Losing Functions without Gaining Data

– a new method for defunctionalisation

Neil Mitchell [*]
University of York, UK
http://www.cs.york.ac.uk/~ndm/

Colin Runciman
University of York, UK
http://www.cs.york.ac.uk/~colin/

Abstract

We describe an automated transformation which takes a higher-order program, and a produces an equivalent first-order program. Unlike Reynolds style defunctionalisation, it does not introduce any new data types, and the results are more amenable to subsequent analysis operations. Our transformation is implemented, and works on a Core language to which Haskell programs can be reduced. Our method cannot always succeed in removing *all* functional values, but in practice it is remarkably successful.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms languages, transformation

Keywords Haskell, firstification, defunctionalisation

1. Introduction

Higher-order functions are widely used in functional programming languages. Having functions as first-class values leads to more concise code, but it often complicates analysis methods, such as those for checking pattern-match safety (Mitchell and Runciman 2008a) or termination (Sereni 2007).

Example 1

Consider this definition of incList:

```
incList :: [Int] → [Int]
incList = map (+1)

map :: (α → β) → [α] → [β]
map f x = case x of
              []     → []
              y : ys → f y : map f ys
```

The definition of incList has higher-order features. The function $(+1)$ is passed as a functional argument to map. The incList definition contains a partial application of map. The use of first-class functions has led to short code, but we could equally have written:

```
incList :: [Int] → [Int]
incList []       = []
incList (x : xs) = x + 1 : incList xs
```

Although this first-order variant of incList is much longer, it is also more amenable to certain types of analysis. The method presented in this paper transforms the higher-order definition into the first-order one automatically. □

Our defunctionalisation method processes the whole program to remove functional values, without changing the semantics of the program. This idea is not new. As far back as 1972 Reynolds gave a solution, now known as *Reynolds style defunctionalisation* (Reynolds 1972). Unfortunately, Reynolds method effectively introduces a mini-interpreter, which causes problems for analysis tools. Our method produces a program closer to what a human might have written, if denied the use of functional values.

Our method has been implemented in Haskell (Peyton Jones 2003), and operates over the Core language from the York Haskell Compiler (Golubovsky et al. 2007). We have used our transformation within the Catch analysis tool (Mitchell and Runciman 2008a), which checks for potential pattern-match errors in Haskell. We have made our defunctionalisation method available as a library on Hackage[1].

1.1 Contributions

Our paper makes the following contributions:

- We define a defunctionalisation method which, unlike some previous work, does not introduce new data types.
- Our method can deal with the complexities of a language like Haskell, including type classes, continuations and monads.
- Our method makes use of standard transformation steps, but combined in a novel way.
- We identify restrictions which guarantee termination, but are not overly limiting.
- We have implemented our method, and present measured results for much of the nofib benchmark suite.

1.2 Roadmap

The sections following begin with a definition of our Core language (§2), including what we consider to be a first-order program. Next we present an overview of our method (§3), followed by a more detailed account (§4), along with a number of examples (§5). We classify where functional values may remain in a resultant program (§6) and show how to modify our method to guarantee termination (§7). Finally we give results (§8), review related work (§9) and conclude (§10).

* The first author is supported by an EPSRC PhD studentship

[1] http://hackage.haskell.org/, under "firstify"

$\mathsf{prog} = [\mathsf{func}]$	program definition
$\mathsf{func} = \mathsf{f}\ \overline{\mathsf{vs}}\ \mathsf{expr}$	function definition
$\mathsf{expr} = \lambda \mathsf{v} \rightarrow \mathsf{x}$	lambda abstraction
$\mid \mathsf{f}\ \overline{\mathsf{xs}}$	function application
$\mid \mathsf{c}\ \overline{\mathsf{xs}}$	constructor application
$\mid \mathsf{x}\ \overline{\mathsf{xs}}$	general application
$\mid \mathsf{v}$	variable
$\mid \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y}$	let expression
$\mid \textbf{case}\ \mathsf{x}\ \textbf{of}\ \overline{\mathsf{alts}}$	case expression
$\mathsf{alt} = \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \mathsf{x}$	case alternative

We let v range over variables, x and y over expressions, f over function names and c over constructors.

Figure 1. Core Language.

2. Core Language

Our Core language is presented in Figure 1. A program is a set of functions, with a root function named main. A function definition consists of a name, a list of argument variables, and a body expression. The arity of a function is the number of arguments in its definition. We initially assume there are no primitive functions in our language, but explain how to extend our method to deal with them in §4.5.

The variable, case, let, application and lambda expressions are much as they would be in any Core language. The constructor expression consists of a constructor and a list of expressions, exactly matching the arity of the constructor. (Any partially applied constructor can be represented as a lambda expression.) A function application consists of a function name and a possibly empty list of argument expressions. If a function is given fewer arguments than its arity we refer to it as *partially-applied*, matching the arity is *fully-applied*, and more than the arity is *over-applied*. We use the meta functions arity f, body f and args f to denote the arity, body and arguments of f. We use the function rhs to extract the expression on the right of a case alternative.

Informally, a program is higher-order if at runtime the program creates functional values. Functional values can only be created in two ways: (1) a lambda expression; or (2) a partially-applied function application. We therefore make the following definition:

Definition: A program which contains no lambda expressions and no partially-applied functions is first-order. □

Example 1 (revisited)

The original definition of incList is higher-order because of the partial applications of both map and (+). The original definition of map is first-order. In the defunctionalised version, the program is first-order. □

We may expect the map definition to be higher-order, as map has the f x subexpression, where f is a variable, and therefore an instance of general application. We do not define instances of general application to be higher-order, but expect that usually they will be accompanied by the creation of a functional value elsewhere within the program.

3. Our First-Order Reduction Method

Our method works by combining three separate and well-known transformations. Each transformation on its own preserves correctness, and none introduces any additional data types. Our method also applies simplification rules before each transformation, most of which may be found in any optimising compiler (Peyton Jones and Santos 1994).

Arity Raising: A function can be arity raised if the body of the function is a lambda expression. In this situation, the variables within the lambda expression can be added as arguments to the function.

Inlining: Inlining is a standard technique in optimising compilers (Peyton Jones and Marlow 2002), and has been studied in depth.

