Computer Science and Information Systems 11(4):1417–1434 DOI: 10.2298/CSIS130923030F

Modular Polymorphic Defunctionalization

Georgios Fourtounis, Nikolaos S. Papaspyrou, and Panagiotis Theofilopoulos

National Technical University of Athens
School of Electrical and Computer Engineering
Software Engineering Laboratory
{gfour, nickie, ptheof}@softlab.ntua.gr

Abstract. Defunctionalization is generally considered a whole-program transformation and thus incompatible with separate compilation. In this paper, we formalize a modular variant of defunctionalization which can support separate compilation for a functional programming language with parametric polymorphism. Our technique allows modules in a Haskell-like language to be separately defunctionalized and compiled, then linked together to generate an executable program. We provide a prototype implementation of our modular defunctionalization technique and we discuss the experiences of its application in compiling a large subset of Haskell to low-level C code, based on the intensional transformation.

Keywords: defunctionalization, separate compilation, polymorphism, Haskell.

1. Introduction

Separate compilation allows programs to be organized in modules that can be compiled separately to produce object files, which the linker can later combine to produce the final executable. Modern compilers support separate compilation for many reasons: (a) it saves development time by avoiding the recompilation of all the source code every time a change is made, (b) object files can be collected together in the form of libraries, which can be distributed as closed-source code, and (c) it can be used by build systems like `make` to tractably recompile big code bases [1].

Defunctionalization [32] is a technique which transforms higher-order programs to first-order programs. It does so by eliminating all closures of the source program, replacing them with simple data types and invocations of special first-order `apply` functions.

The technique has been used in the implementation of programming languages, where the compiler uses an intermediate first-order language [7,8,38]. Such implementations can have an advantage when running on certain processors, making closure dispatch cheap and presenting opportunities for code inlining [31]. The transformation was also rediscovered in the context of higher-order logic programming [40] and was used to implement the HiLog language [10]. Defunctionalization has been used in the implementation of type-preserving garbage collectors [39] and a variant of it has been used to apply the first-order Catch analysis tool to higher-order programs [23]. It has also been used to support higher-order functions in first-order database programming [19,20].

Defunctionalization is the seminal work on higher-order removal transformations, being able to always eliminate all higher-order expressions from a program. This contrasts it with other partial techniques, such as Nelan's firstification [25], the method of Chin and Darlington [11], the deforestation variant of Nishimura [27], or the Firstify algorithm of

Mitchell and Runciman [23]. It is the inverse of Church encoding [12] and can be used to reason about programs in terms of data structures, rather than higher-order functions. In particular, Ager *et al.* used defunctionalization to derive abstract machines and compilers from compositional interpreters [2,3] and Danvy presents it as an important transformation, part of a toolbox of techniques, that can be used to reason about program representation, evaluation order, and abstract machines [13]. Defunctionalization has also been used to prove correct control-flow-based program transformations in functional programming languages [5]. The transformation can be applied in a type-safe way in monomorphic and polymorphic languages [6,31] and its correctness has been proved [5,26]. In practice, when the transformation is used to improve performance, it can also be seen as working in the opposite direction compared to optimizations that introduce higher-order structures [16,28,29].

So far, defunctionalization has been presented as a whole-program transformation, a property that has been frequently cited as its major shortcoming [24,31,35,38], rendering it unsuitable as a realistic implementation approach for most compilers. Although it is used in compilers that run in whole-program mode, such as MLton [41] and UHC [15], it has not yet been used in compilers that support separate compilation to native code.

This paper is a self-contained extension of our previous work [18], where we demonstrated how defunctionalization of a simply-typed functional programming language can be combined with separate compilation. Here we modify this technique to support parametric polymorphism, using an adaptation of the technique suggested by Pottier and Gauthier [31]. We add support for over-saturated higher-order function calls; in our previous work, it was assumed that all functions returned a result of ground type but this was not a real nuisance for lazy languages in the absence of polymorphism. Moreover, we describe a modification of the technique with a lightweight heuristic, which significantly reduces the number of closures that need to be generated during defunctionalization.

In the rest of this paper, we give an introduction to polymorphic defunctionalization and describe the problems that appear when we attempt to combine it with separate compilation. We then demonstrate how these problems can be overcome using *modular polymorphic defunctionalization*, a variant that supports separate compilation of modules and linking. We give a formalization of our transformation and describe how it has been implemented in a compiler for a subset of Haskell. To our knowledge, this is the first time defunctionalization is implemented for a functional programming language with parametric polymorphism, in a way that supports separate compilation to native code.

2. Defunctionalization

In this section we introduce the reader to the basics of defunctionalization, a program transformation that takes a higher-order program and produces an equivalent first-order program with additional data types representing function closures.

Assume that we have the following higher-order program written in Haskell.

```
result   = high (add 1) 1 + high inc 2
high g x = g x
inc z    = z + 1
add a b  = a + b
```

There are three higher-order expressions in this program:

1. `add 1` is a partial application of the `add` function yielding a closure of `add` that binds `a` to `1`; the closure has residual type `Int→Int`.
2. `inc` is the name of the `inc` function yielding a (trivial) closure that binds no variables and has residual type `Int→Int`.
3. `g` is a higher-order formal variable of type `Int→Int`.

