Supporting Separate Compilation in a Defunctionalizing Compiler

Georgios Fourtounis and Nikolaos S. Papaspyrou

School of Electrical and Computer Engineering
National Technical University of Athens, Greece
{gfour, nickie}@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. 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 a compiler from a large subset of Haskell to low-level C code, based on the intensional transformation.

1998 ACM Subject Classification D.1.1 Applicative (functional) programming; D.3.3 Language Constructs and Features: Abstract data types, Modules, Packages; F.3.3 Studies of Program Constructs: Functional constructs; D.3.4 Processors: Compilers.

Keywords and phrases Defunctionalization, functional programming, modules, separate compilation.

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.39

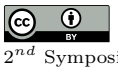
1 Introduction

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. It saves development time by avoiding all the source code to be recompiled every time a change is made. Object files can be collected together in the form of libraries, which can be distributed as closed-source code. It is also used by build systems like `make` to tractably recompile big code bases [1].

Defunctionalization [15] 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. It has been an important theoretical tool, e.g. used by Ager *et al.* to derive abstract machines and compilers from compositional interpreters [3, 2], but it has also been used as a compilation technique [8].

Defunctionalization has so far been presented as a whole-program transformation, a property that has been frequently cited as its major shortcoming, rendering it unsuitable as a realistic implementation approach for most compilers. Although defunctionalization is used in compilers that run in whole-program mode, such as MLton and UHC, so far it has not been used in compilers that support separate compilation to native code.

In the rest of this paper we give an introduction to 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 defunctionalization*, a



2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 39–49

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 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 `Clo` 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:

```
data Clo = Add Int | Inc

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

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 [15]. For applying it to the simply-typed language that we study in this paper, we base our transformation on the type-safe variant of defunctionalization proposed by Bell, Bellegarde, and Hook, which creates different closure dispatching functions for different closure types [4]. For example, assume the following higher-order program:

```
result      = high1 (add 1) 1 1 + high2 inc 2
high1 h i j = h i j
high2 g x   = g x
inc z       = z + 1
add a b c   = a + b + c
```

The types of the closure constructors introduced would be `Int → Int → Int` for `Add1` and `Int → Int` for `Inc`. The example code is then defunctionalized to the following equivalent first-order program:

```
data CloI_I  = Inc | Add2 Int Int
data CloII_I = Add1 Int

result      = high1 (Add1 1) 1 1 + high2 Inc 2
high1 h i j = apply_II_I h i j
high2 g x   = apply_I_I g x
inc z       = z + 1
add a b c   = a + b + c

apply_I_I    clo1 m1    = case clo1 of
                             Inc        → inc m1
                             Add2 a1 b1 → add a1 b1 m1
apply_II_I   clo2 m2 n2 = case clo2 of
                             Add1 a2    → add a2 m2 n2
apply_II_I_I cloC mC    = case cloC of
                             Add1 aC    → Add2 aC mC
```

In this example, the constructors representing closures that can be applied to different types are dispatched by two different functions, `apply_I_I` and `apply_II_I`, that take closures belonging to data types `Clo_I_I` and `Clo_II_I`. We see that another closure constructor is also introduced, `Add2`, representing the closure of `add` binding two arguments. This can be the result of partially applying a closure `Add1` (i.e., `add` with one argument) to another argument, creating a new closure of `add` with two arguments. Partial application of `Add1` closures is done by function `apply_II_I_I`.

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 with modules [9]. A program in HL_M is organized in modules, each having a name, a list of data types and functions that are imported from other modules, a list of data type declarations, and a list of function definitions. HL_M is defined by the following abstract syntax, where μ ranges over module names, a 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, and κ ranges over constructors:

p	$::=$	m^*	*program*
m	$::=$	`module` μ `where imports` I^* δ^* d^*	*module*
I	$::=$	μ $(\mu.a)^*$ $(v : \tau)^*$	*import*
δ	$::=$	`data` $\mu.a = (\mu.\kappa : \tau)^*$	*data type*
τ	$::=$	$b \mid \mu.a \mid \tau \to \tau$	*type*

d	$::=$	$\mu.f\ x^*\ =\ e$	*definition*
e	$::=$	$(x \mid v \mid op)\ e^* \mid$ `case` e `of` b^*	*expression*
v	$::=$	$\mu.f \mid \mu.\kappa$	*top-level variable*
b	$::=$	$\mu.\kappa\ x^*\ \rightarrow\ e$	*case branch*

In HL_M we assume that type names (a), top-level function names (f) and constructor names (κ) are always qualified by the name of the module (μ) 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 that is organized in two modules `Lib` and `Main` is Listing 1.

Listing 1 Example of a program organized in two modules.