Specialisation: Specialisation is another standard technique, used to remove type classes (Jones 1994) and more recently to specialise functions to a given constructor (Peyton Jones 2007).

Each transformation has the possibility of removing some functional values, but the key contribution of this paper is *how they can be used together*. Using the fixed point operator (‡) introduced in §4, their combination is:

$\mathsf{firstify} = \mathsf{simplify} \ddagger \mathsf{arity} \ddagger \mathsf{inline} \ddagger \mathsf{specialise}$

We proceed by first giving a brief flavour of how these transformations may be used in isolation to remove functional values. We then discuss the transformations in detail in §4, including how they can be combined.

3.1 Simplification

Simplification serves to group several simple transformations that most optimising compilers apply. Some of these steps have the ability to remove functional values; others simply ensure a normal form for future transformations.

Example 2

$\mathsf{one} = (\lambda \mathsf{x} \rightarrow \mathsf{x})\ 1$

One of our simplification rules transforms this function to:

$\mathsf{one} = \textbf{let}\ \mathsf{x} = 1\ \textbf{in}\ \mathsf{x}$

□

Other rules do not eliminate lambda expressions, but put them into a form that other stages can remove.

Example 3

```
even = let one = 1
       in  λx → not (odd x)
```

A simplification rule lifts the lambda outside of the let expression.

```
even = λx → let one = 1
            in  not (odd x)
```

In general this transformation may cause duplicate computation to be performed, an issue we return to in §4.1.2. □

3.2 Arity Raising

The arity raising transformation increases the definition arity of functions with lambdas as bodies.

Example 4

$\mathsf{even} = \lambda \mathsf{x} \rightarrow \mathsf{not}\ (\mathsf{odd}\ \mathsf{x})$

Here the arity raising transformation lifts the argument to the lambda into a definition-level argument, increasing the arity.

$\mathsf{even}\ \mathsf{x} = \mathsf{not}\ (\mathsf{odd}\ \mathsf{x})$

□

3.3 Inlining

We use inlining to remove functions which return data constructors containing functional values. A frequent source of data constructors containing functional values is the dictionary implementation of type classes (Wadler and Blott 1989).

Example 5

```
main = case eqInt of
           (a, b) → a 1 2

eqInt = (primEqInt, primNeqInt)
```

Both components of the eqInt tuple, primEqInt and primNeqInt, are functional values. We can start to remove these functional values by inlining eqInt:

```
main = case (primEqInt, primNeqInt) of
           (a, b) → a 1 2
```

The simplification stage can now turn the program into a first-order variant, using a rule dealing with a case scrutinising a known constructor.

```
main = primEqInt 1 2
```

□

3.4 Specialisation

Specialisation works by replacing a function application with a specialised variant. In effect, at least one argument is passed at transformation time.

Example 6

```
notList xs = map not xs
```

Here the map function takes the functional value not as its first argument. We can create a variant of map specialised to this argument:

```
map_not x = case x of
                []     → []
                y : ys → not y : map_not ys

notList xs = map_not xs
```

The recursive call in map is replaced by a recursive call to the specialised variant. We have eliminated all functional values. □

3.5 Goals

We define a number of goals: some are *essential*, and others are *desirable*. If essential goals make desirable goals unachievable in full, we still aim to do the best we can.

Essential

Preserve the result computed by the program. By making use of three established transformations, correctness is relatively easy to show.

Ensure the transformation terminates. The issue of termination is much harder. Both inlining and specialisation could be applied in ways that diverge. In §7 we develop a set of criteria to ensure termination.

Recover the original program. Our transformation is designed to be performed before analysis. It is important that the results of the analysis can be presented in terms of the original program. We need a method for transforming expressions in the resultant program into equivalent expressions in the original program.

Introduce no data types. Reynolds method introduces a new data type that serves as a representation of functions, then embeds an interpreter for this data type into the program. We aim to eliminate the higher-order aspects of a program *without* introducing any new data types. By composing our transformation out of existing transformations, none of which introduces data types, we can easily ensure that our resultant transformation does not introduce data types.

Desirable

Remove all functional values. We aim to remove as many functional values as possible. In §6 we make precise where functional values may appear in the resultant programs. If a totally first-order program is required, Reynolds' method can be always be applied after our transformation.

Preserve the space/sharing behaviour of the program. In the expression **let** y = f x **in** y + y, according to the rules of lazy evaluation, f x will be evaluated at most once. We could inline the let binding to give f x + f x, but this expression evaluates f x twice. Where possible, we will avoid changing the sharing of the program. Our goals are primarily for analysis of the resultant code, not to compile and execute the result. Because we are not interested in performance, we permit the loss of sharing in computations if to do so will remove functional values.

Minimize the size of the program. Previous defunctionalisation methods have reflected a concern to avoid undue code-size increase (Chin and Darlington 1996). A smaller resultant program would be desirable, but not at the cost of clarity.

Make the transformation fast. The implementation must be fast enough to permit proper evaluation. Ideally the implementation should be fast enough not to impact on the speed of any subsequent analysis.

4. Method in Detail

Our method proceeds in four iteratively nested steps, simplification (simplify), arity raising (arity), inlining (inline) and specialisation (specialise). Our goal is to combine these steps to remove as many functional values as possible. For example, the initial incList example requires simplification, arity raising and specialisation.

We have implemented our steps in a monadic framework to deal with issues such as obtaining unique free variables and tracking termination constraints. But to simplify the presentation here, we ignore these issues – they are mostly tedious engineering concerns, and do not effect the underlying algorithm.

Our method is written as:

```
firstify = simplify ‡ arity ‡ inline ‡ specialise
```

Each stage will be described separately. The overall control of the algorithm is given by the (‡) operator, defined as follows.

```
infixl ‡

(‡) :: Eq α ⇒ (α → α) → (α → α) → α → α
(‡) f g = fix (g ∘ fix f)

fix :: Eq α ⇒ (α → α) → α → α
fix f x = if x ≡ x′ then x else fix f x′
  where x′ = f x
```

The expression f ‡ g applies f to an input until it reaches a fixed point, then applies g. If g changes the value, then the whole process is repeated until until a fixed point of both f and g is achieved. This formulation has several important properties:

Joint fixpoint After the operation has completed, applying either f or g does not change the value.

