Defunctionalization of Typed Programs

Jeffrey M. Bell and James Hook*
Oregon Graduate Institute of Science & Technology †

August 25, 1994

Abstract

In a 1972 paper, Reynolds outlined a method for removing higher-order functions from a functional language program by representing functional values as data and interpreting the representations as needed. This transformation is known as *defunctionalization*. This paper describes a defunctionalization algorithm extended to operate over a significant portion of core Standard ML syntax. The algorithm addresses issues not addressed in Reynolds's presentation or subsequent examples. The most significant extension is that the algorithm operates on a statically typed language. The algorithm also addresses issues such as a method for representing function values as ML datatype values, function values "shared" by more than one higher-order function, local declarations, and expressions with multiple alternates (such as **case** expressions). We have implemented the algorithm as a critical part of a larger program transformation system.

1 Introduction

1.1 Higher-Order Functions and Their Elimination

Higher-order functions (HOFs) are either

1. functions with arguments that are themselves functions
2. functions that return function values

Map functions are common examples of HOFs of the first sort, and curried functions are examples of the second sort.

Defunctionalization [Rey72], *firstification* [Nel91], and *higher-order removal* [CD93] are methods for eliminating higher-order functions from a program, while preserving the semantics of the program. It is desirable to eliminate higher-order functions from a program because some program transformations which improve program efficiency can only be performed on first-order programs (*i.e.* functional programs without higher-order functions).

Higher-order functions of the second sort can be simply eliminated from a program via uncurrying. The uncurrying transformation ensures that every function call includes values for all

*The authors are supported in part by a grant from the NSF (CCR-9101721) and by a contract with Air Force Material Command (F19628-93-C-0069).

†PO Box 91000, Portland, Oregon 97291-1000, USA; email: {bell,hook}@cse.ogi.edu

arguments to the function, and that function declarations include all necessary parameters. If a value for one of a function's arguments is not available at a call site, a lambda abstraction is constructed at the call site, supplying the function with all its arguments.

Higher-order functions of the first sort are not so easily eliminated from a program. Thus, this research focuses on **HOF**s with function-valued parameters. We will call such a parameter a **HOP** (for "higher-order parameter").

Reynolds presents defunctionalization via a specific example. This is also true in other presentations of the method [BJ82, Sch85a, Sch85b, Sch88, Wan85]. Thus there are many details left out of these presentations, and there are theoretical issues not addressed in the presentations. Implementation details include the choice of a representation for function values, a method for detecting **HOF**s and **HOP**s, handling **HOP**s that are used by more than one **HOF**, and handling expressions with multiple alternates (such as **case** expressions). Theoretical issues include polymorphic **HOF**s, local declarations, and function values contained in data structures.

Reynolds's defunctionalization is the basis for Sestoft's closure analysis, which is used for globalization [Ses89], control-flow analysis [Shi88, Shi91] and binding-time analysis [Bon90]. The fundamental difference between our approach and closure analysis is that we construct first-order datatypes that characterize exactly the structure of each higher-order value, whereas closure analysis has one large, implicitly recursive type allowing arbitrary structures to be represented.

Characterizing the structure of higher-order values exactly facilitates subsequent transformations. Bellegarde and Hook use the data structures calculated by defunctionalization as the basis for a series of program transformations that rewrite a higher-order, monad-based definition of de Bruijn substitution into the standard, efficient first-order implementation using arithmetic operations for index manipulation [BH93]. In that derivation the form of defunctionalization presented here reveals that natural numbers can encode the higher-order function values. It is not clear how this could have been accomplished with an untyped closure analysis.

Firstify, the implementation of this defunctionalization algorithm, has been used together with Schism[Con92], to automatically perform the specialization and defunctionalization transformations presented in Bellegarde and Hook[BH93].