```
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.h    :: Int→Int , Lib.high :: (Int→Int)→Int→Int
             Lib.test :: Int     , Lib.add  :: 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. All module boundaries have been 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` that were defined in §3.1. If we defunctionalize them separately, we obtain the two modules presented in Listing 2.

Listing 2 `Main` and `Lib` modules, defunctionalized independently.

```
module Lib where

data Lib.CloI_I  = Lib.H

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

Lib.apply_I_I c z = case c of
                      Lib.H → h z


module Main where

import Lib ( Lib.h    :: Int→Int , Lib.high :: (Int→Int)→Int→Int
             Lib.test :: Int      , Lib.add  :: Int→Int→Int        )

data Main.CloI_I = Lib.Add Int | Main.Dec

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_I_I g 10
Main.dec x  = x - 1

Main.apply_I_I c z = case c of
                       Lib.Add aC → Lib.add aC z
                       Main.dec   → Main.dec z
```

First of all, we see that different modules generate closure constructors that may populate the same closure type, here `Int → Int → Int`, but these constructors and their closure dispatching functions are spread over different modules. This problem is evident when the expression `Lib.high Main.Dec 2` is evaluated: `Lib.high` will call `Lib.apply_I_I`, which does not know the closure constructor `Main.Dec` and the program will terminate with an error.

We observe that closure types, closure constructors and closure dispatching functions must be treated specially, 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 types, 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 to FL 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 generates code for the closure dispatchers based on the collected information.

Our modular defunctionalization is therefore a two-step transformation:

1. *Separate defunctionalization*: Each module is defunctionalized separately. This results to (i) a set of defunctionalized data type declarations; (ii) a set of defunctionalized top-level function definitions; and (iii) information about the closures that were used 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 are used in the transformed code. In the rest of this section, we describe how a single module m is defunctionalized.

The variant of defunctionalization presented here is type-driven (however, this is not essential for our technique, which can also be used for defunctionalizing untyped source languages). We therefore 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:

- $\mathcal{N}(\mu.a)$, $\mathcal{N}(\mu.f)$, and $\mathcal{N}(\mu.\kappa)$ generate names for module-qualified types, top-level functions and constructors that appear in the source code of a module;
- $\mathcal{CL}(\tau)$ generates the name of a data type corresponding to closures of type τ;
- $\mathcal{C}(v, n)$ generates the name of a constructor corresponding to the closure of v, binding n arguments; and
- $\mathcal{A}(\tau, n)$ generates the name of the closure dispatching function for closures of type τ, supplying n arguments.

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

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

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

$$\begin{array}{lcl} \mathsf{arity}(b) & \doteq & 0 \\ \mathsf{arity}(\mu.a) & \doteq & 0 \\ \mathsf{arity}(\tau_1 \to \tau_2) & \doteq & 1 + \mathsf{arity}(\tau_2) \end{array}$$

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

$$\begin{array}{lcl} \mathsf{ground}(b) & \doteq & b \\ \mathsf{ground}(\mu.a) & \doteq & \mathcal{N}(\mu.a) \\ \mathsf{ground}(\tau_1 \to \tau_2) & \doteq & \mathcal{C}\ell(\tau_1 \to \tau_2) \end{array}$$

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

$$\begin{array}{lcl} \mathsf{lower}(b) & \doteq & b \\ \mathsf{lower}(\mu.a) & \doteq & \mathcal{N}(\mu.a) \\ \mathsf{lower}(\tau_1 \to \tau_2) & \doteq & \mathsf{ground}(\tau_1) \to \mathsf{lower}(\tau_2) \end{array}$$