Defunctionalization will then convert this program to an extensionally equivalent one, using only first-order functions. This is achieved by introducing a data type `Closure` for closures, with one constructor for each different type of closure. In addition, a special function `apply` is introduced that recognizes these constructors and does function dispatch.

```
result   = high (Add 1) 1 + high Inc 2
high g x = apply g x
inc z    = z + 1
add a b  = a + b

data Closure = Add Int | Inc

apply c c0 = case c of
               Inc    → inc c0
               Add a0 → add a0 c0
```

Defunctionalization is a well-known technique, first introduced by Reynolds as an implementation technique for higher-order languages in an untyped setting [32]. Notice that, in the example presented above, both closures conveniently expect one argument of type `Int` and return a result of type `Int`. Had this not been the case, our function `apply` would not typecheck in Haskell; we would either have to use several different types of closures and apply functions (the approach that we took in our previous work [18]) or resort to GADTs (as we do in the rest of this paper).

The simple example above captures the basic idea of defunctionalization; the next case contains polymorphism. Assume the following polymorphic source program in System F; it is written again in Haskell with explicit types for convenience.

```
result :: Int
result = if h id True then h add 17 25 else h (add 2) 5

h :: (a → b) → a → b
h g x = g x

add :: Int → Int → Int
add a b = a + b

id :: a → a
id z = z
```

Here, `h` is a polymorphic function taking a higher-order parameter `g`; it is used three times with three different instantiations for the type parameters `a` and `b`. Notice that the second call to `h` in `result` is *over-saturated*: it takes three arguments, when the definition of `h` only mentions two. This is legal here because `h` is a polymorphic function and, in this call, the type parameter `b` is instantiated with the function type `Int→Int`; the third argument (`25`) will be passed to the function that is the result of the call to `h`.

As Pottier and Gauthier have noted [31], we can defunctionalize polymorphic programs in a typed setting if our target language is System F with GADTs [42]. The defunctionalized counterpart of our example program is then the following and it typechecks in Haskell, if GADTs are allowed. We use explicit types, again, to facilitate the comparison with the original program.

```
result :: Int
result = if h Id True then apply (h Add 17) 25
                      else h (Add1 2) 5

h :: Closure a b → a → b
h g x  = apply g x

add :: Int → Int → Int
add a b = a + b

id :: a → a
id z = z

data Closure par res where
  Id   :: Closure a a
  Add  :: Closure Int (Closure Int Int)
  Add1 :: Int → Closure Int Int

apply :: Closure par res → par → res
apply cl arg = case cl of
                 Id     → id arg
                 Add    → Add1 arg
                 Add1 i → add i arg
```

The basic intuition behind this technique is that GADTs guarantee the type safety of the `apply` function, which receives an argument of the appropriate type, according to the type of the closure `cl`. The `Closure` data type takes two type parameters, `par` and `res`, so a value of type `Closure par res` is a closure for a function of type `par→res`. Notice that `Id` is a (trivial) polymorphic closure for function `id`, that `Add` and `Add1` are closures for function `add` (the first trivial, the second with one given argument), and that the result of applying an `Add` closure to one argument is an `Add1` closure. Notice also that the over-saturated call to `h` needs an extra `apply`.

3. The Source and Target Languages

In this section we describe HL_M, a higher-order functional language with modules that will serve as the source language for modular defunctionalization. We also describe FL, its first-order subset that is the target language of our algorithm. Finally, we discuss how standard defunctionalization fails to separately transform HL_M modules.

3.1. The Source Language HL_M

The language HL_M is a Haskell-like higher-order functional language based on System F, with GADTs [30,36] and modules [14].

A program in HL_M is organized in modules, each having a name, a list of imports consisting of data types and functions defined in other modules, a list of data type declarations, and a list of function definitions. HL_M is defined by the following abstract syntax, where m ranges over module names, d ranges over data type names, b ranges over basic data types, x ranges over function parameters and pattern variables, *op* ranges over built-in constant operators, f ranges over top-level functions, r ranges over function arities (natural numbers), t ranges over type variables, and κ ranges over constructors. The star notation is used to denote zero or more repetitions.

p	$::=$	M^*	*program*
M	$::=$	$\texttt{module}\ m\ \texttt{where}\ I^*\ D^*\ F^*$	*module*
I	$::=$	$\texttt{import}\ m\ (m.d)^*\ (v/r : \tau)^*$	*import*
D	$::=$	$\texttt{data}\ m.d\ t^*\ \texttt{where}\ (m.\kappa : \tau)^*$	*data type*
τ	$::=$	$b \mid t \mid m.d\ \tau^* \mid \tau \rightarrow \tau$	*type*
F	$::=$	$m.f\ x^* =\ e$	*definition*
e	$::=$	$(x \mid v \mid op)\ e^* \mid \texttt{case}\ e\ \texttt{of}\ b^*$	*expression*
v	$::=$	$m.f \mid m.\kappa$	*top-level variable*
b	$::=$	$m.\kappa\ x^*\ \rightarrow\ e$	*case branch*

In HL_M we assume that type names (d), top-level function names (f) and constructor names (κ) are always qualified by the name of the module (m) in which they are defined. Function parameters and pattern variables (x) are local names; they are not qualified. In this way, every module has its own *namespace*: every top-level function is distinct and two different modules can define functions, data types or constructors with the same name, without the danger of name clashes. In our presentation, we will follow Haskell's convention: all functions and variables start with a lowercase letter, while data types, constructors, and modules start with an uppercase letter.