$\mathsf{propFix\ f\ g\ x} = \textbf{let}\ \mathsf{r} = (\ddagger)\ \mathsf{f\ g\ x}\ \textbf{in}\ (\mathsf{f\ r} \equiv \mathsf{r}) \wedge (\mathsf{g\ r} \equiv \mathsf{r})$

Idempotence The operation as a whole is idempotent.

$\mathsf{propIdempotent\ f\ g\ x} = \textbf{let}\ \mathsf{op} = \mathsf{f} \ddagger \mathsf{g}\ \textbf{in}\ \mathsf{op\ (op\ x)} \equiv \mathsf{op\ x}$

Function ordering The function f reaches a fixed point before the function g is applied. If a postcondition of f implies a precondition of g, then we can guarantee g's precondition is always met.

These properties allow us to separate the individual transformations from the overall application strategy. The first two properties ensure that the method terminates only when no transformation is applicable. The function ordering allows us to overlap the application sites of two stages, but prefer one stage over another.

The (‡) operator is left associative, meaning that the code can be rewritten with explicit bracketing as:

$\mathsf{firstify} = ((\mathsf{simplify} \ddagger \mathsf{arity}) \ddagger \mathsf{inline}) \ddagger \mathsf{specialise}$

Within this chain we can guarantee that the end result will be a fixed point of every component transformation. Additionally, before each transformation is applied, those to the left will have reached fixed points.

The operator (‡) is written for clarity, not for speed. If the first argument is idempotent, then additional unnecessary work is performed. In the case of chaining operators, the left function is guaranteed to be idempotent if it is the result of (‡), so much computation is duplicated. We describe further optimisations in §8.6.

We describe each of the stages in the algorithm separately. In all subsequent stages, we assume that all the simplification rules have been applied.

4.1 Simplification

The simplification stage has the goal of moving lambda expressions upwards, and introducing lambdas for partially applied functions. This stage makes use of standard simplification rules given in Figure 2, which are found in most optimising compilers, such as GHC (Peyton Jones and Santos 1994). We also make use of additional rules which deal specifically with lambda expressions, given in Figure 3. All of the simplification rules are correct individually. The rules are applied to any subexpression, as long as any rule matches.

4.1.1 Lambda Introduction

The (eta) rule inserts lambdas where possible, using η-expansion. For each partially applied function, a lambda expression is inserted to ensure that the function is given at least as many arguments as its associated arity.

Example 7

$(\circ)\ \mathsf{f\ g\ x} = \mathsf{f\ (g\ x)}$

$\mathsf{even} = (\circ)\ \mathsf{not\ odd}$

Here the functions (∘), not and odd are all partially applied. Lambda expressions can be inserted to saturate these applications.

$\mathsf{even} = \lambda \mathsf{x} \to (\circ)\ (\lambda \mathsf{y} \to \mathsf{not\ y})\ (\lambda \mathsf{z} \to \mathsf{odd\ z})\ \mathsf{x}$

Here the even function, which previously had three instances of partial application, has three lambda expressions inserted. Now each function is fully-applied. This transformation enables the arity raising transformation, resulting in:

$$
\begin{array}{ll}
(\mathsf{x}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} & \\
\quad \Rightarrow \mathsf{x}\ \overline{\mathsf{xs}}\ \overline{\mathsf{ys}} & \text{(app-app)} \\
(\mathsf{f}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} & \\
\quad \Rightarrow \mathsf{f}\ \overline{\mathsf{xs}}\ \overline{\mathsf{ys}} & \text{(fun-app)} \\
(\lambda \mathsf{v} \to \mathsf{x})\ \mathsf{y} & \\
\quad \Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{y}\ \textbf{in}\ \mathsf{x} & \text{(lam-app)} \\
(\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y})\ \mathsf{z} & \\
\quad \Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y\ z} & \text{(let-app)} \\
(\textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\mathsf{p}_1 \to \mathsf{y}_1; \ldots; \mathsf{p}_n \to \mathsf{y}_n\})\ \mathsf{z} & \\
\quad \Rightarrow \textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\mathsf{p}_1 \to \mathsf{y}_1\ \mathsf{z}; \ldots; \mathsf{p}_n \to \mathsf{y}_n\ \mathsf{z}\} & \text{(case-app)} \\
\textbf{case}\ \mathsf{c}\ \overline{\mathsf{xs}}\ \textbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \to \mathsf{y}; \ldots\} & \\
\quad \Rightarrow \textbf{let}\ \overline{\mathsf{vs}} = \overline{\mathsf{xs}}\ \textbf{in}\ \mathsf{y} & \text{(case-con)} \\
\textbf{case}\ (\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y})\ \textbf{of}\ \overline{\mathsf{alts}} & \\
\quad \Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ (\textbf{case}\ \mathsf{y}\ \textbf{of}\ \overline{\mathsf{alts}}) & \text{(case-let)} \\
\textbf{case}\ (\textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \to \mathsf{y}; \ldots\})\ \textbf{of}\ \overline{\mathsf{alts}} & \\
\quad \Rightarrow \textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \to \textbf{case}\ \mathsf{y}\ \textbf{of}\ \overline{\mathsf{alts}}; \ldots\} & \text{(case-case)} \\
\textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \to \lambda \mathsf{v} \to \mathsf{y}; \ldots\} & \\
\quad \Rightarrow \lambda \mathsf{z} \to \textbf{case}\ \mathsf{x}\ \textbf{of} & \text{(case-lam)} \\
\qquad\qquad \{\ldots \mathsf{z}; \mathsf{c}\ \overline{\mathsf{vs}} \to (\lambda \mathsf{v} \to \mathsf{y})\ \mathsf{z}; \ldots \mathsf{z}\} & \\
\mathsf{f}\ \overline{\mathsf{xs}} & \\
\quad \Rightarrow \lambda \mathsf{v} \to \mathsf{f}\ \overline{\mathsf{xs}}\ \mathsf{v} & \text{(eta)} \\
\quad \textbf{where}\ \mathsf{arity\ f} > \mathsf{length}\ \overline{\mathsf{xs}} &
\end{array}
$$

Figure 2. Standard Core simplification rules.

$$
\begin{array}{ll}
\textbf{let}\ \mathsf{v} = (\lambda \mathsf{w} \to \mathsf{x})\ \textbf{in}\ \mathsf{y} & \\
\quad \Rightarrow \mathsf{y}\ [\mathsf{v}\ /\ \lambda \mathsf{w} \to \mathsf{x}] & \text{(bind-lam)} \\
\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y} & \\
\quad \Rightarrow \mathsf{y}\ [\mathsf{v}\ /\ \mathsf{x}] & \text{(bind-box)} \\
\quad \textbf{where}\ \mathsf{x}\ \text{is a boxed lambda (see §4.3)} & \\
\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in} \lambda \mathsf{w} \to \mathsf{y} & \\
\quad \Rightarrow \lambda \mathsf{w} \to \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y} & \text{(let-lam)}
\end{array}
$$

Figure 3. Lambda Simplification rules.