1.2 HOF Elimination via Specialization

Chin and Darlington [CD93] have developed a **HOF** elimination algorithm based on fold/unfold rules that uses *specialization*. The following example demonstrates the algorithm. The program below contains a **HOF** called *map*, which applies a function to each argument in a list. After its definition, *map* is called with the function *increment*.

$$\begin{array}{l} \textbf{fun}\ \ map\ f\ Nil\ =\ Nil \\ \quad |\ \ map\ f\ Cons(x, xs)\ =\ Cons(f\ x, map\ f\ xs) \end{array}$$

$$\textbf{fun}\ addone\ l\ \ =\ \ map\ increment\ l$$

After specialization, the call to *map* is unfolded to the body of *map*, specialized with respect to the **HOP** used. The resulting program is:

$$\begin{array}{l}\textbf{fun}\ \ map\ f\ Nil\ =\ Nil \\ \quad |\ \ map\ f\ Cons(x, xs)\ =\ Cons(f\ x, map\ f\ xs)\end{array}$$

$$\begin{array}{l}\textbf{fun}\ \ addone\ Nil\ =\ Nil \\ \quad |\ \ addone\ Cons(x, xs)\ =\ Cons(increment\ x, addone\ xs)\end{array}$$

Note that *map* is not modified. However, since it is no longer called, it can safely be eliminated from the program.

This method is limited. Only higher-order functions whose functional arguments are *variable-only* can be specialized with this approach. A functional argument in a recursive call to a higher-order function is variable-only if the argument is an identifier. Functional arguments in non-recursive function calls are all variable-only. If a functional argument in a recursive call to a higher-order function is not variable-only, the algorithm will go into an infinite loop trying to specialize the function.

For an example of a **Hop** that is not variable-only, consider the definition of the *map_with_policy* function introduced by Hook, Kieburtz and Sheard[HKS92]:

$$\begin{array}{lll}\textbf{fun}\ map\_with\_policy\ Z\ f\ (Var\ x) & = & Var\ (f\ x) \\ \quad |\ \ map\_with\_policy\ Z\ f\ (Abs\ t) & = & Abs\ (map\_with\_policy\ Z\ (Z\ f)\ t) \\ \quad |\ \ map\_with\_policy\ Z\ f\ (App\ (t, t')) & = & App\ (\ map\_with\_policy\ Z\ f\ t, \\ & & \qquad map\_with\_policy\ Z\ f\ t')\end{array}$$

This function maps a sequence of functions defined by the functional Z and the function f over a term in the lambda calculus. In the recursive call in the *Abs* case, the second **Hop** is not variable-only, and so specialization fails to give a first order program.

1.3 Reynolds's Defunctionalization Method

Reynolds outlined a more general method for eliminating higher-order functions known as defunctionalization [Rey72]. The key to this method is representing **Hop**s as data values. When **Hop**s are transformed into data values, higher-order functions become first order, since they no longer have functions as arguments. However, applications of the **Hop**s need to be modified, since a data value cannot be applied as a function. This is accomplished by creating an "apply" function for each **Hop** of a **Hof**. The apply function is called wherever the **Hop** was applied in the original **Hof**. The apply function takes as arguments the data value representing the **Hop** and all the arguments to the **Hop**. The apply function dispatches based on the **Hop** encoding, and applies the appropriate function to the remaining arguments.

The following example illustrates Reynolds's defunctionalization method.

$$\begin{array}{l}\textbf{fun}\ \ map\ f\ Nil\ =\ Nil \\ \quad |\ \ map\ f\ Cons(x, xs)\ =\ Cons(f\ x, map\ f\ xs)\end{array}$$

$$\textbf{fun}\ addone\ l\ =\ map\ increment\ l$$

$$\textbf{fun}\ subone\ l\ =\ map\ decrement\ l$$

If the program is defunctionalized using strings containing the function name as the representation of function values, the following program is the result.

```
fun apply_map (function_rep, function_arg) =
    case function_arg of
        "increment" ⇒ increment function_arg
      | "decrement" ⇒ decrement function_arg

fun map' f Nil = Nil
  | map' f Cons(x, xs) = Cons(apply_map(f, x), map' f xs)

fun addone l = map' "increment" l

fun subone l = map' "decrement" l
```

Reynolds's method is more powerful than the specialization method of Chin and Darlington because it is not limited to variable-only HOPs. However, a defunctionalized program is only *pseudo first-order* [CD93] because it still mimics the runtime characteristics of the original higher-order program. Chin and Darlington's method produces more efficient implementations. Therefore, when either method is applicable, specialization is preferable to defunctionalization.

1.4 Implementing Defunctionalization

This research is an attempt to produce an automated defunctionalization system for a typed functional language. This required the identification and resolution of the issues and details listed in Section 1.1. *Firstify*, the result of this effort, is a fully automated implementation of Reynolds's method.