An example program organized in two modules `Lib` and `Main` is the following. For the sake of simplicity, let us assume that the types of `Lib.high` and `Main.high` are monomorphic.

```
module Lib where

Lib.high g x = g x
Lib.h y      = y + 1
Lib.test     = Lib.high Lib.h 1
Lib.add a b  = a + b

module Main where

import Lib ( Lib.high/2 :: (Int→Int)→Int→Int
           , Lib.test/0 :: Int
           , Lib.add/2  :: Int→Int→Int )

Main.result = Main.f 10 + Lib.test
Main.f a    = a + Main.high (Lib.add 1) + Lib.high Main.dec 2
Main.high g = g 10
Main.dec x  = x - 1
```

3.2. The Target Language FL

The language FL is the first-order subset of HL_M, without modules. In other words, in programs written in FL:

1. All functions and data type constructors are first-order.
2. Module qualifiers are considered parts of the names of functions, data types and constructors.
3. Module boundaries are eliminated; programs are lists of data type declarations and function definitions.

For the purpose of our presentation, FL is used as the target language of our defunctionalization transformation. In a real compiler, FL would be replaced by (or subsequently transformed to) native object code.

3.3. The Problem with Naïve Separate Defunctionalization

Let us go back to the two modules `Lib` and `Main` defined in Section 3.1. If we defunctionalize them separately, we obtain the following two pieces of code.

```
-- module Lib where

Lib.high g x = Lib.apply g x
Lib.h y      = y + 1
Lib.test     = Lib.high Lib.H 1
Lib.add a b  = a + b

data Lib.Closure par res where
  Lib.H :: Lib.Closure Int Int

Lib.apply c z = case c of
                  Lib.H → Lib.h z
```

```
-- module Main where

Main.result = Main.f 10 + Lib.test
Main.f a    = a + Main.high (Lib.Add 1) + Lib.high Main.Dec 2
Main.high g = Main.apply g 10
Main.dec x  = x - 1

data Main.Closure par res where
  Lib.Add1 :: Int → Main.Closure Int Int
  Main.Dec :: Main.Closure Int Int

Main.apply c z = case c of
                   Lib.Add a  → Lib.add a z
                   Main.Dec   → Main.dec z
```

First of all, we see that the two different modules generate two different declarations of a `Closure` data type, each having possibly different constructors. Assuming that these two pieces of code are linked successfully and the two `Closure` types are blindly merged, the problem will become evident when the expression `Lib.high Main.Dec 2` will be evaluated: `Lib.high` will call `Lib.apply`, which does not know the closure constructor `Main.Dec` and the program will terminate with an error.

We observe that closure data types, closure constructors and closure dispatching functions must be treated in a special way, if functions from different modules are to exchange higher-order expressions. On the other hand, all other data types, constructors and functions can be safely compiled separately and coexist, since it is guaranteed that there are no name clashes.

4. Modular Defunctionalization

The solution to the problem described in the previous section is to have a proper way of managing the code that is generated by defunctionalization: closure data types, closure constructors and their dispatchers must be collected together from all modules and code for them must only be generated at link-time. Our technique applies defunctionalization separately to each module, transforming from HL_M source code to FL target code, introducing closure constructors and invoking closure dispatchers whenever needed. It remembers the closure constructors that were required for each module and collects this information together with the target code generated for each module. Subsequently, in a final linking step, it merges the closure data type declarations and generates code for the closure dispatcher, based on the collected information.

Our modular defunctionalization is therefore a two-step transformation:

1. *Separate defunctionalization*: Each module is defunctionalized separately. This produces (i) a set of defunctionalized data type declarations; (ii) a set of defunctionalized top-level function definitions; and (iii) information about the closures that correspond to the top-level functions defined in this module. The third part serves as the *defunctionalization interface* of the module. At this point, the defunctionalized definitions from each module can be compiled separately to object code, assuming that closure

constructors and dispatching functions are external symbols to be resolved later, at link-time.

2. *Linking*: The separately defunctionalized code is combined and the missing code is generated for closure constructors and dispatching functions, using the defunctionalization interfaces from the previous step. The missing code can then be compiled and linked with the rest of the already generated code, to produce the final program.

This section formally presents a module-aware variant of defunctionalization. The two steps mentioned above are described in the next two subsections.[1]

4.1. Separate Defunctionalization

This step defunctionalizes each module, generating a list of defunctionalized data type and function definitions, and a list of all closure constructors that correspond to the top-level functions defined in this module. In the rest of this section, we describe how a single module M is defunctionalized.

We assume that type checking (and/or type inference) has already taken place and that all type information is readily available. To simplify presentation, we assume that expressions are annotated with their types (e.g., e^{τ}) but most of the times we will omit such annotations to facilitate the reader.

We also assume a mechanism for generating unique *names* during defunctionalization. All such names will be free of module qualifiers and suitable for use in FL. In particular:

– $\mathcal{N}(m.d)$, $\mathcal{N}(m.f)$, and $\mathcal{N}(m.\kappa)$ generate names for module-qualified types, top-level functions and constructors that appear in the source code of a module;
– $\mathcal{C}\ell(\tau_0, \tau_1)$ generates the GADT corresponding to closures that take arguments of type τ_0 and return a result of type τ_1 (we used `Closure` $\tau_o\ \tau_1$ in the examples);
– $\mathcal{C}(v, n)$ generates the name of a constructor corresponding to the closure of v, binding n arguments; and
– $\mathcal{A}$ generates the name of the closure dispatching function (we used `apply` in the examples).