$\mathsf{even\ x} = (\circ)\ (\lambda \mathsf{y} \to \mathsf{not\ y})\ (\lambda \mathsf{z} \to \mathsf{odd\ z})\ \mathsf{x}$

□

This replaces partial application with lambda expressions, and has the advantage of making functional values more explicit, permitting arity raising.

4.1.2 Lambda Movement

The (case-lam) rule lifts a lambda within a case alternative outside the case expression. The (bind-lam) rule inlines a lambda bound in a let expression. The (bind-box) rule will be discussed as part of the inlining stage, see §4.3. The (let-lam) rule can be responsible for a reduction in sharing:

Example 8

$\mathsf{f\ x} = \textbf{let}\ \mathsf{i} = \mathsf{expensive\ x}$
$\qquad\quad \textbf{in}\ \lambda \mathsf{j} \to \mathsf{i} + \mathsf{j}$

$$\begin{aligned}
&\mathsf{isBox}\ [\![\mathsf{c}\ \overline{\mathsf{xs}}]\!] &&= \mathsf{any\ isLambda}\ \overline{\mathsf{xs}} \vee \mathsf{any\ isBox}\ \overline{\mathsf{xs}}\\
&\mathsf{isBox}\ [\![\mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \mathsf{y}]\!] &&= \mathsf{isBox\ y}\\
&\mathsf{isBox}\ [\![\mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \overline{\mathsf{alts}}]\!] &&= \mathsf{any\ (isBox} \circ \mathsf{rhs)}\ \overline{\mathsf{alts}}\\
&\mathsf{isBox}\ [\![\mathsf{f}\ \overline{\mathsf{xs}}]\!] &&= \mathsf{isBox\ (body\ f)}\\
&\mathsf{isBox}\ \_ &&= \mathsf{False}\\
\\
&\mathsf{isLambda}\ [\![\lambda \mathsf{v} \rightarrow \mathsf{x}]\!] &&= \mathsf{True}\\
&\mathsf{isLambda}\ \_ &&= \mathsf{False}
\end{aligned}$$

Figure 4. The isBox function, to test if an expression is a boxed lambda.

```
main xs = map (f 1) xs
```

Here (expensive 1) is computed once and saved. Every application of the functional argument within map performs a single $(+)$ operation. After applying the (let-lam) rule we get:

```
f x = λj → let i = expensive x
           in  i + j
```

Now expensive is recomputed for every element in xs. We include this rule in our simplifier, focusing on functional value removal at the expense of sharing. □

4.2 Arity Raising

Definition: The arity raising step is:

$$\begin{aligned}
&\mathsf{function}\ \overline{\mathsf{vs}} = \lambda \mathsf{v} \rightarrow \mathsf{x}\\
&\quad \Rightarrow \mathsf{function}\ \overline{\mathsf{vs}}\ \mathsf{v} = \mathsf{x}
\end{aligned}$$

□

Given a body which is a lambda expression, the arguments to the lambda expression can be lifted into the definition-level arguments for the function. If a function has its arity increased, fully-applied uses become partially-applied, causing the (eta) simplification rule to fire.

4.3 Inlining

We use inlining as the first stage in the removal of functional values stored within a data value – for example Just $(\lambda x \rightarrow x)$. We refer to expressions that evaluate to functional values inside data values as *boxed lambdas*. If a boxed lambda is bound in a let expression, we substitute the let binding, using the (bind-box) rule from Figure 3. We only inline a function if two conditions both hold: (1) the function's body is a boxed lambda; (2) the function application occurs within a case scrutinee.

Definition: An expression e is a boxed lambda if isBox e $\equiv$ True, where isBox is defined as in Figure 4. □

The isBox function as presented may not terminate, but by simply keeping a list of followed functions, we can assume the result is false in any duplicate call. This modification does not change the result of any previously terminating evaluations.

Example 9

Recalling that [e] is shorthand for (:) e [], where (:) is the cons constructor, the following expressions are boxed lambdas:

```
[λx → x]
(Just [λx → x])
(let y = 1 in [λx → x])
[Nothing, Just (λx → x)]
```

The following are *not* boxed lambdas:

```
λx → id x
[id (λx → x)]
id [λx → x]
```

The final expression *evaluates to* a boxed lambda, but this information is hidden by the id function. We assume that specialisation will deal with any boxed lambdas passed to functions. □

Definition: The inlining transformation is specified by:

$$\begin{aligned}
&\mathbf{case}\ (\mathsf{f}\ \overline{\mathsf{xs}})\ \mathbf{of}\ \overline{\mathsf{alts}}\\
&\quad \Rightarrow \mathbf{case}\ (\mathbf{let}\ \overline{\mathsf{vs}} = \overline{\mathsf{xs}}\ \mathbf{in}\ \mathsf{y})\ \overline{\mathsf{alts}}\\
&\quad \mathbf{where}\\
&\qquad \overline{\mathsf{vs}} = \mathsf{args\ f}\\
&\qquad \mathsf{y} = \mathsf{body\ f}\\
&\qquad \text{If } \mathsf{isBox\ y}\ \text{ evaluates to } \mathsf{True}
\end{aligned}$$

□

As with the simplification stage, there may be some loss of sharing if the definition being inlined has an arity of 0, known as a constant applicative form (CAF). A Haskell implementation computes these expressions only once, and reuses their value as necessary. If they are inlined, this sharing will be lost.

