Lambda: final report

Mike Burrell (burrellm@cpsc.ucalgary.ca)
Supervisor: Robin Cockett (robin@cpsc.ucalgary.ca)

April 26, 2004

Abstract

A complete implementation of a functional language is covered briefly. Components of the implementation, such as parser, lambda lifter, flattener, pattern-matcher, and abstract machine, will be covered in an attempt to convey the state of the art and the problems in implemented these. The subject of typing will be discussed in greater detail due to the novelty of the implementation used here.

1 Introduction

The project covers a complete implementation of a new functional programming language Lambda. The project was initially to research program transformations and, specifically, deforestation, that has not happened. Unexpected discoveries, especially surrounding typing, have arisen and will take on the main focus of the paper. Because deforestation is still an attainable goal, a short introduction to deforestation is provided in appendix A.

Complementary to this document is the implementation document[Bur04]. That document covers the implementation in detail; this document is simply a survey.

Functional languages derive directly from the lambda calculus, a universal computational model and offer its advantages. An attraction of the lambda calculus comes from its simplicity: a “program” is a term, which is either a variable, an abstraction (or “function”), or an application between two terms. A computation is then repeated reduction of a term: an application between an abstraction and another term is conceptually a call of a function.

Because functional programs are free from side-effects and low-level manipulations, they make good candidates for transformations and optimizations. They also offer opportunity for research, despite their maturity.

The implementation of a functional language, as a rule, is the composition of three broad functions:

1. syntax-checking, parsing and transforming into a parse tree;
2. semantic-checking and transformation into a more efficient form;
3. compilation to an abstract machine.
Items #2 and #3 have the most potential for research, especially research specific to functional programming. The semantic-checking and transformation comprises the bulk of this project, as the abstract machine which was used was rudimentary. However, the description of the abstract machine, to a large degree, dictates the transformations which are needed.

The novel research in this project is in the semantic-checking and specifically in the type-checking. Since the introduction of Hindley-Milner typing [Mil78], functional languages have used static second-order typing appropriately. Issues in typing are still being resolved, as is shown by the need for a class system to Haskell [HHJW96].

The algorithm discussed in this paper solves the problem of under-generalizing types in conventional type-checking algorithms. Regarding types as more specific, while still being correct, offers the capability for work whole bodies of semantically-correct programs.
2 Implementation

The language, in brief, is a pure functional language — i.e., with no side effects — and evaluated lazily. Disallowing any side effects allows the transformations to become much simpler, and as the language is for research purposes only, side effects are not needed. Input is done via command-line arguments, and output is done simply by pretty-printing the evaluated expression to standard output.

Another important feature of the language worth mentioning in introduction is its call-by-need reduction strategy (or "lazy evaluation"). This point is discussed in further depth in section 2.5. As the language is "pure", this does not affect any semantics, but it is pertinent to the abstract machine.

2.1 Syntax

The syntax is based largely on Haskell's syntax, except that it is much more free-form: whitespace is insignificant, and constructors for a common type need not be grouped together. Consider the following (poorly-styled!) example:

A :: Z.
x =
% A => 0
| (B x) => x * 2
| (C (a, b)) => a' + b' where { a' = x a.
b' = x b }.
B :: Int => Z.
C :: (Z, Z) => Z in
x

The type Z has three constructors (A, B, C), which are defined apart from one another. Further, a' and b' defined in the where clause are written with poor white-space aesthetics.

2.2 Lambda lifting

Lambda lifting is a process whereby closures are “lifted” out of their contexts. For example, consider the function:

\[ f = \% \, n \, \Rightarrow \, \text{map addn} \, \text{where} \, \{ \, \text{addn} = \% \, x \, \Rightarrow \, n + x \, \} \]

The function addn makes use of the variable n, which is bound at a higher scope (specifically, it is bound by the function f). When represented in an internal form, embedded abstractions, as addn is, are cumbersome. The process of lambda lifting lifts embedded abstractions up to the top level, and thus become simple functions. In this example, lambda lifting would give the resultant program:

\[ \text{addn'} = \% \, n \, x \, \Rightarrow \, n + x. \]
\[ f = \% \, n \, \Rightarrow \, \text{map addn} \, \text{where} \, \{ \, \text{addn} = \text{addn'} \, n \, \} \]
Note that addn now has no abstractions in it; all abstractions have been lifted to the top level.

