Compilation of non-strict functional programming languages

Harald Steinlechner

haraldsteinlechner@gmail.com

Abstract. In non-strict programming languages as *Haskell* recursive and infinite data-structures come out of the box and can therefore be modelled in a natural manner. *Lazy-Evaluation*, an implementation technique which realizes non-strict semantics became very popular because its clever execution model automatically caches values and maximizes sharing within the execution. At first glance however the execution model seems to be quite expensive and not particularly well suited for compilation. In this paper we survey some important compilation techniques for lazy-evaluation and give an outlook on recent developments in this area.

1 Introduction

In programming languages as *C* and *Java* function arguments get evaluated before the actual function call occurs. Interprocedural optimization aside, this evaluation takes place even if the argument is not actually used within the callee. Also functional languages as *LISP* [1] and *Standard ML* [2] work this way, which is often referred to as *call-by-value*. Semantically functions are *strict* in its argument which means divergence in one of its arguments leads to divergence of the program itself. Alternatively *non-strict* languages like *Miranda* [3], *Lazy ML* [4] and *Haskell* [5] reduce functions before evaluating its parameters.

```
-- function, which returns 1 no matter what argument provided
const1 :: Int -> Int
const1 _ = 1

-- diverging function (endless loop)
bomb = bomb
```

While in languages like ML `const1 bomb` diverges, in Haskell for example the program will happily print `1` due to the fact that `bomb` was not required to compute `const1`. This behaviour is consistent with the ordinary lambda calculus, i.e. β-reduction holds. Note, that due to the nature of conceptually replacing formal parameters by its arguments equivalent expressions might be evaluated more than once, which is clearly not desirable. This evaluation strategy is often referred to as *call-by-name*[1].

[1] Notation is quite subtle here—*call-by-name* works like *normal-order* reduction known from lambda calculus but does not evaluate inside the body of an unapplied function, i.e. is not *stronly* normalizing.

Call-by-need, which can be seen as an optimization to ordinary call-by-name, overcomes this efficiency issue by caching already evaluated expressions. Expressions are either never evaluated, or exactly once. The Glasgow Haskell Compiler [6] (`ghc`) realizes *call-by-need* with its implementation of *lazy evaluation* called graph reduction. Moreover `ghc` implements *full-laziness*, that is sharing of common sub-expressions is maximized which means the reduction of expressions is optimal respective to reduction-steps [7, 8].

Non-strict languages support infinite datastructures in a vary natural way. In Haskell the inifite stream $2^n,\ n >= 0$ may be defined as:

```
[2^n | n <- [0..]]
```

The given expression of course diverges due to its infinite formulation. However it works as expected if just a finite number of elements is requested by the program:

```
take 5 [2^n | n <- [0..]]
> [1,2,4,8,16]
```

With lazy evaluation simple algorithms may be composed for example by using function composition without computing to much in each building block [9].

```
sort :: [Int] -> [Int]
sort = ... -- a clever written sort algorithm

minimumElement :: [Int] -> Int
minimumElement = head . sort -- sorts a list and takes
                             -- the first element of it (which is min)
```

Note that here, `minimumElement` has worst-case runtime $O(n)$ due to laziness[2].

In *strict* programming languages binding an expression to a variable (`let`-expressions) not just means to bind the variable but also to evaluate the expression. In lazy languages evaluation and binding is decoupled in source code [8] which means that the binding is lazy and its value is computed only if absolutely required.

Given all these benefits lazy evaluation unfortunately suffers from unpredictable run time. More seriously *space-leaks* pose problems in practice. Due to the delaying expression evaluation as long as possible, big expressions get accumulated—even if the result is finally needed and these expressions could have been executed immediately in the first place. The following example uncovers the severity of space leaks:

[2] Full example and proof can be found at http://apfelmus.nfshost.com/articles/quicksearch.html.

```
fSum :: (Int, Int) -> Int
fSum (0, accum) = accum
fSum (n, accum) = fSum (n-1,f n + accum)
```

`fSum` computes $f(0) + f(1).. + f(n)$ while its implementation seems efficient due to tail-recursive formulation the program performs awful. It requires $O(n)$ stack-space which quickly exceeds available stack budget.
$fSum(n, 0)$ for some f reduces as follows:

$$\begin{aligned}
&\Rightarrow\ fSum(n-1, f(n)+0)\\
&\Rightarrow\ fSum(n-2, f(n-1)+(f(n)+0))\\
&\Rightarrow^*\ fSum(0, f(1)+f(2)+..+f(n)+0)\\
&\Rightarrow\ f(1)+f(2)+..+f(n)+0..
\end{aligned}$$

This reduction scheme looks simple but reveals to be inefficient—a naive implementation therefore seems impractical. Although each partial sum can be reduced to a single value each iteration conceptually pushes another unevaluated expression onto the stack. If the final result is needed by the caller of the function `fSum` (which is the hole point of the function call) the evaluation finally reduces the huge tail of unevaluated computations. An efficient implementation of lazy-evaluation therefore has to deal with this issue. Aside from this our implementation should employ sharing of common expressions and implement other features like pattern matching and higher-order-functions—efficient code generation seems out of reach.