4.4 Specialisation (specialise)

For each application of a function to arguments containing lambdas, a specialised variant is created, and used where applicable. The process follows the same pattern as constructor specialisation (Peyton Jones 2007), but applies where function arguments are lambda expressions, rather than known constructors. Examples of common functions whose applications can usually be made first-order by specialisation include map, filter, foldr and foldl.

The specialisation transformation makes use of *templates*. A template is an expression where some sub-expressions are omitted, denoted by an underscore. The process of specialisation proceeds as follows:

1. Find all functions which have arguments containing lambdas, and generate templates, omitting first-order components (see §4.4.1).
2. For each template, generate a function specialised to that template (see §4.4.2).
3. For each subexpression matching a template, replace it with the generated function (see §4.4.3).

Example 10

```
main xs = map (λx → x) xs

map f xs = case xs of
                []     → []
                y : ys → f y : map f ys
```

Specialisation first finds the application of map in main, and generates the template map $(\lambda x \rightarrow x)$ _. It then generates a unique name for the template (we choose map_id), and generates an appropriate function body. Next all calls matching the template are replaced with calls to map_id, including the call to map within the freshly generated map_id.

```
main xs = map_id xs

map_id xs = case xs of
                 []     → []
                 y : ys → y : map_id ys
```

The resulting code is first-order. □

Specialisation is the only stage which introduces new function names. In order to translate an expression in the result program

```
template :: Expr → Expr
template ⟦f $\overline{xs}$⟧        = apply ⟦f ·⟧ $ map (tem []) $\overline{xs}$
template _ = _

tem :: [String] → Expr → Expr
tem seen y = let tem′ = tem seen in case y of
   ⟦λv → x⟧        → ⟦λv → tem (v : seen) x⟧
   ⟦f $\overline{xs}$⟧          | isBox y → ⟦f (tems′ $\overline{xs}$)⟧
   ⟦v⟧             → if v ∈ seen then ⟦v⟧ else _
   ⟦f $\overline{xs}$⟧          → apply ⟦f ·⟧ $ tems′ $\overline{xs}$
   ⟦c $\overline{xs}$⟧          → apply ⟦c ·⟧ $ tems′ $\overline{xs}$
   ⟦x $\overline{xs}$⟧          → apply (λ(x : $\overline{xs}$) → ⟦x $\overline{xs}$⟧) $ tems′ (x : $\overline{xs}$)
   ⟦let v = x in y⟧ → apply (λ[x, _] →
                        ⟦let v = x in (tem (v : seen) y)⟧) $
                        tems′ [x, y]
   ⟦case x of $\overline{alts}$⟧ → apply (λ(x : _) →
                        ⟦case x of (map alt $\overline{alts}$)⟧) $
                        tems′ (x : map rhs $\overline{alts}$)
  where
    tem′ = tem seen
    tems′ = map tem′
    alt (Alt c $\overline{vs}$ x) = Alt c $\overline{vs}$ (tem ($\overline{vs}$ ++ seen) x)

apply :: ([Expr] → Expr) → [Expr] → Expr
apply f xs = if all (≡ _) xs then _ else f xs
```

Figure 5. Template generation function.

to an equivalent expression in the input program, it is sufficient to replace all generated function names with their associated template, supplying all the necessary variables.

4.4.1 Generating Templates

A template is generated if an expression is a function application, whose arguments include a sub-expression which is either a lambda expression or a boxed lambda – as calculated by the template function in Figure 5. The template additionally includes all sub-expressions whose removal would lead to functional values as omitted subexpressions, and all free variables bound within the template.

Example 11

Expression	Template
id (λx → x)	id (λx → x)
id (Just (λx → x))	id (Just (λx → x))
id (λx → **let** y = 12 **in** 4)	id (λx → _)
id (λx → **let** y = 12 **in** x)	id (λx → **let** y = _ **in** x)

In all examples, the id function has an argument which has a lambda expression as a subexpression. In the final two cases, there are subexpressions which do not depend on variables bound within the lambda – these have been removed and replaced with underscores. The Just constructor is also not dependent on the bound variables, but its removal would require a functional argument as a parameter, so it is left as part of the template. □

4.4.2 Generating Functions

Given a template, to generate an associated function, a unique function name is allocated to the template. For each _ template a fresh argument variable is assigned. The body is produced by unfolding the outer function symbol in the template once.

Example 10 (revisited)

Consider the template map (λx → x) _. Let v_1 be the fresh argument variable for the single _ placeholder, and map_id be the function name:

```
map_id v₁ = map (λx → x) v₁
```

We unfold the definition of map once:

```
map_id v₁ = let f  = λx → x
                xs = v₁
            in  case xs of
                    []     → []
                    y : ys → f y : map f ys
```

After the simplification rules from Figure 3, we obtain:

```
map_id v₁ = let xs = v₁
            in  case xs of
                    []     → []
                    y : ys → y : map (λx → x) ys
```

□

4.4.3 Using Templates

After a function has been generated for each template, every expression matching a template can be replaced by a call to the new function. Every subexpression corresponding to an _ is passed as an argument.

Example 10 (continued)

```
map_id v₁ = let xs = v₁
            in  case xs of
                    []     → []
                    y : ys → y : map_id ys
```

We now have a first-order definition. □

4.5 Extending the method to allow primitive functions

Primitive functions do not have an associated body, and therefore cannot be examined or inlined. We make just two simple changes to support primitives.

1. We define that a primitive application is *not* a boxed lambda.
2. We restrict specialisation so that if a function to be specialised is actually a primitive, no template is generated. The reason for this restriction is that the generation of code associated with a template requires a one-step unfolding of the function, something which cannot occur with a primitive.

Example 12

```
main = (λx → x) `seq` 42
```

Here a functional value is passed as the first argument to the primitive seq. As we are not able to peer inside the primitive, and must preserve its interface, we cannot remove this functional value. For most primitives, such as arithmetic operations, the types ensure that no functional values are passed as arguments. However, the seq primitive is of type $\alpha \rightarrow \beta \rightarrow \beta$, allowing any type to be passed as either of the arguments, including functional values.