Algorithms for lambda lifting come from Peyton Jones and Lester[JL92]. The process of lambda lifting basically involves passing in all free variables as arguments. Refinements to the lambda lifting process involve detecting some redundant lifts, and implementing recursion with lifted abstractions more efficiently. Research into speeding the process of lambda lifting itself is ongoing, as described in [DS02].

2.3 Flattening

Flattening is not a process required by all implementations. However, it will provide simplified data structures and some simplified algorithms, as in type-checking. The process of flattening involves removing all where clauses. When used with lambda lifting, it ensures that every symbol is declared at the top level, or bound in an abstraction.

\[
f = \% \ n \rightarrow \text{map addn where } \{
  \text{addn} = \% \ x \rightarrow \ n + x
\}
\]

When lifted and flattened, the above program becomes:

\[
\text{addn}' = \% \ n \ x \rightarrow \ n + x.
\]

\[
f = \% \ n \rightarrow (\% \ \text{addn} \rightarrow \text{map addn}) \ (\text{addn}' \ n)
\]

Flattening can easily lead to a quite messy expression for even slightly complex programs.

2.4 Pattern-matching

In order for a program to be compiled for a reasonably efficient abstract machine, patterns must be transformed. Specifically, cases must be built which have exactly one branch for each constructor of a type. Where patterns are nested, this process can be complex.

For each case or abstraction, a list of discriminants is built. When a discriminant offers a chance to distinguish between constructors of a type, a “simple” case statement is built which enumerates the constructors.

For this stage, it is important that there be an exhaustive coverage of all constructors of a type. If there is not, an error is raised.

2.5 Reductions

Reducing, or evaluating, the program is the final step in completing the implementation. It’s also the most straight-forward. Simple weak head normal form reduction strategies will be used.

The key to lazy evaluation is to pass in references to computations in place of actual values. In this way, a minimum number of reductions is performed to complete the computation.
The naïve approach is to make multiple copies of the term being applied. However, where a bound variable is non-linear in an expression, computation of the same expression occurs more than once. So, a reference to an expression is used: when that expression is evaluated, the reference is updated to point at the computed value, and the expression is computed only once, no matter how many times the variable occurs in the expression.
3 Type-checking

The type-checking system offered in the implementation of Lambda is truly novel. While it does build on conventional algorithms where collecting type equations[Mil78] and unifying[Vis97], it offers greater correctness in the typing than in other systems.

A simple example is the following Haskell code:

\[\begin{align*}
g &= () \\
f \; \quad xs &= g \; xs \\
f \; (x:xs) \; (y:ys) &= f \; ys \; xs \\
z &= f \; [1, \; 2] \; "ab"
\end{align*}\]

This code does not type-check in Haskell, claiming that \( f \) is of type \([a] \rightarrow [a] \rightarrow \text{O}\). The reason is that, conventionally, a call graph is constructed when deriving type information. Every call to a function which is in the same strongly-connected component is considered part of definition of the types of that component. In this case, the call to \( f \; ys \; xs \) is considered part of the definition of the type of \( f \), and hence the interpretation is that the type of \( xs \) is equal to the type of \( ys \).

Consider the analogous Lambda code:

\[\begin{align*}
g &= \% \; \_ \Rightarrow () \\
f &= \\
\% \; \quad xs \Rightarrow g \; xs \\
\_ \; (x:xs) \; (y:ys) \Rightarrow f \; ys \; xs \\
in \\
f \; [1, \; 2] \; "ab"
\end{align*}\]

This code will pass type-checking and, more to the point, gives \( f \) the more correct typing of \([a] \rightarrow [b] \rightarrow \text{O}\). The reason is that calls to functions are always considered at-least-as-specific, but not necessarily strictly equal to the type of the function being referenced.

One negative point to the algorithm, which needs stating, is that because unification happens with all components of the call graph at once, type-checking is incredibly slow. Solutions are being considered for future work.

3.1 Collecting type-equations

As with any Hindley-Milner typing system, collecting equations consists of trying to construct a proof from a few rules which will satisfy the typing of the final equation. From this proof we get a list of equations of type substitutions which must be satisfied for the typing to be considered consistent.