Although in this presentation we limit ourselves to monomorphic functions, our implementation works with polymorphic functions as well. Alternative methods for defunctionalization of polymorphic HOFs is an issue we are currently exploring.

In this paper we discuss the algorithm developed for the implementation of *Firstify*. Included is our method for representing function values as data (Section 2), an overview of the algorithm (Section 3), and a description of the implementation (Section 4) .

2 Representing Function Values

Critical to the algorithm is the choice of data representation for function values.

Functions passed as HOPs are assumed to be in one of two forms at the HOF call site—a simple identifier or a lambda abstraction. We assume that lambda abstractions are limited to the following form:

$$\lambda\ p.f(\ldots,\ \hat{p},\ \ldots)$$

Where f is a function, p is a pattern, and $\hat{p}$ is the expression equivalent of p. This assumption is guaranteed by the uncurrying transformation described above.

2.1 Representing Function Identifiers

Each function passed as a **HOP** is represented with a constructed value in a datatype (known as a **HOP** *datatype*). Each identifier passed as a **HOP** is represented by a nullary constructor that merely encodes the name of the function. For example, consider the following **HOF** call site where the first value in the pair passed to *high* is the **HOP**.

$$\textbf{val } z \;=\; high\ (f, x)$$

The identifier f would be represented by a nullary constructor, resulting in the following transformed declaration.

$$\textbf{val } z \;=\; high\ (P\_f\_1\_of\_high, x)$$

2.2 Representing Lambda Abstractions

In the spirit of Reynolds's presentation, each lambda abstraction passed as a **HOP** is represented by a constructor with the domain being a product of type variables, each type variable representing a variable in the abstraction that is free in the context of the abstraction but bound in the context of some surrounding function. The lambda abstraction is represented at the call site by applying the constructor to these variables. Variables that are free in the abstraction but *not* bound in the context of a surrounding function are defined at a more outer scope than the surrounding function, and thus do not need to be represented in the datatype. For an example, consider the following **HOF** call site.

$$\textbf{fun } call\_high\ x \;=\; high\ (\lambda z.g(x, y, z), 0)$$

In the lambda abstraction, x and y are free variables. However, only x is bound in the context of *call_high*. Thus, the only value that needs to be a constructor argument is x. The lambda abstraction would be represented by a constructor with argument x, as follows.

$$\textbf{fun } call\_high\ x \;=\; high\ (P\_Lambda1\_1\_of\_high\ x, 0)$$

The **HOP** datatype for representing the two calls to *high* presented is as follows.

datatype *type_1_of_high* =
  P_f_1_of_high
| *P_Lambda1_1_of_high* **of** τ where τ is the type of the argument to *call_high*

Note that if we use a polymorphic datatype parameterized with a type variable for the type of x, we do not need to determine what the type of x is. This technique is in fact used in the implementation.

2.3 Interpreting the Representations

Apply functions have two parameters—the representation of a function (a value in the **HOP** datatype) and the argument[1] to that function.

An apply function interprets function representations to apply the appropriate function to the argument for that function. Interpreting representations of function identifiers is trivial. Representations of lambda abstractions are interpreted by evaluating the body of the abstraction, modified as follows:

[1] For simplicity we will assume that each higher-order function has a single argument.

- Each variable value stored in the constructor representing the abstraction is substituted into the abstraction body as appropriate.
- The argument to the represented function is substituted into the abstraction body as appropriate.

The apply function corresponding to the datatype shown in the previous section is:

$$\begin{array}{l}\textbf{fun } \mathit{apply\_1\_of\_high}\ (\mathit{function\_rep}, \mathit{function\_arg})\ = \\ \quad \textbf{case } \mathit{function\_rep} \textbf{ of} \\ \qquad \mathit{P\_f\_1\_of\_high}\ \Rightarrow\ \mathit{function\_arg} \\ \qquad \mid\ \mathit{P\_Lambda1\_1\_of\_high}\ x\ \Rightarrow g(x, y, \mathit{function\_arg})\end{array}$$

2.4 Introduction of Recursive Datatypes

Suppose a lambda abstraction **Hop** is used in a recursive call, and the abstraction references the formal **Hop**. For example:

$$\textbf{fun } \mathit{high}\ f\ =\ \ldots\ \mathit{high}\ (\lambda x.\ \ldots\ f\ \ldots)\ \ldots$$

According to the technique described above, f must be encapsulated in the datatype value that represents the abstraction. However, in this case we know the type of f—it is a value in the datatype we wish to create! Thus, the datatype representing **Hop**s of *high* must be recursive.

$$\begin{array}{l}\textbf{datatype } \mathit{type\_1\_of\_high}\ = \\ \quad \mathit{P\_Lambda1\_1\_of\_high} \textbf{ of } \mathit{type\_1\_of\_high}\end{array}$$

When a **Hop** datatype is recursive, the corresponding apply function is either recursive or mutually recursive with another function in the transformed program.