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

$$\begin{array}{l} \mathcal{T}(\texttt{data}\ \mu.a\ =\ \mu.\kappa_1 : \tau_1 \mid \ldots \mid \mu.\kappa_n : \tau_n) \doteq \texttt{data}\ \mathcal{N}(\mu.a)\ =\ \mathcal{N}(\mu.\kappa_1) : \mathsf{lower}(\tau_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \mathcal{N}(\mu.\kappa_n) : \mathsf{lower}(\tau_n) \end{array}$$

$$\mathcal{D}(\mu.f\ x_1 \ldots x_n\ =\ e) \doteq \mathcal{N}(f)\ x_1 \ldots x_n\ =\ \mathcal{E}(e)$$

$$\begin{array}{lcll} \mathcal{E}(x) & \doteq & x & \\ \mathcal{E}(x^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{A}(\tau, n)\ x\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n > 0 \\ \mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{N}(v)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n = \mathsf{arity}(\tau) \\ \mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{C}(v, n)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n < \mathsf{arity}(\tau) \\ \mathcal{E}(op\ e_1\ \ldots\ e_n) & \doteq & op\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \\ \mathcal{E}(\texttt{case}\ e\ \texttt{of}\ b_1\ ;\ \ldots\ ;\ b_n) & \doteq & \texttt{case}\ \mathcal{E}(e)\ \texttt{of}\ \mathcal{B}(b_1)\ ;\ \ldots\ ;\ \mathcal{B}(b_n) & \end{array}$$

$$\mathcal{B}(\mu.\kappa\ x_1\ \ldots\ x_n\ \to\ e) \doteq \mathcal{N}(\mu.\kappa)\ x_1\ \ldots\ x_n\ \to\ \mathcal{E}(e)$$

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

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 that is used in a closure and τ is its type.

$$\mathcal{F}(v^\tau) \doteq \mathsf{info}(v, \tau, [])$$

$$\begin{array}{lcll} \mathsf{info}(v, \tau, \tau^*) & \doteq & \{(\tau, \mathcal{N}(v), \tau^*)\} \cup \mathsf{info}(v, \tau_2, \tau^* +\!\!+ [\mathsf{ground}(\tau_1)]) & \text{if } \tau = \tau_1 \to \tau_2 \\ \mathsf{info}(v, \tau, \tau^*) & \doteq & \emptyset & \text{if } \tau \text{ is a ground type} \end{array}$$

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 type of the closure; (ii) the name of v; (iii) the types of arguments contained in the closure. Notice that, for each triple, the number of arguments remaining to be supplied is equal to the arity of the closure's type. As an example, consider the function `add` from an earlier example:

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

This function can be used in three closures, when 0, 1 and 2 arguments are supplied:

$$\begin{aligned}\mathcal{F}(\texttt{add}^{\texttt{Int}\,\to\texttt{Int}\,\to\texttt{Int}\,\to\texttt{Int}}) \;=\; \{\; & (\texttt{Int}\,\to\texttt{Int}\,\to\texttt{Int}\,\to\texttt{Int}, \texttt{add}, []),\\ & (\texttt{Int}\,\to\texttt{Int}\,\to\texttt{Int}, \texttt{add}, [\texttt{Int}]),\\ & (\texttt{Int}\,\to\texttt{Int}, \texttt{add}, [\texttt{Int}, \texttt{Int}])\;\}\end{aligned}$$

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.

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. For each closure type τ, we generate a definition for $\mathcal{C}\ell(\tau)$ as follows:

$$\texttt{data}\;\mathcal{C}\ell(\tau) \;=\; \{\; \mathcal{C}(x,n) : \tau^* \to \mathcal{C}\ell(\tau) \mid (\tau, x, \tau^*) \in I \text{ and } n = \mathsf{arity}(\tau)\;\}$$

