Supporting Separate Compilation in a Defunctionalizing Compiler

Georgios Fourtounis  Nikolaos Papaspyrou

National Technical University of Athens
School of Electrical and Computer Engineering

2nd International Symposium on Languages, Applications and Technologies (SLATE 2013)
Porto, June 20-21, 2013

Work supported by the project Handling Uncertainty in Data Intensive Applications, co-financed by the European Union (European Social Fund - ESF) and Greek national funds, through the Operational Program “Education and Lifelong Learning”, under the program THALES.
Defunctionalization

- Transforms a higher-order program to an equivalent first-order one (Reynolds, 1972)

- Requirement: the language of the target program must support data types with different constructors (sum types) and pattern matching

- Applicable to both typed and untyped settings
Defunctionalization

Example:

result = double (add 1) 3
double f x = f (f x)
add a b = a + b

| data Cl = Add Int |
| result = double (Add 1) 3 |
| double f x = apply f (apply f x) |
| add a b = a + b |
| apply c z = case c of Add n → add n z |

Main ideas:

1. represent higher-order expressions (closures) with constructors of a new data type Cl
2. higher-order expressions are now applied to arguments through a special apply() function that does pattern matching
Uses of Defunctionalization

1. Implementation of higher-order source languages with first-order target languages (MLton, GRIN)
2. Inter-derivation of abstract machines (Danvy et al.)
3. Transfer of first-order results to higher-order languages
In practice we have a problem: defunctionalization is considered a whole-program transformation but to transform big code bases we need separate compilation.

This work: adding support for separate compilation to a compiler based on defunctionalization.
The Problem

- The `apply()` function must know all functions of the program that may be used to form higher-order expressions.
- Defunctionalizing two separate pieces of code would create two different, incomplete versions of `apply()`.
- Can be addressed in a language with multi-methods (Pottier & Gauthier), but this limits the choice of the target first-order language.
Our Solution

Don’t create the `apply()` function when defunctionalizing a piece of code but keep enough metadata to reconstruct it later, during *linking* of the separately defunctionalized code.
A simple higher-order functional programming language with support for modules:

\[
p ::= m^* \\
m ::= \text{module } \mu \text{ where imports } I^* \delta^* d^* \\
I ::= \mu (\mu.a)^* (v : \tau)^* \\
\delta ::= \text{data } \mu.a = (\mu.\kappa : \tau)^* \\
\tau ::= b \mid \mu.a \mid \tau \rightarrow \tau \\
d ::= \mu.f \ x^* = e \\
e ::= (x \mid v \mid op) e^* \mid \text{case } e \text{ of } b^* \\
v ::= \mu.f \mid \mu.\kappa \\
b ::= \mu.\kappa \ x^* \rightarrow e
\]
Our Source Language $\text{HL}_M$

A simple higher-order functional programming language with support for modules:

\begin{align*}
\text{program} & : \quad P ::= \text{return } m^* \\
\text{module} & : \quad M ::= \text{module } \mu \text{ where imports } I^* \delta^* \text{ d}^* \\
\text{import} & : \quad I ::= \mu (\mu.a)^* (v : \tau)^* \\
\text{data type} & : \quad \delta ::= \text{data } \mu.a = (\mu.\kappa : \tau)^* \\
\text{type} & : \quad \tau ::= b \mid \mu.a \mid \tau \to \tau \\
\text{definition} & : \quad \text{d} ::= \mu.f \ x^* = e \\
\text{expression} & : \quad e ::= (x \mid v \mid \text{op}) e^* \mid \text{case } e \text{ of } b^* \\
\text{top-level name} & : \quad \text{v} ::= \mu.f \mid \mu.\kappa \\
\text{case branch} & : \quad \text{b} ::= \mu.\kappa \ x^* \to e
\end{align*}

Namespaces implemented with module-qualified names
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
The subset of $\text{HL}_M$ where:

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
A transformation in two stages:

1. **Separate defunctionalization**
   Each module is separately defunctionalized to:
   - the equivalent first-order code (without the `apply()` functions)
   - a *defunctionalization interface*

2. **Linking**
   All compiled modules are linked together and their defunctionalization interfaces are read to generate the final `apply()` code
Stage 1: Separate Defunctionalization

Separate defunctionalization of a module:
- transforms all data types and defined functions
- keeps the necessary metadata

We do defunctionalization in a typed setting:
- instead of one big `apply()`, we have a family of `apply_\tau()` functions, to apply closures of type \( \tau \)
- instead of one closure data type, we have a family of \( C_\ell(\tau) \) data types, each containing closures of type \( \tau \)
Stage 1: Defunctionalize Data Types

Transform all higher-order types to first-order:

\[ T(\text{data } \mu.a = \mu.\kappa_1 : \tau_1 \ldots \mu.\kappa_n : \tau_n) \]

\[ \Downarrow \]

\[ \text{data } N(\mu.a) = N(\mu.\kappa_1) : \text{lower}(\tau_1) \]

\[ \ldots \]

\[ N(\mu.\kappa_n) : \text{lower}(\tau_n) \]
Stage 1: Defunctionalize Data Types

Transform all higher-order types to first-order:

\[ T(\text{data } \mu.a = \mu.\kappa_1 : \tau_1 \ldots \mu.\kappa_n : \tau_n) \]
\[ \Downarrow \]
\[ \text{data } \mathcal{N}(\mu.a) = \mathcal{N}(\mu.\kappa_1) : \text{lower}(\tau_1) \]
\[ \ldots \]
\[ \mathcal{N}(\mu.\kappa_n) : \text{lower}(\tau_n) \]