A number of auxiliary functions for manipulating types will be useful:

– $\mathsf{arity}_T(\tau)$ returns the arity of a type (i.e., how many formal arguments must be supplied before a value of ground type is reached).

$$\begin{array}{lcl} \mathsf{arity}_T(b) & \doteq & 0 \\ \mathsf{arity}_T(t) & \doteq & 0 \\ \mathsf{arity}_T(m.d\ \tau^*) & \doteq & 0 \\ \mathsf{arity}_T(\tau_1 \to \tau_2) & \doteq & 1 + \mathsf{arity}_T(\tau_2) \end{array}$$

– $\mathsf{arity}_V(v)$ returns the arity of a top-level function or constructor (i.e., how many arguments exist in its definition). This is known from the syntax, for top-level functions of the current module, or from information present in the `import` clause. Notice that, for a given top-level function f of type τ, it is always the case that $\mathsf{arity}_V(f) \leq \mathsf{arity}_T(\tau)$; however, the two arities are equal only if the body of the function's definition is of ground type.

[1] A prototype implementation in Haskell of the technique described in this section is available at: `http://www.softlab.ntua.gr/~gfour/mdefunc/`.

– $\mathsf{ground}(\tau)$ converts higher-order types to ground types, by replacing function types with the corresponding closure types.

$$\begin{array}{lcl} \mathsf{ground}(b) & \doteq & b \\ \mathsf{ground}(t) & \doteq & t \\ \mathsf{ground}(m.d\,\tau^*) & \doteq & \mathcal{N}(m.d)\ \mathsf{ground}(\tau)^* \\ \mathsf{ground}(\tau_1 \to \tau_2) & \doteq & \mathcal{C}\ell(\mathsf{ground}(\tau_1), \mathsf{ground}(\tau_2)) \end{array}$$

– $\mathsf{lower}(\tau)$ converts higher-order types to first-order, by replacing the arguments of function types with the corresponding closure types, if necessary.

$$\begin{array}{lcl} \mathsf{lower}(b) & \doteq & b \\ \mathsf{lower}(t) & \doteq & t \\ \mathsf{lower}(m.d\,\tau^*) & \doteq & \mathcal{N}(m.d)\ \mathsf{ground}(\tau)^* \\ \mathsf{lower}(\tau_1 \to \tau_2) & \doteq & \mathsf{ground}(\tau_1) \to \mathsf{lower}(\tau_2) \end{array}$$

The defunctionalization process is formalized using four transformations: $\mathcal{T}(D)$, $\mathcal{D}(F)$, $\mathcal{E}(e)$, $\mathcal{B}(b)$, for type declarations, top-level function definitions, expressions and case branches, respectively. They are defined as follows.

$$\begin{array}{l} \mathcal{T}(\texttt{data}\ m.d\,t^*\ \texttt{where}\ (m.\kappa : \tau)^*) \ \doteq \\ \quad \texttt{data}\ \mathcal{N}(m.d)\ t^*\ \texttt{where}\ (\mathcal{N}(m.\kappa) : \mathsf{lower}(\tau))^* \end{array}$$

$$\begin{array}{lcl} \mathcal{D}(m.f\ x^* \ = \ e) & \doteq & \mathcal{N}(m.f)\ x^* \ = \ \mathcal{E}(e) \\[6pt] \mathcal{E}(x\ e_1\ \ldots\ e_n) & \doteq & \mathcal{A} \ldots (\mathcal{A}\ x\ \mathcal{E}(e_1)) \ldots \mathcal{E}(e_n) \\ \mathcal{E}(v\ e_1\ \ldots\ e_n) & \doteq & \mathcal{C}(v,n)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\ & & \qquad \text{if } n < n' = \mathsf{arity}_V(v) \\ \mathcal{E}(v\ e_1\ \ldots\ e_n) & \doteq & \mathcal{A} \ldots (\mathcal{A}\ (\mathcal{N}(v)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_{n'}))\ \mathcal{E}(e_{n'+1})) \ldots \mathcal{E}(e_n) \\ & & \qquad \text{if } n \geq n' = \mathsf{arity}_V(v) \\ \mathcal{E}(op\ e_1\ \ldots\ e_n) & \doteq & op\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\ \mathcal{E}(\texttt{case}\ e\ \texttt{of}\ b^*) & \doteq & \texttt{case}\ \mathcal{E}(e)\ \texttt{of}\ \mathcal{B}(b)^* \\[6pt] \mathcal{B}(m.\kappa\ x^* \ \to \ e) & \doteq & \mathcal{N}(m.\kappa)\ x^* \ \to \ \mathcal{E}(e) \end{array}$$

In principle: (i) data types are also defunctionalized: all higher-order types in the signatures of constructors are replaced by the corresponding closure data types; (ii) functional parameters or pattern variables are applied by using the corresponding closure dispatching functions; (iii) partial applications of top-level functions and constructors are replaced by closure constructors; (iv) over-saturated calls are replaced by applications of closures returned as function results.