To generate the closure dispatching functions we use again the closure information I. As the program is closed at link-time, we only need to create closure dispatchers for all constructors in I. For every closure type τ, there may be two kinds of closure dispatchers. One is for the full application of such a closure, when all remaining arguments are supplied. However, if $n = \mathsf{arity}(\tau) > 1$, there are also $n-1$ closure dispatchers corresponding to the partial application of such a closure, when only m arguments are supplied ($1 \le m < n$). The first kind of dispatchers returns ground values, whereas the second kind returns closures of smaller arity. Both kinds can be treated uniformly if we define $\mathcal{C}(x,0) \doteq x$. The definition for $\mathcal{A}(\tau, m)$, where now $1 \le m \le n$, can be written as follows: a dispatcher for closures of type τ when m arguments are supplied.

$$\begin{aligned}\mathcal{A}(\tau,m)\; x_0\; x_1\; \ldots\; x_m \;=\; & \texttt{case}\; x_0\; \texttt{of}\\ & \{\; \mathcal{C}(x,n)\; y_1\; \ldots\; y_k \to \mathcal{C}(x, n-m)\; y_1\; \ldots\; y_k\; x_1\; \ldots\; x_m\\ & \quad \mid (\tau, x, \tau^*) \in I \text{ and } n = \mathsf{arity}(\tau) \text{ and } k = |\tau^*|\;\}\end{aligned}$$

5 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 [11]. 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 [16, 17] to generate C code using lazy activation records [7].

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. Closure constructor information for each module is kept in a separate file, which describes the *defunctionalization interface* of the module.

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.

6 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 [14]. Our technique is simpler, as it only records closure constructor information for every module.

GRIN's front-end had some support for separate compilation, but the back-end was a whole-program compiler [5]. 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 [10]. 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 [12]. 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 [13]. 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.

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

7 Conclusion

To the best of our knowledge, our approach is the first concrete implementation of the defunctionalization transformation that supports separate compilation 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.

An open problem is how to combine our technique with polymorphism. There are more than one ways to implement polymorphism in a defunctionalizing compiler similar to ours, such as MLton's monomorphisation [6], UHC's type classes with dictionaries [10], or the techniques used in other defunctionalizing compilers [18, 19]. Each technique may interact differently with the modular defunctionalization presented here. Pottier and Gauthier's polymorphic defunctionalization [14] is another approach to implement polymorphism under defunctionalization; it requires guarded algebraic data types in the target language.

Acknowledgements Work partially supported by the research project "ΘΑΛΗΣ−ΕΜΠ: Handling uncertainty in data intensive applications on a distributed computing environment (cloud computing)" (MIS 380153), funded by the European Social Fund and the Greek national funds through the Operational Program "Education and Lifelong Learning".

References

1 Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

2 Mads Sig Ager, Dariusz Biernacki, Olivier. Danvy, and Jan. Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, March 2003.

3 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19, New York, NY, USA, 2003. ACM.

4 Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, New York, NY, USA, 1997. ACM.

5 Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages*, number 1268 in LNCS, pages 58–84, Berlin, Heidelberg, September 1996. Springer-Verlag.

6 Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 56–71, London, UK, 2000. Springer-Verlag.

7 Angelos Charalambidis, Athanasios Grivas, Nikolaos S. Papaspyrou, and Panos Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, 2(1):123–141, 2008.

8 Lasse R. Danvy, Olivier; Nielsen. Defunctionalization at work. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 162–

174, 2001. A more comprehensive version is available as Technical Report BRICS-RS-01-23, Department of Computer Science, University of Aarhus.

9 Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification of the Haskell 98 module system. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 17–28, New York, NY, USA, 2002. ACM.

10 Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM.

11 Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis. The generalized intensional transformation for implementing lazy functional languages. In *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages*, January 2013. In print.

12 Arie Middelkoop. Uniqueness Typing Refined. Master's thesis, Universiteit Utrecht, the Netherlands, 2006.

13 Neil Mitchell and Colin Runciman. Losing functions without gaining data: Another look at defunctionalisation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 13–24, New York, NY, USA, 2009. ACM.

14 François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, 2006.

15 John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM Annual Conference*, volume 2, pages 717–740, New York, NY, USA, 1972. ACM. Reprinted in *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

16 P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, January 1997.

17 P. Rondogiannis and W. W. Wadge. Higher-order functional languages and intensional logic. *Journal of Functional Programming*, 9(5):527–564, September 1999.

18 Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, 1997.

19 Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.