To the set of equations we add a set of inequalities. The inequality \( A \rightarrow B \) reads “\( A \) is at least as general as \( B \)”. This equation is produced when trying to project against a function. In other words, when we have the inference \( f : F \vdash f : A \), where \( f \) is a function, we complete the proof as \( \vdash F \vdash f : A \).

3.2 Unification

Unification of a mixture of equalities and inequalities form a complex process. Like simple unification in other implementation, the crux of it is to consume an equality
and return a series of substitutions. However, to ensure consistency with respect to inequalities, each time a substitution is performed on an inequality, the inequality is broken up such that a type variable is on the left-hand-side of the "\( \Rightarrow \)" at all times. This very often results in creating new type equalities.

Further, to ensure termination, consistency checking must happen at every substitution, to ensure that cyclic inequalities — e.g., \( A \Rightarrow S(B), B \Rightarrow C, C \Rightarrow A \) — will not lead to endless generation of equalities.
4 Conclusion and further directions

In this paper we have looked briefly at the issues involved in implementing a complete functional programming language. Also, a novel approach to type-checking which offers very correct typing was discussed.

Though the implementation is largely complete in terms of correctness, there are many ways in which development can, and will, continue. First, research into deforestation, as originally planned, could be done. A brief overview of deforestation is provided in appendix A. Secondly, as the type-checking seems the most exciting result of the project, work on creating a more efficient type-checking system seems an obvious extension.

As the abstract machine is somewhat simplistic and inefficient, consideration must be given to translating to another abstract machine, such as that used by the Charity programming language.
A Deforestation

Optimizations on functional languages in any capacity requires transforming expressions on a rather high level. Programming in functional languages involves the creation and use of combinators to build up complex expressions. This is the benefit to functional programming: extreme flexibility and code re-use; however, it is also the major problem with functional programming in that performance suffers. Combinators build up data structures only to have them destroyed immediately. Even in lazy implementations where memory consumption is not an issue, there are time issues in executing constructors.

The major transformation in this project is deforestation. Deforestation removes instances of combinators in an effort to remove constructors and remove performance. Here we introduce an example function ss:

\[ \text{ss n = fold (+) 0 (map square (range 1 n))} \]

ss takes a single integer argument \( n \) and evaluates to the sum of all the squares between 1 and \( n \). Here range evaluates to a list of integers between its two arguments; square evaluates to the square of its argument; map applies its first argument to each element of its second argument; and fold is a fold as from the lambda calculus, with its first two arguments replacing the cons and nil constructors.

Stepping through a naive evaluation of ss \( n \): range produces a list; map consumes that list and produces another list; fold consumes that list and produces the integer which is the result. This is precisely the problem: two lists have been produced, though none is useful. The aim of the transformation process, for this example, is to remove the construction of these two lists. Through deforestation, this can be transformed to:

\[ \text{ss n = f 0 1 n where} \]
\[ f s a b = \]
\[ \text{if a > b then} \]
\[ s \]
\[ \text{else} \]
\[ f (s + \text{square a}) b \]

In this case no lists are produced, yet the semantics of the function are entirely equivalent to those before the transformation.

The concept of deforestation came from Phil Wadler[Wad90] and arose largely out of his work with listlessness, another program transformation designed to reduce the number of constructions. Where listlessness reduced the number of lists produced, deforestation was an attempt to work with general tree structures as described by any recursively defined type. Unfortunately, early deforestation was not applicable to general program structures.

Wadler’s student, David Marlow, began working on deforestation. In 1992, they released a conference paper[MW92], which generalized deforestation but did not guarantee termination. In 1995, Marlow completed his Ph.D. thesis[Mar95], which introduced knot-tying as a means to guarantee termination. It also provided generality over the 1992 paper in allowing deforestation of higher-order expressions.

Since then, small advances have been made on Marlow’s thesis. Notable among these are Hamilton’s 1995 paper[Ham95], which guaranteed termination with slightly
fewer restrictions on the generality of expressions. Seidl and Sørenson in 1997[SS97] offered an even more general algorithm by creating higher-order treeless terms and a constraint system.
References