Some primitives not only *permit* functional values, but actually *require* them. For example, the primCatch function within the Yhc standard libraries implements the Haskell exception handling function catch. The type of primCatch is $\alpha \rightarrow (\mathsf{IOError} \rightarrow \alpha) \rightarrow \alpha$, taking an exception handler as one of the arguments. □

5. Examples

We now give two examples. Our method can convert the first example to first-order, but not the second.

Example 13 (Inlining Boxed Lambdas)

An earlier version of our defunctionaliser inlined boxed lambdas everywhere they occurred. This made the detection of boxed lambdas much simpler, and did not require looking into external functions. However, it was unable to cope with programs like this one:

$$\begin{array}{l}\mathsf{main} = \mathsf{map}\ (\$1)\ \mathsf{gen} \\ \mathsf{gen} = (\lambda \mathsf{x} \rightarrow \mathsf{x}) : \mathsf{gen}\end{array}$$

The gen function is both a boxed lambda and recursive. If we inlined gen before it was within a case expression, then the method would not be able to remove all lambda expressions. By first specialising map with respect to gen, and waiting until gen is the subject of a case, we are able to remove the functional values. This operation is effectively deforestation (Wadler 1988), which also only performs inlining within the subject of a case. ☐

Example 14 (Functional Lists)

Sometimes lambda expressions are used to build up lists which can have elements concatenated onto the end. Using Hughes lists (Hughes 1986), we can define:

$$\begin{array}{l}\mathsf{nil} = \mathsf{id} \\ \mathsf{snoc\ x\ xs} = \lambda \mathsf{ys} \rightarrow \mathsf{xs}\ (\mathsf{x} : \mathsf{ys}) \\ \mathsf{list\ xs} = \mathsf{xs}\ [\,]\end{array}$$

This list representation provides nil as the empty list, but instead of providing a (:) or "cons" operation, it provides snoc which adds a single element on to the end of the list. The function list is provided to create a standard list. We are unable to defunctionalise such a construction, as it stores unbounded information within closures. We have seen such constructions in both the lines function of the HsColour program, and the sort function of Yhc. However, there is an alternative implementation of these functions:

$$\begin{array}{l}\mathsf{nil} = [\,] \\ \mathsf{snoc} = (:) \\ \mathsf{list} = \mathsf{reverse}\end{array}$$

We have benchmarked these operations in a variety of settings and the list based version appears to use approximately 75% of the memory, and 65% of the time. We suggest that people using continuations for snoc-lists move instead to a list type! ☐

6. Restricted Completeness

Our method would be *complete* if it removed all lambda expressions and partially-applied functions from a program. All partially-applied functions are translated to lambda expressions using the (eta) rule. We therefore need to determine where a lambda expression may occur in a program after the application of our defunctionalisation method.

6.1 Notation

To examine where lambda expressions may occur, we model expressions in our Core language as a set of *syntax trees*. We define the following rules, which generate sets of expressions:

$$\begin{array}{ll}
\mathsf{lam\ x} & = \{\lambda \mathsf{v'} \rightarrow \mathsf{x'} \mid \mathsf{v'} \in \mathsf{v}, \mathsf{x'} \in \mathsf{x}\} \\
\mathsf{fun\ x\ y} & = \{\mathsf{f'}\ \overline{\mathsf{ys}}' \mid \mathsf{f'} \in \mathsf{f}, \mathsf{x'} \in \mathsf{x}, \overline{\mathsf{ys}}'\overline{\in}\mathsf{y}, \mathsf{body\ f'} \equiv \mathsf{x'}\} \\
\mathsf{con\ x} & = \{\mathsf{c'}\ \overline{\mathsf{xs}}' \mid \overline{\mathsf{xs}}'\overline{\in}\mathsf{x}\} \\
\mathsf{app\ x\ y} & = \{\mathsf{x'}\ \overline{\mathsf{ys}}' \mid \mathsf{x'} \in \mathsf{x}, \overline{\mathsf{ys}}'\overline{\in}\mathsf{y}\} \\
\mathsf{var} & = \{\mathsf{v'} \mid \mathsf{v'} \in \mathsf{v}\} \\
\mathsf{let\ x\ y} & = \{\mathbf{let}\ \mathsf{v'} = \mathsf{x'}\ \mathbf{in}\ \mathsf{y'} \mid \mathsf{x'} \in \mathsf{x}, \mathsf{y'} \in \mathsf{y}\}
\end{array}$$

$$\begin{array}{l}
\mathsf{case\ x\ y} = \{\mathbf{case}\ \mathsf{x'}\ \mathbf{of}\ \overline{\mathsf{alts}}' \mid \mathsf{x'} \in \mathsf{x}, \\
\qquad\qquad \overline{\mathsf{alts}}'\overline{\in}\{\mathsf{c'}\ \overline{\mathsf{vs}}' \rightarrow \mathsf{y'} \mid \mathsf{c'} \in \mathsf{c}, \overline{\mathsf{vs}}'\overline{\in}\mathsf{v}, \mathsf{y'} \in \mathsf{y}\}\}
\end{array}$$

Here v is the set of all variables, f the set of function names, and c the set of constructors. We use $\overline{\mathsf{xs}}\overline{\in}\mathsf{e}$ to denote that $\overline{\mathsf{xs}}$ is a sequence of any length, whose elements are drawn from e. In the definition of fun x y, x represents the possible expressions of the body of the function, while y represents the arguments. We can now define an upper bound on the set of unrestricted expressions in our Core language as the smallest solution to the equation s_0:

$$\begin{array}{l}
\mathsf{s}_0 = \mathsf{lam\ s}_0 \cup \mathsf{fun\ s}_0\ \mathsf{s}_0 \cup \mathsf{con\ s}_0 \cup \mathsf{app\ s}_0\ \mathsf{s}_0 \cup \mathsf{var} \cup \\
\qquad \mathsf{case\ s}_0\ \mathsf{s}_0 \cup \mathsf{let\ s}_0\ \mathsf{s}_0
\end{array}$$

6.2 A Proposition about Residual Lambdas

We classify the location of lambdas within the residual program, assuming the program satisfies the following two conditions:

1. The termination criteria does not stop further defunctionalisation (see §7).
2. A primitive function is neither passed a functional argument, nor returns a functional value.