\( \mathcal{N}(\ldots) \) generates unique names for source names

\( \text{lower}(\tau) \) transforms higher-order types to first-order, e.g.:
\( \text{lower}(\text{Int} \to (\text{Int} \to \text{Int}) \to \text{Int}) = \text{Int} \to \mathcal{C}\ell(\text{Int} \to \text{Int}) \to \text{Int} \)
Stage 1: Defunctionalize Types

Example, higher-order record:

```haskell
data Record = R : Int → (Int → Int) → Record

⇓

data Record = R : Int → Cl(Int → Int) → Record
```
Stage 1: Defunctionalize Function Definitions

Standard defunctionalization, formally:

\[ \mathcal{D}(\mu. f \ x_1 \ldots x_n = e) \triangleq \mathcal{N}(f) \ x_1 \ldots x_n = \mathcal{E}(e) \]

\[ \mathcal{E}(x) \triangleq x \]

\[ \mathcal{E}(x^\tau e_1 \ldots e_n) \triangleq \mathcal{A}(\tau, n) \ x \ \mathcal{E}(e_1) \ldots \mathcal{E}(e_n) \quad \text{if } n > 0 \]

\[ \mathcal{E}(v^\tau e_1 \ldots e_n) \triangleq \mathcal{N}(v) \ \mathcal{E}(e_1) \ldots \mathcal{E}(e_n) \quad \text{if } n = \text{arity}(\tau) \]

\[ \mathcal{E}(v^\tau e_1 \ldots e_n) \triangleq \mathcal{C}(v, n) \ \mathcal{E}(e_1) \ldots \mathcal{E}(e_n) \quad \text{if } n < \text{arity}(\tau) \]

\[ \mathcal{E}(\text{op } e_1 \ldots e_n) \triangleq \text{op } \mathcal{E}(e_1) \ldots \mathcal{E}(e_n) \]

\[ \mathcal{E}(\text{case } e \text{ of } b_1 ; \ldots ; b_n) \triangleq \text{case } \mathcal{E}(e) \text{ of } \mathcal{B}(b_1) ; \ldots ; \mathcal{B}(b_n) \]

\[ \mathcal{B}(\mu. \kappa \ x_1 \ldots x_n \to e) \triangleq \mathcal{N}(\mu. \kappa) \ x_1 \ldots x_n \to \mathcal{E}(e) \]

\( \text{arity}(\tau) \) returns the arity of a type, \( \mathcal{A}(\tau, n, ) \) is the apply\(_\tau\) function of closures of type \( \tau \) to \( n \) arguments
Defunctionalization interface of a module: the set of all closure constructors for the functions of the module

Example: \( add : Int \rightarrow Int \rightarrow Int \rightarrow Int \) can form these closures:

<table>
<thead>
<tr>
<th>arguments</th>
<th>residual type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( Int \rightarrow Int \rightarrow Int \rightarrow Int )</td>
</tr>
<tr>
<td>1</td>
<td>( Int \rightarrow Int \rightarrow Int )</td>
</tr>
<tr>
<td>2</td>
<td>( Int \rightarrow Int )</td>
</tr>
</tbody>
</table>
Stage 1: Separate Defunctionalization

Defunctionalization interface for the example:

\[ F(\text{add}^{\text{Int} \to \text{Int} \to \text{Int} \to \text{Int}}) = \{ (\text{Int} \to \text{Int} \to \text{Int} \to \text{Int}, \text{add}, [\text{[]}]), (\text{Int} \to \text{Int} \to \text{Int}, \text{add}, [\text{Int}]), (\text{Int} \to \text{Int}, \text{add}, [\text{Int}, \text{Int}]) \} \]
At the final linking stage, we must generate:

(a) all closure constructors ($\mathcal{Cl}(\tau)$ data types)
(b) all closure dispatchers (apply$_\tau()$ functions)

Given $I$: the union of all generated defunctionalization interfaces
Stage 2: (a) Generate Closure Constructors

For each closure type \( \tau \), generate data type \( \mathcal{C}\ell(\tau) \):

\[
data \mathcal{C}\ell(\tau) = \{ C(x, n) : \tau^* \rightarrow \mathcal{C}\ell(\tau) \mid (\tau, x, \tau^*) \in I \text{ and } n = \text{arity}(\tau) \}
\]
Stage 2: (b) Generate Closure Dispatchers

For all constructors of closures of type $\tau$ in the defunctionalization interfaces, create the $\text{apply}_\tau()$ function to $m$ arguments:

$$A(\tau, m) \ x_0 \ x_1 \ldots \ x_m = \text{case } x_0 \text{ of}$$

$$\{ \ C(x, n) \ y_1 \ldots \ y_k \rightarrow$$

$$C(x, n - m) \ y_1 \ldots \ y_k \ x_1 \ldots \ x_m$$

$$| (\tau, x, \tau^*) \in I, n = \text{arity}(\tau), k = |\tau^*| \}$$
Implementation

- We use modular defunctionalization in GIC, a compiler from a subset of Haskell to C.
- The standard infrastructure of C linking fits well with our technique:
  - separate defunctionalization generates C object files with extern symbols
  - our linker uses the C linker
- Simple heuristics can slim down the defunctionalization interfaces, to control closure constructor explosion.
Future Work

- Extend the technique to polymorphic higher-order languages
- Benchmark separate compilation and linking times for different kinds of programs
Thank you!