To sum up we have seen persuasive arguments for lazy evaluation, but its implementation seems rather inefficient and tricky at first glance. In fact early implementations of lazy evaluation differ substantially with conventional compiler technology but very active research made lazy evaluation reasonable fast even on non-specialized hardware. In this paper we will show some important implementation techniques to implement non-strict programming languages with focus on compilation techniques.

The remainder of this paper is organized as follows. Section 2 gives a high level description of the compilation process and its immediate languages. Section 3 starts with an interpreted approach and develops different compilation approaches. Section 4 summarizes recent developments and points out other possible implementation techniques

2 Overview

Especially in functional languages the λ-calculus plays an important role. In this paper we will use an enriched version of the lambda calculus similar to GHCs *Core* language [6] as input language:

```
<exp> ::= <constant>                    -- Built-in constants
      |   <variable>                    -- Variable names
      |   <exp> <exp>                   -- Applications
      |   Lam <variable> <exp>          -- Abstractions
      |   Let <variable> = <exp>        -- Binding
          In <exp>
      |   If <exp> Then <exp> Else <exp>
      |   Plus, Minus, Not...           -- Built-in functions
```

Note that `Let a = b In c` is syntactic sugar for `(Lam x c) b` and is therefore not necessary but simplifies the translation and makes code more readable. Structured types and pattern matching should be of no concern here.

3 Implementation of Lazy-Evaluation

When reducing λ-Expressions to normal-form, an efficient β-reduction is crucial.

$$(\lambda x.b)a \Rightarrow b[x/a]$$

Naive interpretative implementations of functional languages directly work as rewriting system [10]. Rewriting alone, however is not efficient due to its interpretative nature. Furthermore these approaches merely implement call-by-name instead of call-by-need i.e. sharing is not employed at all.

3.1 Graph-Reduction

Similarly to abstract syntax trees we choose to use graphs as program representation whereby each node corresponds to production in our grammar (@ denotes application and λ corresponds to *Lam* production in the grammar given before).

For example the β-reduction $((\lambda x.Not\ x)\ True) \Rightarrow Not\ True$ is represented as follows[3]:

Of course this seems rather straightforward—the argument $True$ is copied to all occurrences of the formal parameter x. The argument may be large and copying thus is inefficient. More severely we have achieved no sharing between multiple occurrences of the argument. Now instead of copying the argument over into the body of the function only the pointers of the nodes are updated (that is, substituted). This is called *graph-reduction* and first was introduced by Waddsworth [11].

[3] example taken from [7]

The principle of pointer substitution is shown in Figure 1 representing the reduction

$$(\lambda x.\ And\ x\ x)\ (Not\ \ True) \Rightarrow And\ (Not\ \ True)\ (Not\ \ True)$$

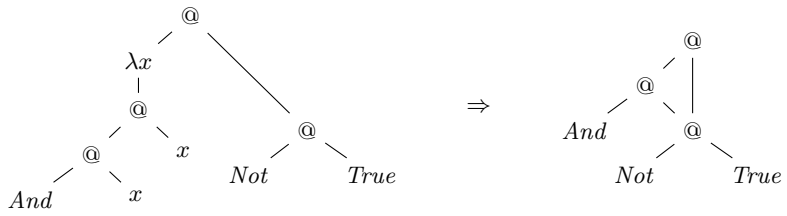

Fig. 1: Pointer substitution

Using graphs and physical overriding of nodes (pointer substitution) β-reduction with sharing can be implemented effectively. Further details are given for example by Jones [7].

3.2 Towards a compiled implementation

Graph-Reduction as shortly presented in the previous section works well for interpreters. We will now look into code generation techniques realizing graph-reduction. From an abstract point of view codegeneration for lambda expressions is faced with the problem of generating fixed code sequences for each lambda body.