Given these assumptions, a lambda or boxed lambda may only occur in the following places: (1) the body of a lambda expression; (2) an argument to a general application; (3) the body of the main function.

6.3 Proof of the Proposition

First we show that residual expressions belong to a subset of s_0, by defining successively smaller subsets, where $\mathsf{s}_\mathsf{n} \supset \mathsf{s}_{\mathsf{n}+1}$. We use the joint fixpoint property of the (‡) operator to calculate the restrictions imposed by each stage. Secondly we describe which expressions may be the parent of residual lambda expressions, using our refined set of possible expressions.

Restriction 1: Type Safety We know our original program is type safe, and each of our stages preserves semantics, and therefore type safety. This means that the scrutinee of a case cannot be a functional value. Likewise, we know that all constructor expressions are saturated, so will evaluate to a data value, and cannot be the first argument of an application. Refining our bounding set to take account of this, we have:

$$\begin{array}{l}
\mathsf{s}_1 = \mathsf{lam\ s}_1 \cup \mathsf{fun\ s}_1\ \mathsf{s}_1 \cup \mathsf{con\ s} \cup \mathsf{app}\ (\mathsf{s}_1 - \mathsf{con\ s}_1)\ \mathsf{s}_1 \cup \mathsf{var} \cup \\
\qquad \mathsf{case}\ (\mathsf{s}_1 - \mathsf{lam\ s}_1)\ \mathsf{s}_1 \cup \mathsf{let\ s}_1\ \mathsf{s}_1
\end{array}$$

Restriction 2: Standard Simplification Rules Our simplification rules given in Figure 2 are applied until a fixed point is found, meaning that the left-hand side of any rule cannot occur in the output. In view of these restrictions we can further reduce the bounding set of residual expressions:

$$\begin{array}{l}
\mathsf{s}_2 = \mathsf{lam\ s}_2 \cup \mathsf{fun\ s}_2\ \mathsf{s}_2 \cup \mathsf{con\ s}_2 \cup \mathsf{app\ var\ s}_2 \cup \mathsf{var} \cup \\
\qquad \mathsf{case}\ (\mathsf{fun\ s}_2\ \mathsf{s}_2 \cup \mathsf{app\ var\ s}_2 \cup \mathsf{var})\ \mathsf{s}_2 \cup \mathsf{let\ s}_2\ \mathsf{s}_2
\end{array}$$

Restriction 3: Lambda Simplification Rules We apply the lambda rules from Figure 3. As (e − lam e) occurs repeatedly we have factored it out as $\mathsf{l'}$. To allow reuse of $\mathsf{l'}$ in future definitions, we parameterise by n to obtain l'_n.

$$\begin{array}{ll}
\mathsf{s}_3 & = \mathsf{e}_3 \\
\mathsf{e}_3 & = \mathsf{lam\ e}_3 \cup \mathsf{fun\ e}_3\ \mathsf{e}_3 \cup \mathsf{con\ e}_3 \cup \mathsf{app\ var\ e}_3 \cup \mathsf{var} \cup \\
 & \quad \mathsf{case}\ (\mathsf{fun\ e}_3\ \mathsf{e}_3 \cup \mathsf{app\ var\ e}_3 \cup \mathsf{var})\ \mathsf{l}'_3 \cup \mathsf{let\ l}'_3\ \mathsf{l}'_3 \\
\mathsf{l}'_\mathsf{n} & = \mathsf{e}_\mathsf{n} - \mathsf{lam\ e}_\mathsf{n}
\end{array}$$

Restriction 4: Arity Raising Arity raising guarantees that no function body is a lambda expression.

$$s_4 = l'_4$$
$$\begin{aligned}e_4 = {} & \mathsf{lam}\ e_4 \cup \mathsf{fun}\ l'_4\ e_4 \cup \mathsf{con}\ e_4 \cup \mathsf{app}\ \mathsf{var}\ e_4 \cup \mathsf{var} \cup \\ & \mathsf{case}\ (\mathsf{fun}\ l'_4\ e_4 \cup \mathsf{app}\ \mathsf{var}\ e_4 \cup \mathsf{var})\ l'_4 \cup \mathsf{let}\ l'_4\ l'_4\end{aligned}$$

Restriction 5: Inlining and (bind-box) To deal with inlining, we need to work with lambda boxes, as defined by the function isBox, from Figure 4. We first define the subset of expressions which are *not* lambda boxes:

$$\begin{aligned}b'_n = {} & \mathsf{lam}\ e_n \cup \mathsf{fun}\ b'_n\ e_n \cup \mathsf{con}\ (b'_n - \mathsf{lam}\ e_n) \cup \mathsf{app}\ e_n\ e_n \cup \mathsf{var} \cup \\ & \mathsf{case}\ e_n\ b'_n \cup \mathsf{let}\ e_n\ b'_n\end{aligned}$$

Now $(e - b')$ is the set of lambda boxes. We use the inlining and the (bind-box) rules to justify the refinement:

$$s_5 = l'_5$$
$$\begin{aligned}e_5 = {} & \mathsf{lam}\ e_5 \cup \mathsf{fun}\ l'_5\ e_5 \cup \mathsf{con}\ e_5 \cup \mathsf{app}\ \mathsf{var}\ e_5 \cup \mathsf{var} \cup \\ & \mathsf{case}\ (\mathsf{fun}\ (l'_5 \cap b'_5)\ e_5 \cup \mathsf{app}\ \mathsf{var}\ e_5 \cup \mathsf{var})\ l'_5 \cup \\ & \mathsf{let}\ (l'_5 \cap b'_5)\ l'_5\end{aligned}$$

Restriction 6: Specialisation As specialisation removes all lambdas and boxed lambdas from the arguments of function applications we define:

$$s_6 = l'_6$$
$$\begin{aligned}e_6 = {} & \mathsf{lam}\ e_6 \cup \mathsf{fun}\ l'_6\ (l'_6 \cap b'_6) \cup \mathsf{con}\ e_6 \cup \mathsf{app}\ \mathsf{var}\ e_6 \cup \mathsf{var} \cup \\ & \mathsf{case}\ (\mathsf{fun}\ (l'_6 \cap b'_6)\ (l'_6 \cap b'_6) \cup \mathsf{app}\ \mathsf{var}\ e_6 \cup \mathsf{var})\ l'_6 \cup \\ & \mathsf{let}\ (l'_6 \cap b'_6)\ l'_6\end{aligned}$$

