and type (Equation 2.5). $\Gamma \vdash x : \sigma$ implies a *typing judgment* meaning that given Γ the variable x has type σ.

$$\Gamma \ = \epsilon \,|\, \Gamma, x : \sigma \tag{2.5}$$

Like in the untyped lambda calculus types also have notions of free and bound type varables. Varables are bound when they have been introduced by a quantification or exist in the environment set.

$$free(a) = \{a\} \tag{2.6}$$

$$free(C\tau_1 \ldots \tau_n) = \bigcup_{i-1}^{n} free(\tau_i) \tag{2.7}$$

$$free(\Gamma) = \bigcup_{x:\sigma\in\Gamma} free(\sigma) \tag{2.8}$$

$$free(\forall a.\sigma) = free(\sigma) - \{a\} \tag{2.9}$$

$$free(\Gamma \vdash x : \sigma) = free(\sigma) - free(\Gamma) \tag{2.10}$$

2.1.2 Hindley-Milner rules

With the now introduced primitives the Hindley-Milner type system is but a set of inference rules composed by said primitives. There are six rules in the

$$\text{Var}\ \frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \qquad \text{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\text{Abs}\ \frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \qquad \text{Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \ \texttt{let}\ \ x = e_1\ \texttt{in}\ \ e_2 : \tau}$$

$$\text{Ins}\ \frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \qquad \text{Gen}\ \frac{\Gamma \vdash e : \sigma \qquad a \notin free(\Gamma)}{\Gamma e : \forall a.\sigma}$$

Figure 2.1: Hindley-Milner type rules

Hindley-Milner rules outlined in Figure 2.2. The first rule and also the only axiom is the Variable. The Variable rule states that if some variable x with type σ has been deemed to exist, then they must be in the environment. The Application rule states that if $e_1 e_2$ is of type τ_2 then e_1 must conform to a type that can produce a type τ_2 given a type τ_1 and e_2 must conform to the type of this τ_1. The Instatiate rule are important to specify a quantified type σ_1 to a specific one σ_2. Generalization lifts a type into a quantified type for all types which are bound.

2.1.3 Damas-Milner Algorithm W

$$\text{Var}\ \frac{x:\sigma \in \Gamma \qquad \tau = inst(\sigma)}{\Gamma \vdash x : \tau, \emptyset} \quad \text{Abs}\ \frac{\tau_1 = fresh \qquad \Gamma, x : \tau_1 \vdash e : \tau_2, S}{\Gamma \vdash \lambda x.e : S\tau_1 \rightarrow \tau_2, S}$$

$$\text{App}\ \frac{\Gamma \vdash e_1 : \tau_1, S_1 \quad \tau_3 = fresh \qquad S_1\Gamma \vdash e_2 : \tau_2, S_2 \quad S_2 = unify(S_2\tau_1, \tau_2 \rightarrow \tau_3)}{\Gamma \vdash e_1 e_2 : S_2\tau_3, S_3 S_2 S_1}$$

$$\frac{\Gamma \vdash_W e_0 : \tau, S_0 \qquad S_0\Gamma,\ x : \overline{S_0\Gamma}(\tau) \vdash_W e_1 : \tau', S_1}{\Gamma \vdash_W \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau', S_1 S_0} \ [\texttt{Let}]$$

Figure 2.2: Hindley-Milner type rules

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] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[5] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

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

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[8] Dana Scott. A system of functional abstraction, 1968. lectures delivered at university of california, berkeley. *Cal*, 63:1095, 1962.