During the first step of the transformation, useful information is collected for every closure corresponding to a top-level function or constructor. This is achieved with function $\mathcal{F}(v^\tau)$, defined as follows. We assume that v is a top-level function or constructor and τ is its type.

$$\begin{array}{lcl} \mathcal{F}(v^\tau) & \doteq & \mathsf{info}(v, \tau, [\,]) \\[6pt] \mathsf{info}(v, \tau, \tau^*) & \doteq & \{(v/r, \tau^*, \mathsf{ground}(\tau))\} \cup \mathsf{info}(v, \tau_2, \tau^* +\!\!+ [\mathsf{ground}(\tau_1)]) \\ & & \qquad \text{if } \tau = \tau_1 \to \tau_2, \text{ and } \mathsf{length}(\tau^*) < \mathsf{arity}_V(v) = r \\ \mathsf{info}(v, \tau, \tau^*) & \doteq & \emptyset \qquad \text{otherwise} \end{array}$$

Function $\mathcal{F}(v^\tau)$ returns a set of triples, one for each possible closure in which v can be used. Each triple contains: (i) the name v and the arity r of the corresponding top-level function, (ii) a list with the types of the arguments that have already been supplied, and (iii) the type of the corresponding closure.

As an example, consider a function `add` with three integer arguments.

```
add a b c = a + b + c
```

This function of arity 3 has type `Int→Int→Int→Int`; it can be used in three closures, when 0, 1 and 2 arguments are supplied:

$$\begin{array}{l}\mathcal{F}(\mathtt{add}^{\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}}) = \\ \quad \{\,(\mathtt{add}/3, [\,], \mathtt{Closure\ Int\ (Closure\ Int\ (Closure\ Int\ Int))}), \\ \quad\ \ (\mathtt{add}/3, [\mathtt{Int}], \mathtt{Closure\ Int\ (Closure\ Int\ Int)}), \\ \quad\ \ (\mathtt{add}/3, [\mathtt{Int}, \mathtt{Int}], \mathtt{Closure\ Int\ Int})\,\}\end{array}$$

It is possible that not all of the different closures generated by function $\mathcal{F}(v^\tau)$ will actually be used in the final program. The implementation is free to use a subset of these closures, e.g. taking just the ones that are generated in the code of the module. However, the final set of closures after linking is not just the union of those generated in the code of each linked module; more closures need to be automatically generated by the dispatching functions, in the case of partial application. We will come back to this point, when we describe our heuristic in Section 5.

4.2. Linking

After separately defunctionalizing a number of modules, we are left with object code, i.e., defunctionalized definitions, and information about closures. To link the final executable program, we must merge all defunctionalized definitions and add the missing closure dispatching functions. Let I be the union of closure information from all modules to be linked. As our presentation is at the source level, we start by generating data type definitions for closures; this would not be necessary if we were linking native code.

The data type for closures, parameterized by the type of the closure's argument a and the type of the result b, is defined as follows.

$$\mathtt{data}\ \mathcal{C}\ell(a,b)\ \mathtt{where}\ \{\ \mathcal{C}(v,n) : \tau^* \to \tau \mid (v/r, \tau^*, \tau) \in I, n = \mathsf{length}(\tau^*)\ \}$$

where $\tau^* \to \tau$ is an abuse of notation for constructing function types with more than one curried arguments, e.g., $[\tau_1, \tau_2, \tau_3] \to \tau \ \doteq\ \tau_1 \to \tau_2 \to \tau_3 \to \tau$.

To generate the closure dispatching function $\mathcal{A}$, we use again the closure information I. As the program is closed at link-time, we only need to generate a big `case` expression with one branch for each closure constructor found in I. The result of such branches can be of two different types: (i) if by supplying one more argument we have reached the corresponding function's arity, then a full application of this function can be performed; or (ii) if more arguments remain to be supplied, a new closure (expecting one argument less) is returned. The two alternatives are treated in common by the auxiliary function $\mathsf{next}(v/r, n)$, defined below.

$$\begin{array}{lcll}\mathsf{next}(v/r,n) & \doteq & \mathcal{N}(v) & \text{if } n = r \\ \mathsf{next}(v/r,n) & \doteq & \mathcal{C}(v,n) & \text{if } n < r\end{array}$$

The definition for $\mathcal{A}$ can then be written as follows.

$$\mathcal{A}\, c\, x \;=\; \texttt{case}\; c\; \texttt{of}\; \{\, \mathcal{C}(v,n)\, y_1\, \ldots\, y_n \to \mathsf{next}(v/r, n+1)\, y_1\, \ldots\, y_n\, x \\ \mid (v/r, \tau^*, \tau) \in I, n = \mathsf{length}(\tau^*) \,\}$$

4.3. Example Revisited

We now return to the example of Section 3.1. Applying the first step of the technique described in the previous section, we obtain two defunctionalized pieces of code and two sets of closure information. Both are given in Fig. 1. The code is very similar to that

Defunctionalized module Lib

```
f_Lib_high g x = apply g x
f_Lib_h y = y + 1
f_Lib_test = f_Lib_high Cf_Lib_h_0 1
f_Lib_add a b = a + b
```

Closure information for module Lib

```
(f_Lib_add/2,  0, Closure Int (Closure Int Int))
(f_Lib_add/2,  1, Int → Closure Int Int)
(f_Lib_h/1,    0, Closure Int Int)
(f_Lib_high/2, 0, Closure (Closure Int Int) (Closure Int Int))
(f_Lib_high/2, 1, Closure Int Int → Closure Int Int)
```

Defunctionalized module Main

```
f_Main_result = f_Main_f 10 + f_Lib_test
f_Main_f a = a + f_Main_high (Cf_Lib_add_1 1)
               + f_Lib_high Cf_Main_dec_0 2
f_Main_high g = apply g 10
f_Main_dec x = x - 1
```

Closure information for module Main

```
(f_Lib_add/2,   0, Closure Int (Closure Int Int))
(f_Lib_add/2,   1, Int → Closure Int Int)
(f_Lib_high/2,  0, Closure (Closure Int Int) (Closure Int Int))
(f_Lib_high/2,  1, Closure Int Int → Closure Int Int)
(f_Main_dec/1,  0, Closure Int Int)
(f_Main_f/1,    0, Closure Int Int)
(f_Main_high/1, 0, Closure (Closure Int Int) Int)
```

Fig. 1. The result of defunctionalization for the example of Section 3.1.

obtained in Section 3.3, except for naming conventions and the fact that they now use only one `Closure` data type and only one `apply` function. Also, notice that the definitions of these two are not included in the compiled code. After linking these two modules, the definitions of the `Closure` GADT and the `apply` function are as given in Fig. 2.