Lets now consider the simple example:

$$\lambda x.(\lambda y.x + y)$$

If applied with an argument z we instantiate a new $\lambda y.z+y$ which makes clear that it in general it is not possible to generate a single fixed code sequence for each lambda expression. The core problem are free variables in lambda expressions. One possibility to solve this issue is to introduce an environment object which holds free variables similarly to Landins SECD machine [12, 13]. Although a valid approach for the implementation of free variables, it is not clear how to handle sharing properly with explicit environments.

Expressions without free variables are called *combinators*. For combinators there is no need to handle free variables which makes them especially suitable for code generation. Limiting expressions to combinators seems quite restrictive—however it turns out that each closed expression with free variables can be transformed into a combinator. The next chapters provide algorithms to transform arbitrary expressions into combinators.

3.3 Compilation with SKI-Combinators

Turner utilizes *Combinatory-Logic* instead of working with λ-calculus directly [14]. Syntactically terms in combinatory logic are solely built of applications of three[4] basic combinators: S, K, I being as:

$$S\ f\ g\ x = f\ x\ (g\ x) \quad \textbf{(S)}$$
$$K\ x\ y = x \quad \textbf{(K)}$$
$$I\ x = x \quad \textbf{(I)}$$

Each λ-expression can be translated into a term in combinatorial logic (and vice versa) [15]. According to *church-turing-thesis* each computable function can be therefore be expressed in terms of SKI-Combinators. Turner [14] utilizes this equivalence to implement the SASL [16] language by using a transformation to SKI-Combinators terms instead of performing lambda-lifting. Formally transformation and reduction rules are given as:

$$S\ f\ g\ x \Rightarrow f\ x\ (g\ x) \quad \textbf{(S-Reduction)}$$
$$K\ x\ y \Rightarrow x \quad \textbf{(K-Reduction)}$$
$$I\ x \Rightarrow x \quad \textbf{(I-Reduction)}$$
$$\lambda x.\ x \rightarrow I \quad \textbf{(I-Transformation)}$$
$$\lambda x.\ c \rightarrow K\ \ c \quad \textbf{(K-Transformation)}$$
$$\lambda x.\ e_1 e_2 \rightarrow S(\lambda x.\ e_1)(\lambda x.\ e_2) \quad \textbf{(S-Transformation)}$$

Given this, a simple compilation algorithm for λ-expressions e can be formulated as:

```
while e contains a lambda abstraction do
  abstr := chose any lambda within e
  if abstr.body is Application
     Apply_S_Transformation(e)
  else if abstr.body is Variable
     Apply_I_Transformation(e)
  else Apply_K_Transformation(e) // must be constant
```

The following example demonstrates the transformation of a λ-expression to combinatorial logics:

[4] in fact S and K are sufficient, since (Ix) may be expressed as $((SKK)\ x)$ which is extensionally equivalent

$$
\begin{aligned}
& (\lambda x. + x\ x)\ 5 \\
\rightarrow^S\ & S\ (\lambda x. + x)(\lambda x.x)\ 5 \\
\rightarrow^S\ & S\ (S\ (\lambda x.+)\ (\lambda x.x))\ (\lambda x.x)\ 5 \\
\rightarrow^I\ & S\ (S\ (\lambda x.+)\ I)\ (\lambda x.x)\ 5 \\
\rightarrow^I\ & S\ (S\ (\lambda x.+)\ I)\ I\ 5 \\
\rightarrow^K\ & S\ (S\ (K\ +)\ I)\ I\ 5
\end{aligned}
$$

To verify the application of the transformation rules we use reduction rules given before to reduce the expression to normal-form.

$$
\begin{aligned}
& S\ (S\ (K\ +)\ I)\ I\ 5 \\
\Rightarrow\ & S\ (K\ +)\ I\ 5\ (I\ 5) \\
\Rightarrow\ & K\ +\ I\ 5\ (I\ 5) \\
\Rightarrow\ & +\ (I\ 5)\ (I\ 5) \\
\Rightarrow\ & +\ 5\ (I\ 5) \\
\Rightarrow\ & +\ 5\ 5 \\
\Rightarrow\ & 10
\end{aligned}
$$

A simple abstract machine implements reduction of these terms. As an example consider the reduction of the S combinator as shown in Figure 2.

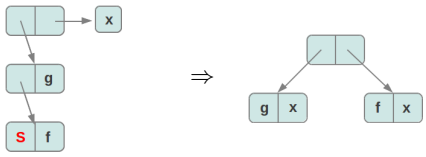

Fig. 2: Reduction of $S\ f\ g\ x \Rightarrow f\ (g\ x)$. Each Application-Node is represented as cell with two elements.