Note that in this example, f is not a variable-only parameter, and so Chin and Darlington's method would not be able to make this function first-order.

2.5 Shared Hops and Hof Sets

Higher-order functions *share* **Hop**s if both **Hof**s are passed the same **Hop**. One way for **Hof**s to share **Hop**s is if one **Hof** passes a **Hop** to another **Hof** via a function call. A collection of **Hof**s that share **Hop**s is known as a **Hof** *set*.

When a datatype and apply function are created by *Firstify*, they each correspond to a **Hof** set, rather than a single **Hof** as has been implied by the examples. This is because in a typed language the representation of all **Hop**s passed to every **Hof** in a **Hof** set must have the same type.

3 The Defunctionalization Algorithm

The defunctionalization algorithm is shown in Figure 2, and the types and functions used in the algorithm are shown in Figure 1.

hof_set : *function_id list*
A **Hof** set. Values of type *function_id* uniquely identify a function in the source program.

add_annotations : *declaration list* → *declaration list*
Annotates a program with type information.

find_hof : *declaration list* → *function_id maybe*
Searches for an untransformed **Hof** in a program. **Hof**s are detected by examining type annotations. If one is found, returns just the function identification. Otherwise returns nothing.

transform_hof : *declaration list* → *function_id* → *hof_set* → *declaration list* ∗ *hof_set*
Transforms the body of a **Hof**. If other **Hof**s are detected which share a **Hop** with this **Hof**, they are appended to the **Hof** set.

transform_hof_calls : *declaration list* → *function_id* → *hof_set* → *declaration list* ∗ *hof_set*
Transforms all calls to a **Hof**. If other **Hof**s are detected which share a **Hop** with this **Hof**, they are appended to the **Hof** set.

Figure 1: Types and functions used in the defunctionalization algorithm

.

```
fun defunctionalize_hof program nil = program
  | defunctionalize_hof program (current_hof :: hof_set) =
    let val (program', hof_set') =
            transform_hof program current_hof hof_set
        val (program'', hof_set'') =
            transform_hof_calls program' current_hof hof_set'
    in
        defunctionalize_hof program'' hof_set''
    end

fun defunctionalize program =
    case find_hof (add_annotations program) of
        nothing ⇒ program
      | just hof_id ⇒ defunctionalize (defunctionalize_hof program [hof_id])
```

Figure 2: The defunctionalization algorithm

.

Defunctionalize_hof defunctionalizes a single **HOF** by transforming the body of the **HOF** and all calls to the **HOF**. During these transformations other **HOF**s in the same **HOF** set might be detected. The function is called recursively on each of these **HOF**s until no more **HOF**s with shared **HOP**s are found.

Defunctionalize performs defunctionalization on a program by annotating the program, finding a **HOF** and calling *defunctionalize_hof* on that **HOF**. The function is recursive, and terminates when no more untransformed **HOF**s are found. Note that the program is re-annotated after each **HOF** set is processed, since the transformations will change the types of declarations.

4 The Implementation

Firstify is our implementation of the above algorithm. In the implementation, type annotation is not performed. Only the types of identifiers declared at the top level are determined. If the input program does not contain local declarations, this is sufficient information to defunctionalize most higher-order functions. However, some **HOF**s with **HOP**s that are themselves **HOF**s cannot be defunctionalized without more comprehensive type annotations.

Firstify operates on Standard ML programs which meet the following criteria:

- All functions are uncurried such that each function has only one argument (possibly a tuple) and all function call sites are fully uncurried. This criterion can be met via the $\mathcal{A}$ transformation of Chin and Darlington[CD93].
- The program contains no **let** declarations. This criterion can be met by lambda lifting[Joh85]. The algorithm has been extended to **let** declarations, but this is not currently reflected in the implementation.
- The only lambda abstractions occur as **HOP**s. This criterion can be met by lambda lifting and the $\mathcal{A}$ transformation of Chin and Darlington.
- No functions are contained in datatype values. A solution to this limitation has not yet been developed.