GADT for closures

```
data Closure p r where
  Cf_Lib_add_0   :: Closure Int (Closure Int Int)
  Cf_Lib_add_1   :: Int → Closure Int Int
  Cf_Lib_h_0     :: Closure Int Int
  Cf_Lib_high_0  :: Closure (Closure Int Int) (Closure Int Int)
  Cf_Lib_high_1  :: Closure Int Int → Closure Int Int
  Cf_Main_dec_0  :: Closure Int Int
  Cf_Main_f_0    :: Closure Int Int
  Cf_Main_high_0 :: Closure (Closure Int Int) Int
```

Closure dispatch function

```
apply c x = case c of
              Cf_Lib_add_0      → Cf_Lib_add_1 x
              Cf_Lib_add_1 y1   → f_Lib_add y1 x
              Cf_Lib_h_0        → f_Lib_h x
              Cf_Lib_high_0     → Cf_Lib_high_1 x
              Cf_Lib_high_1 y1  → f_Lib_high y1 x
              Cf_Main_dec_0     → f_Main_dec x
              Cf_Main_f_0       → f_Main_f x
              Cf_Main_high_0    → f_Main_high x
```

Fig. 2. Code generated during linking for the example of Section 3.1.

5. Optimizing Closure Constructors

For big programs containing a large number of top-level functions, the technique that we described in Section 4 may end up with a large number of closure constructors and an equally large `apply` function. Here we describe a heuristic that can eliminate a potentially significant number of closure constructors (and corresponding cases in the `apply` function), based on a very simple usage analysis for closure constructors.

Our analysis is based on the observation that currying always appends arguments to the right of the argument list of a closure. Therefore, the only closures that can appear during program run-time are those closures that are initially created (and appear in the program text), or the new closures that result from appending extra arguments to existing closures.

We define the set of all closure constructors that may be generated during program run-time as $C_u = C_0 \cup C_a$, where:

1. C_0 is the set of all constructors appearing in the defunctionalized bodies of top-level functions.
2. C_a is the set of all constructors that be created by appending one or more arguments to all closures represented by $c \in C_0$.

In practice, this heuristic is dependent on the number of functions forming closures in the original program and the arities of these functions. Since most Haskell programs do not use all of their functions to form closures, we believe this is a lightweight enough heuristic that is useful in practice.

To make use of this heuristic in a modular way, our technique described in Section 4 must be slightly modified. The defunctionalization interface of a module must not keep information about the "exported" closures that can be formed from its top-level functions and constructors. Instead, it must keep information about the closures that the module actually uses, either formed by local functions and constructors, or by imported ones. Again, the set of all possible closures used in the program will be the union of all such information from all modules, gathered during linking.

Applying this heuristic to the example of Section 3.1, we end up with a `Closure` GADT containing just three constructors, in comparison to the eight constructors in Fig. 2. The obtained GADT and `apply` function are given in Fig. 3.

GADT for closures

```
data Closure p r where
  Cf_Lib_add_1 :: Int → Closure Int Int
  Cf_Lib_h_0 :: Closure Int Int
  Cf_Main_dec_0 :: Closure Int Int
```

Closure dispatch function

```
apply c x = case c of
              Cf_Lib_add_1 y1 → f_Lib_add y1 x
              Cf_Lib_h_0 → f_Lib_h x
              Cf_Main_dec_0 → f_Main_dec x
```

Fig. 3. Code generated during linking for the example of Section 3.1, with our heuristic.

6. Modular Defunctionalization in a Haskell-to-C Compiler

Apart from a simple prototype implementation for a small subset of a Haskell-like language with modules, we have implemented this technique in GIC,[2] a compiler from a large subset of Haskell to low-level C that is based on the intensional transformation [17]. Defunctionalization is used in the front-end of the GIC compiler, transforming from Haskell to a first-order language with data types, which is subsequently processed by the intensional transformation [33,34] to generate C code using lazy activation records [9].

As in our prototype implementation, defunctionalizing a Haskell module in GIC generates a set of function definitions. These can be transformed to C and then compiled to native code. The defunctionalized definitions contain references to external symbols corresponding to closure dispatching functions and constructors. Closure constructor information for each module is kept in a separate file, which describes the *defunctionalization interface* of the module.

[2] Available at `http://www.softlab.ntua.gr/~gfour/dftoic/`.

This technique permits each module to be independently compiled to an object file. These files can be combined by the linker, which does the following:

- It builds the final closure constructor functions and closure dispatchers for all closures in the defunctionalization interfaces;
- It compiles the generated code of closure constructors and dispatching functions to a separate object file; and
- It calls the C linker to combine the compiled code of the modules and the compiled generated code of defunctionalization, in order to build the final executable.