The resulting machine is very simple, the set of constants (combinators and builtins) is small and each reduction can be implemented efficiently. For compilation each top-level function is transformed to terms in `SKI`-form, which contain no free variables by definition. In his work Turner suggested to generate machine code for these terms but did not provide an appropriate translation scheme. Although his interpretative implementation seemed to be competitive with *LISP*-machines at that time, his approach suffers from fundamental problems regarding the granularity of computation.

Each step is very small which has negative effect on the program size and overall performance. By adding additional ‘shortcut‘ combinators, code size becomes quite manageable but this comes at cost of fast compilation times due

to a number of extra rules for compilation. Another issue comes from the for humans rather obscure program representation. From a practical point of view programs in `SKI` are harder to debug because of their huge difference to original programs.

3.4 Compilation with Supercombinators

Hughes as well works with combinators, however instead of eliminating variables completely a variant of *lambda-lifting* [17] is employed to eliminate free variables yielding combinators. The proposed algorithm works on more coarse grained expressions. In fact each lambda lifted user-defined function is used as a combinator, called *supercombinator* [18].

A supercombinator S of arity n is a lambda expression of the form:

$$\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$$

where E is not a lambda abstraction such that:

- S has no free variables
- any lambda abstraction in E is a supercombinator
- $n \geq 0$, that is, there need to be no λ's at all.

As an example of transforming a λ-expression into supercombinators, let's consider the λ-expression:

```
(Lam x . Lam y . x + y) 4 5
```

Each λ body has free variables. $\lambda.x + y$ is no combinator but we can make it into one by abstracting the free variable which yields:

```
(Lam w y . w + y)
```

In the original program this gives:

```
(Lam x . (Lam w y . w + y) x) 4 5
```

Next we give the supercombinator a name, say $\$Y$

```
Let $Y w y = w + y
In (Lam x . $Y x) 4 5
```

The remaining expression is itself a combinator—and again we give it a name.

```
Let $Y w y = w + y
In
  Let $X x = $Y x
  In $X 4 5
```

Hughes introdues the notion of *full-laziness* which means each subexpression is indeed evaluated only at most once. Transformation rules must be chosen carefully in order to preserve full laziness [18]. Notably, it turned out that Turners `SKI` approach also implements full laziness.

In 1984 Johnson introduced an influential abstract machine realizing lazy evaluation—the *G-Machine* [19] which is the basis for *Lazy-ML* [4] and other implementations like the *Spineless-Tagless-G-Machine (STG)* [20]. Similarly to supercombinators user-defined functions are transformed into combinator-form. Notable, for simplicity the original compilation scheme used by Johnson does not employ full laziness. The codegenerator generates a machine-coded combinator interpreter from each program. This means each program is transformed into an intermediate form, called the *G-Machine-code*. Afterwards the stacks of the abstract machine is simulated, constant values evaluated and machine code with fixed or stack-relative addresses emitted.

The semantics of the G-Machine is fairly complex and a full description is out of scope of this article. However we will briefly describe some principles in order to motivate other machines (as the STG) and further optimizations. In the G-Machine cells may be of four kinds:

1. application
2. cons-Cel
3. primitive type like `int`
4. n-ary application of a supercombinator

In general, call sites as well as arguments are unknown. Combinator X for example may be called with another (unevaluated) application, or with a primitive value. In the abstract machine the `EVAL` instruction evaluates a closure. Depending on the type of the argument different code has to be executed. This is usually realized via a tag stored with each cell. Eval instructions subsequently execute specific reduction code depending of this tag. Usually this is realized via indirect jumps. More precise description of code generation can be found in [4].

4 Optimizations and recent developments

The Spineless Tagless G-Machine (STG) [20] improves the G-Machine in various ways. From performance-point-of-view most importantly:

- Unboxed values are supported directly.
- Implementation of pattern matching is supported directly and therefore very efficient.
- Unevaluated suspensions and values (both called closures) have the same representations. Instead of inspecting a tag which dispatches on the state of the closure an indirect branch executes appropriate code. After evaluation the code pointer is modified to point to code which returns the cached value directly.

4.1 Semi-Tagging and dynamic pointer tagging

On modern hardware indirect branches are very expensive. In consequence programs compiled by the Glasgow Haskell Compiler suffered from bad branch-prediction behaviour. Recently Marlow et al. proposed to *Semi-tagging* which breaks the tagless nature of the STG [21]. The optimization is based on the observation that on 32bit architectures the 2 least significant bits[5] are unused due to the fact that closures are always *word-size* aligned which is ensured by the runtime-system. The STG now by default enters each closure even if it is a value, and could be used directly in the caller. The key idea of the optimization is now to encode the status of the closure into the closure pointer directly. This way expensive indirect branches are no longer required for evaluated thunks. According to their benchmarks this optimization in the execution model reduces almost 50% of the cache misses which results in 10-14% improvement at runtime.