5 Future Work

Future work on the implementation includes eliminating or weakening the assumptions about the input to *Firstify*. It is particularly desirable to extend *Firstify* as follows:

- Perform type annotation, as described above.
- Weaken the assumption about uncurrying to the assumption that all function declarations and function calls include all arguments to the function, but the arguments are not necessarily contained in a tuple.
- Extend *Firstify* to handle the module system of Standard ML. Presumably *Firstify* would then operate on signatures and structures rather than lists of declarations.

Future work on the algorithm includes the following:

- Extend the algorithm to handle functions "contained" in datatype values. It appears that constructors of such datatypes can be treated "just like" HOFs and defunctionalized as such. We have not, however, worked on the details of this transformation, as this case does not arise in our use of *Firstify*.

- Extend the algorithm to handle polymorphism. Currently the implementation deals with polymorphic higher-order functions with type specialization. Other possibilities are being investigated. Polymorphism is particularly troublesome when a polymorphic HOF is used as a parameter to another HOF. The implementation currently works with HOPs that are HOFs, but only if they are monomorphic.

A full description of the implementation, including other minor restrictions on program input, the extension to **let** declarations, and the treatment of polymorphic higher-order functions can be found in [Bel94].

6 Summary

We have outlined an algorithm for eliminating higher-order functions from a typed functional language. The algorithm is an extension of Reynolds's defunctionalization method. This research addresses features of functional programming languages not explicitly addressed in Reynolds's paper or other references. These features include types, local declarations, and multiple alternate statements. This work also elaborates on implementation details, including detection of higher-order functions, function values that are "shared" by several higher-order functions, and higher-order functions with higher-order parameters. The algorithm has been implemented and is in use as a critical part of a source-to-source program transformation system for Standard ML.

7 Acknowledgements

The authors wish to thank Françoise Bellegarde, Olivier Danvy, and Tim Sheard for their useful comments.

References

[Bel94] Jeffrey M. Bell. An implementation of Reynold's defunctionalization method for a modern functional language. Master's thesis, Oregon Graduate Institute of Science & Technology, January 1994.

[BH93] Françoise Bellegarde and James Hook. Monads, indexes, and transformations. In *TAPSOFT '93: Theory and Practice of Software Development*, volume 668 of *LNCS*, pages 314–327. Springer-Verlag, 1993. A page was omitted from the proceedings, it may be obtained via ftp from `ftp.cse.ogi.edu` in the file `pub/pacsoft/papers/tapsoft.dvi`.

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development*, pages 277–280. Prentice-Hall, 1982.

[Bon90] A. Bondorf. Automatic autoprojectin of higher order recursive equations. In N. Jones, editor, *ESOP '90*, volume 432 of *LNCS*, pages 70–87. Springer-Verlag, May 1990.

[CD93] Wei-Ngan Chin and John Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.

[Con92] Charles Consel. The Schism Manual, version 2.0. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1992.

[HKS92] James Hook, Richard Kieburtz, and Tim Sheard. Generating programs by reflection. Technical Report 92-015, Department of Computer Science and Engineering, Oregon Graduate Institute, July 1992.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. *Lecture Notes in Computer Science*, 201:190–203, 1985.

[Nel91] George C. Nelan. *Firstification*. PhD thesis, Arizona State University, December 1991.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[Sch85a] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[Sch85b] David A. Schmidt. An implementation from a direct semantics definition. *Lecture Notes in Computer Science*, 217:222–235, 1985.

[Sch88] David A. Schmidt. Detecting stack-based environments in denotational definitions. *Science of Computer Programming*, 11(2):107–131, December 1988.

[Ses89] Peter Sestoft. Replacing function parameters by global variables. In *Functional Programming Languages and Computer Architecture*, pages 39–53. ACM Press, September 1989.

[Shi88] Olin Shivers. Control-flow analysis in Scheme. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press, June 1988.

[Shi91] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198. ACM Press, June 1991.

[Wan85] Mitchell Wand. From interpreter to compiler: A representational derivation. *Lecture Notes in Computer Science*, 217:306–324, 1985.