Modular defunctionalization enables incremental software rebuilding for our Haskell subset. Moreover, it enables the building of shared libraries from defunctionalized Haskell code, provided that defunctionalization interfaces are distributed together with object files; such libraries can then be used by any third-party source code that has an appropriate linker.

7. Related Work

Pottier and Gauthier point out that defunctionalization can be modular for languages that are richer than our HL_M and support recursive multi-methods [31]. Our technique is simpler, as it only records closure constructor information for every module. In effect, our separate defunctionalization and linking can be seen as implementing the multi-methods needed by modular defunctionalization.

GRIN's front-end had some support for separate compilation, but the back-end was a whole-program compiler [7]. The Utrecht Haskell Compiler (UHC), which is also based on the GRIN approach, supports separate compilation for a special bytecode format that runs on an interpreter but not for native code [15]. In the context of the specialization transformation in UHC, Middelkoop pointed out that to fully support separate compilation in the presence of defunctionalization, some information should be kept that looks like the abstract syntax tree of a function [22]. We do the same by keeping only closure constructor type information, which is enough to generate the final abstract syntax tree of the required closure dispatchers.

A variant of defunctionalization that introduces no closure constructors nor dispatchers was proposed by Mitchell [23]. Consequently, it is not affected by modularity problems of generated code and is compatible with separate compilation. However, it cannot defunctionalize all higher-order programs, while our transformation is equally powerful with traditional defunctionalization.

Tolmach sketched an extension of his typed closure conversion [37] that can support separate compilation. His sketch shares ideas with our approach: it postpones the generation of constructors and a set of dispatching functions for linking time. However it applies to monomorphic languages and uses extensible data types and a special analysis of global-vs-local closures and dispatching functions, while we support System F with GADTs and a uniform treatment of all function closures, based on module-qualified names. In subsequent work [38], Tolmach mentions that his technique can be applied to polymorphic programs, again using extensible data type declarations.

As noted in Section 6, our prototype compiler still does not support full Haskell; it currently lacks support for ad-hoc polymorphism using type classes. However, this is just a feature missing from the compiler front-end: our approach supports type classes, if their concretization encoding is used [31]. This contrasts our technique with MLton's monomorphisation [8], JHC's alternative approach based on a whole program analysis and the lambda cube [21] and GHC and UHC's type classes with dictionaries [4,15].

8. Conclusion

To the best of our knowledge, our approach is the first concrete implementation of the defunctionalization transformation that supports separate compilation of polymorphic functional programs to native code. We do so by defunctionalizing program modules separately while at the same time recording information about closure constructors. We then build and compile closure dispatchers for these constructors and for all program modules at link-time.

Our technique may lose opportunities of inter-module optimizations such as inlining, but loss of these optimizations is a general problem of separate compilation. Although what we described in this paper is related to separate compilation and static linking, it is also applicable to language implementations with dynamic linking without significant modifications, provided that the dynamic loader and linker handles not only the object code but also the defunctionalization interfaces.

Acknowledgments. This research has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: THALES. Investing in knowledge society through the European Social Fund. Project code: MIS 380153. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve this paper.

References

1. Adams, R., Tichy, W., Weinert, A.: The cost of selective recompilation and environment processing. ACM Transactions on Software Engineering and Methodology 3(1), 3–28 (Jan 1994)
2. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: From interpreter to compiler and virtual machine: A functional derivation. Tech. Rep. BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus (Mar 2003)
3. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 8–19. ACM, New York, NY, USA (2003)
4. Augustsson, L.: Implementing Haskell overloading. In: Functional Programming Languages and Computer Architecture. pp. 65–73. ACM Press (1993)
5. Banerjee, A., Heintze, N., Riecke, J.G.: Design and correctness of program transformations based on control-flow analysis. In: Kobayashi, N., Pierce, B.C. (eds.) Theoretical Aspects of Computer Software, Lecture Notes in Computer Science, vol. 2215, pp. 420–447. Springer Berlin Heidelberg (2001)
6. Bell, J.M., Bellegarde, F., Hook, J.: Type-driven defunctionalization. In: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming. pp. 25–37. ACM, New York, NY, USA (1997)

