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

Defunctionalization

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

Our Source Language HL_M

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

p	$::=$	m^*	*program*
m	$::=$	$\texttt{module}\ \mu\ \texttt{where imports}\ I^*\ \delta^*\ d^*$	*module*
I	$::=$	$\mu\ (\mu.a)^*\ (v : \tau)^*$	*import*
δ	$::=$	$\texttt{data}\ \mu.a = (\mu.\kappa : \tau)^*$	*data type*
τ	$::=$	$b \mid \mu.a \mid \tau \rightarrow \tau$	*type*
d	$::=$	$\mu.f\ x^* = e$	*definition*
e	$::=$	$(x \mid v \mid op)\ e^* \mid \texttt{case}\ e\ \texttt{of}\ b^*$	*expression*
v	$::=$	$\mu.f \mid \mu.\kappa$	*top-level name*
b	$::=$	$\mu.\kappa\ x^* \rightarrow e$	*case branch*

Our Source Language HL_M

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

p	$::=$	m^*	*program*
m	$::=$	$\texttt{module}\ \mu\ \texttt{where imports}\ I^*\ \delta^*\ d^*$	*module*
I	$::=$	$\mu\ (\mu.a)^*\ (v : \tau)^*$	*import*
δ	$::=$	$\texttt{data}\ \mu.a = (\mu.\kappa : \tau)^*$	*data type*
τ	$::=$	$b \mid \mu.a \mid \tau \to \tau$	*type*
d	$::=$	$\mu.f\ x^* = e$	*definition*
e	$::=$	$(x \mid v \mid op)\ e^* \mid \texttt{case}\ e\ \texttt{of}\ b^*$	*expression*
v	$::=$	$\mu.f \mid \mu.\kappa$	*top-level name*
b	$::=$	$\mu.\kappa\ x^* \to e$	*case branch*

Namespaces implemented with module-qualified names

HL_M Example

```
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 Target First-Order Language FL

The subset of 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

Modular Defunctionalization

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 $\texttt{apply}_\tau\texttt{()}$ functions, to apply closures of type τ
- instead of one closure data type, we have a family of $\mathcal{C}\ell(\tau)$ data types, each containing closures of type τ

Stage 1: Defunctionalize Data Types

Transform all higher-order types to first-order:

$$\mathcal{T}(\texttt{data}\ \mu.a\ =\ \mu.\kappa_1 : \tau_1\ \ldots\ \mu.\kappa_n : \tau_n)$$

$$\Downarrow$$

$$\begin{array}{lcl} \texttt{data}\ \mathcal{N}(\mu.a) & = & \mathcal{N}(\mu.\kappa_1) : \mathsf{lower}(\tau_1) \\ & & \ldots \\ & & \mathcal{N}(\mu.\kappa_n) : \mathsf{lower}(\tau_n) \end{array}$$

Stage 1: Defunctionalize Data Types

Transform all higher-order types to first-order:

$$\mathcal{T}(\texttt{data}\ \mu.a\ =\ \mu.\kappa_1 : \tau_1\ \ldots\ \mu.\kappa_n : \tau_n)$$

$$\Downarrow$$

$$\begin{array}{lll} \texttt{data}\ \mathcal{N}(\mu.a) & = & \mathcal{N}(\mu.\kappa_1) : \mathsf{lower}(\tau_1) \\ & & \ldots \\ & & \mathcal{N}(\mu.\kappa_n) : \mathsf{lower}(\tau_n) \end{array}$$

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

$\mathsf{lower}(\tau)$ transforms higher-order types to first-order, e.g.:
$\mathsf{lower}(Int \to (Int \to Int) \to Int) = Int \to \mathcal{C}\ell(Int \to Int) \to Int$

Stage 1: Defunctionalize Types

Example, higher-order record:

$\texttt{data Record} = \texttt{R : Int}\rightarrow\texttt{(Int}\rightarrow\texttt{Int)}\rightarrow\texttt{Record}$

$$\Downarrow$$

$\texttt{data Record} = \texttt{R : Int}\rightarrow\texttt{Cl(Int}\rightarrow\texttt{Int)}\rightarrow\texttt{Record}$

Stage 1: Defunctionalize Function Definitions

Standard defunctionalization, formally:

$$
\begin{array}{lcl}
\mathcal{D}(\mu.f\ x_1 \ldots x_n\ =\ e) & \doteq & \mathcal{N}(f)\ x_1 \ldots x_n\ =\ \mathcal{E}(e) \\
\mathcal{E}(x) & \doteq & x \\
\mathcal{E}(x^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{A}(\tau, n)\ x\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\
& & \hfill \text{if } n > 0 \\
\mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{N}(v)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\
& & \hfill \text{if } n = \text{arity}(\tau) \\
\mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{C}(v, n)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\
& & \hfill \text{if } n < \text{arity}(\tau) \\
\mathcal{E}(op\ e_1\ \ldots\ e_n) & \doteq & op\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\
\mathcal{E}(\texttt{case}\ e\ \texttt{of}\ b_1\ ;\ \ldots\ ;\ b_n) & \doteq & \texttt{case}\ \mathcal{E}(e)\ \texttt{of}\ \mathcal{B}(b_1)\ ;\ \ldots\ ;\ \mathcal{B}(b_n) \\
\mathcal{B}(\mu.\kappa\ x_1\ \ldots\ x_n\ \to\ e) & \doteq & \mathcal{N}(\mu.\kappa)\ x_1\ \ldots\ x_n\ \to\ \mathcal{E}(e)
\end{array}
$$

arity(τ) returns the arity of a type, $\mathcal{A}(\tau, n,)$ is the $\texttt{apply}_\tau\texttt{()}$ function of closures of type τ to n arguments

Stage 1: Generate Defunctionalization Interfaces

Defunctionalization interface of a module: the set of all closure constructors for the functions of the module

Example: $add : Int \to Int \to Int \to Int$ can form these closures:

arguments	residual type
0	$Int \to Int \to Int \to Int$
1	$Int \to Int \to Int$
2	$Int \to Int$

Stage 1: Separate Defunctionalization

Defunctionalization interface for the example:

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

Stage 2: Linking

At the final linking stage, we must generate:

(a) all closure constructors ($\mathcal{C}\ell(\tau)$ data types)

(b) all closure dispatchers ($\texttt{apply}_\tau\texttt{()}$ functions)

given I: the union of all generated defunctionalization interfaces

Stage 2: (a) Generate Closure Constructors

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

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

Stage 2: (b) Generate Closure Dispatchers

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

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

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!