4.2 Optimistic evaluation

Even tough modern lazy-evaluation implementations have highly optimized code-generation and runtime systems, lazy-evaluation involves constant execution overhead. Furthermore space behaviour can be quite unpredictable and a major performance killer (as noted in Section 1). Strictness analysis overcomes this issue by automatically marking arguments as strict if they are required in any case (for any continuations). If marked so, the compiler may perform transformations to reduce the constant overhead imposed by lazy-evaluation. It is known that strictness analysis in general is incomplete, i.e. not all possibly strict parts of the programs can be determined statically [22, 23]. Recent research tried to execute closures speculatively [8, 24]. Each expression is assigned a specific time budget for execution—if execution terminates within that budget the value is produced immediately and no suspended value (thunk) must be generated.

We note that no prior optimistic evaluation has been proposed in the setting of dynamic compilation.

5 Conclusions

Non-strict semantics nicely supports infinite data-structures in high-level functional programming languages. Unfortunately its evaluation model is very different from conventional programming languages and therefore performs poor on stock hardware when implemented naively. We have shown a brief overview of important implementations techniques, optimizations thereof and a short overview on recent developments in this area.

References

1. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, Part I. Commun. ACM **3**(4) (April 1960) 184–195

[5] 3 least significant bits on 64bit architectures

2. Milner, R., Tofte, M., Harper, R., Macqueen, D.: The Definition of Standard ML - Revised. Rev sub edn. The MIT Press (May 1997)
3. Turner, D.A.: Miranda: A non-strict functional language with polymorphic types. In: FPCA. (1985) 1–16
4. Augustsson, L.: A compiler for lazy ML. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming. LFP '84, New York, NY, USA, ACM (1984) 218–227
5. Peyton Jones, S., et al.: The Haskell 98 Language and Libraries: The Revised Report. Journal of Functional Programming **13**(1) (Jan 2003) 0–255 http://www.haskell.org/definition/.
6. Marlow, S., Jones, S.P.: The Glasgow Haskell Compiler. In: The Architecture of Open Source Applications. Volume II. Independent (May 2012) 67–88
7. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
8. Ennals, R.J.: Adaptive evaluation of non-strict programs. Technical report (2004)
9. Hughes, J.: Why functional programming matters. Comput. J. **32**(2) (April 1989) 98–107
10. Augustsson, L.: Lambda-calculus cooked four ways (2000)
11. Wadsworth, C.P.: Semantics and pragmatics of the lambda-calculus. PhD thesis, Programming Research Group, Oxford University (1971)
12. Landin, P.J.: The Mechanical Evaluation of Expressions. The Computer Journal **6**(4) (January 1964) 308–320
13. Henderson, P.: Functional programming - application and implementation. Prentice Hall International Series in Computer Science. Prentice Hall (1980)
14. Turner, D.A.: A new implementation technique for applicative languages. Softw., Pract. Exper. **9**(1) (1979) 31–49
15. Turner, D.A.: Another algorithm for bracket abstraction. Journal of Symbolic Logic **44**(2) (1979) 267–270
16. Turner, D.: SASL Language Manual. (1976)
17. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations, Springer-Verlag (1985) 190–203
18. Hughes, R.J.M.: Super-combinators a new implementation method for applicative languages. In: Proceedings of the 1982 ACM symposium on LISP and functional programming. LFP '82, New York, NY, USA, ACM (1982) 1–10
19. Johnsson, T.: Efficient compilation of lazy evaluation. In: SIGPLAN NOTICES. (1984) 58–69
20. Jones, S.L.P.: Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5. Journal of Functional Programming **2** (1992) 127–202
21. Marlow, S., Yakushev, A.R., Peyton Jones, S.: Faster laziness using dynamic pointer tagging. In: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. ICFP '07, New York, NY, USA, ACM (2007) 277–288
22. Mycroft, A.: The Theory and Practice of Transforming Call-by-need into Call-by-value. In Robinet, B., ed.: Symposium on Programming. Volume 83 of Lecture Notes in Computer Science., Springer (1980) 269–281
23. Sekar, R., Ramakrishnan, I.V., Mishra, P.: On the power and limitations of strictness analysis. J. ACM **44**(3) (May 1997) 505–525
24. Ennals, R.: Adaptive Evaluation of Non-Strict Programs. PhD thesis, King's College (2003)