7. Boquist, U., Johnsson, T.: The GRIN project: A highly optimising back end for lazy functional languages. In: Proceedings of the 8th International Workshop on Implementation of Functional Languages. pp. 58–84. No. 1268 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg (Sep 1996)
8. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: Smolka, G. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 1782, pp. 56–71. Springer Berlin Heidelberg (2000)
9. Charalambidis, A., Grivas, A., Papaspyrou, N.S., Rondogiannis, P.: Efficient intensional implementation for lazy functional languages. Mathematics in Computer Science 2(1), 123–141 (2008)
10. Chen, W., Kifer, M., Warren, D.S.: HiLog: A foundation for higher-order logic programming. The Journal of Logic Programming 15(3), 187 – 230 (1993)
11. Chin, W., Darlington, J.: A higher-order removal method. Higher-order and Symbolic Computation / Lisp and Symbolic Computation 9, 287–322 (1996)
12. Danvy, Olivier; Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. pp. 162–174 (2001), a more comprehensive version is available as Technical Report BRICS-RS-01-23, Department of Computer Science, University of Aarhus
13. Danvy, O.: An analytical approach to programs as data objects. Doctor scientarum thesis, University of Aarhus (2006)
14. Diatchki, I.S., Jones, M.P., Hallgren, T.: A formal specification of the Haskell 98 module system. In: Proceedings of the ACM SIGPLAN Workshop on Haskell. pp. 17–28. ACM, New York, NY, USA (2002)
15. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. pp. 93–104. ACM, New York, NY, USA (2009)
16. Fernandes, J.P., Saraiva, J.: Tools and libraries to model and manipulate circular programs. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. pp. 102–111. ACM, New York, NY, USA (2007)
17. Fourtounis, G., Papaspyrou, N., Rondogiannis, P.: The generalized intensional transformation for implementing lazy functional languages. In: Sagonas, K. (ed.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 7752, pp. 157–172. Springer Berlin Heidelberg (2013)
18. Fourtounis, G., Papaspyrou, N.S.: Supporting separate compilation in a defunctionalizing compiler. In: Leal, J.P., Rocha, R., Simões, A. (eds.) Proceedings of the 2nd Symposium on Languages, Applications and Technologies. OpenAccess Series in Informatics (OASIcs), vol. 29, pp. 39–49. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (Jun 2013)
19. Grust, T., Schweinsberg, N., Ulrich, A.: Functions are data too (defunctionalization for PL/SQL). Proceedings of the VLDB Endowment 6(12), 1214–1217 (2013)
20. Kennedy, O., Ahmad, Y., Koch, C.: DBToaster: Agile views for a dynamic data management system. In: Proceedings of the 5th Biennial Conference on Innovative Data Systems Research. pp. 284–295 (2011)
21. Meacham, J.: JHC: John's Haskell compiler (2007), `http://repetae.net/computer/jhc/`
22. Middelkoop, A.: Uniqueness typing refined. Master's thesis, Universiteit Utrecht, The Netherlands (2006)
23. Mitchell, N., Runciman, C.: Losing functions without gaining data: Another look at defunctionalisation. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. pp. 13–24. ACM, New York, NY, USA (2009)
24. Monnier, S., Saha, B., Shao, Z.: Principled scavenging. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 81–91. ACM, New York, NY, USA (2001)

25. Nelan, G.: Firstification. Ph.D. thesis, Department of Computer Science, Arizona State University, U.S.A. (1991)
26. Nielsen, L.R.: A denotational investigation of defunctionalization. Tech. rep., BRICS (2000), `http://brics.dk/RS/00/47/BRICS-RS-00-47.pdf`
27. Nishimura, S.: Deforesting in accumulating parameters via type-directed transformations. In: Asian Workshop on Programming Languages and Systems. pp. 145–159 (2002)
28. Pardo, A., Fernandes, J.P., Saraiva, J.: Shortcut fusion rules for the derivation of circular and higher-order programs. Higher-Order and Symbolic Computation 24(1-2), 115–149 (2011)
29. Pettorossi, A.: Program development using lambda abstraction. In: Nori, K.V. (ed.) Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, vol. 287, pp. 420–434. Springer Berlin Heidelberg (1987)
30. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming. pp. 50–61. ACM, New York, NY, USA (2006)
31. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization and concretization. Higher-Order and Symbolic Computation 19, 125–162 (2006)
32. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the 25th ACM Annual Conference. vol. 2, pp. 717–740. ACM, New York, NY, USA (1972), reprinted in *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998
33. Rondogiannis, P., Wadge, W.W.: First-order functional languages and intensional logic. Journal of Functional Programming 7(1), 73–101 (Jan 1997)
34. Rondogiannis, P., Wadge, W.W.: Higher-order functional languages and intensional logic. Journal of Functional Programming 9(5), 527–564 (Sep 1999)
35. Schöpp, U.: On interaction, continuations and defunctionalization. In: Hasegawa, M. (ed.) Typed Lambda Calculi and Applications, Lecture Notes in Computer Science, vol. 7941, pp. 205–220. Springer Berlin Heidelberg (2013)
36. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 341–352. ACM, New York, NY, USA (2009)
37. Tolmach, A.: Combining closure conversion with closure analysis using algebraic types. In: Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (1997)
38. Tolmach, A., Oliva, D.P.: From ML to Ada: Strongly-typed language interoperability via source translation. Journal of Functional Programming 8(4), 367–412 (Jul 1998)
39. Wang, D.C., Appel, A.W.: Type-preserving garbage collectors. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 166–178. ACM, New York, NY, USA (2001)
40. Warren, D.H.D.: Higher-order extensions to Prolog: Are they needed? In: Michie, D. (ed.) Machine Intelligence, vol. 10, pp. 441–454. Edinburgh University Press (1982)
41. Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the ML Workshop. pp. 1–1. ACM, New York, NY, USA (2006), invited lecture, available from `http://www.mlton.org/guide/20130715/References.attachments/060916-mlton.pdf` (accessed: Mar. 2014)
42. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 224–235. ACM, New York, NY, USA (2003)

Georgios Fourtounis obtained his Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 2005, where he is currently a Ph.D. candidate. His research interests include functional and dataflow programming, program transformation, and compiler engineering.

Nikolaos Papaspyrou is an Associate Professor in the School of Electrical and Computer Engineering of the National Technical University of Athens. His research interests focus in the theory and implementation of programming languages.

Panagiotis Theofilopoulos obtained his Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 2011 and his M.Sc. in Computer Science from the University of Athens in 2014. His research interests include functional programming, progarm transformations, and type systems.

Received: September 23, 2013; Accepted: April 30, 2014.