Residual Forms Having applied all the rules, we now classify what the parent expressions of a lambda may be. A lambda may not be the root of a function body, because lam is not a production in s. If we show only those expressions which permit a lambda expression as a direct child, we are left with the following upper bound for the set of residual expressions:

$$\begin{aligned}e = {} & \mathsf{lam}\ (\mathsf{lam}\ e \cup e) \cup \mathsf{con}\ (\mathsf{lam}\ e \cup e) \cup \\ & \mathsf{app}\ \mathsf{var}\ (\mathsf{lam}\ e \cup e) \cup \ldots\end{aligned}$$

A lambda may therefore occur as the child of a lambda expression, as an argument to a constructor, or as an argument to an application. However, a constructor containing a lambda is a boxed lambda, and therefore is not permitted anywhere b' is intersected with the expression. We can now classify where a boxed lambda may occur, denoting all expressions as either b to denote that a box may occur, or b' to denote that it may not:

$$\begin{aligned}e = {} & \mathsf{lam}\ b \cup \mathsf{fun}\ b\ b' \cup \mathsf{con}\ b \cup \mathsf{app}\ b'\ b \cup \\ & \mathsf{var} \cup \mathsf{case}\ b'\ b \cup \mathsf{let}\ b'\ b\end{aligned}$$

If the body of a function is a boxed lambda, then any named application of the function is a boxed lambda. Therefore, either the body of the main function is a boxed lambda, or any applied function whose body is a boxed lambda is in one of the b positions shown. Of the productions above, only lam and app are not boxed lambdas. A boxed lambda must either be at the root of a program, or must be contained within an expression which is not a boxed lambda. Therefore, if we exclude expressions which are themselves boxed lambdas:

$$e = \mathsf{lam}\ b \cup \mathsf{app}\ b'\ b \cup \ldots$$

Combining with the restrictions imposed on the locations of lambda expressions, we are left with:

$$e = \mathsf{lam}\ (b \cup \mathsf{lam}\ e \cup e) \cup \mathsf{app}\ \mathsf{var}\ (b \cup \mathsf{lam}\ e \cup e) \cup \ldots$$

So we have shown, as required, that a lambda or boxed lambda may only occur as the body of a lambda, applied to a variable in a general application, or as the root of the main function.

6.4 Example Residual Lambdas

The interesting residual form of lambdas is in an application, applied to a variable – i.e. $v\ (\lambda x \to x)$. Assuming all lambdas are in this form, the lambda within the application cannot be bound to the variable. This leaves three possibilities: (1) either v is bound to $\perp$; or (2) v is never bound to anything; or (3) v is bound outside the program. For example:

```
bottom = bottom
main₁ = bottom (λx → x)

nothing = Nothing
main₂ = case nothing of
            Nothing → 1
            Just f    → f (λx → x)

main₃ f = f (λx → x)
```

7. Proof of Termination

Our algorithm, as it stands, may not terminate. In order to ensure termination, it is necessary to bound both the inlining and specialisation stages. In this section we develop a mechanism to ensure termination, by first looking at how non-termination may arise.

7.1 Termination of Simplification

In order to check the termination of the simplifier we have used the AProVE system (Giesl et al. 2006) to model our rules as a *term rewriting system*, and check its termination. A simple encoding of our rules is given in Appendix A. We have proven termination using both this simple formulation, which considers all constructors to have one alternative of exactly arity one, and a more complex encoding. In both cases, the system is able to report success.

The encoding of the (bind-box) and (bind-lam) rules is excluded. Given these rules, there are non terminating sequences. For example:

```
(λx → x x) (λx → x x)
    ⇒    -- (lam-app) rule
let x = λx → x x in x x
    ⇒    -- (bind-lam) rule
(λx → x x) (λx → x x)
```

Such expressions are a problem for GHC, and can cause the compiler to non-terminate if encoded as data structures (Peyton Jones and Marlow 2002). Other transformation systems (Chin and Darlington 1996) are able to make use of type annotations to ensure these reductions terminate. To guarantee termination, we apply (bind-lam) or (bind-box) at most n times in any definition body. If the body is altered by either inlining or specialisation, we reset the count. Currently we have set n to 1000, and have never had this limit reached. This limited is intended to give a strong guarantee of termination, and will only be necessary rarely – hence the high bound.

7.2 Termination of Arity Raising

Functions may only ever increase in arity, and provided the function bodies do not grow without bound, the increase in arity may only occur a finite number of times. Hence, providing the other stages to not generate infinite expressions, the arity raising stage will terminate.

$$\frac{s \trianglelefteq t_i \text{ for some } i}{s \trianglelefteq \sigma(t_1,\ldots,t_n)}$$

$$\frac{\sigma_1 \equiv \sigma_2, s_1 \trianglelefteq t_1,\ldots,s_n \trianglelefteq t_n}{\sigma_1(s_1,\ldots,s_n) \trianglelefteq \sigma_2(t_1,\ldots,t_n)}$$

Figure 6. Homeomorphic embedding relation.

7.3 Termination of Inlining

A standard technique to ensure termination of inlining is to refuse to inline recursive functions (Peyton Jones and Marlow 2002). For our purposes, this non-recursive restriction is too cautious at it would leave residual lambda expressions in cases such as Example 13. We first present a program which causes our method to non-terminate, then our means of ensuring termination.

Example 15

```
data B x = B x
f = case f of
        B _ → B (λx → x)
```

The f inside the case is a candidate for inlining:

```
case f of B _ → B (λx → x)
    ⇒   -- inlining rule
case (case f of B _ → B (λx → x)) of B_ → B (λx → x)
    ⇒   -- (case-case) rule
case f of B _ → case B (λx → x) of B _ → B (λx → x)
    ⇒   -- (case-con) rule
case f of B _ → B (λx → x)
```

So this expression would cause non-termination. □

To avoid such problems, we permit inlining a function f, at all application sites within a function g, but only once per pair (f, g). In the above example we would inline f within its own body, but only once. Any future attempts to inline f within this function would be disallowed, although f could still be inlined within other function bodies. This restriction is sufficient to ensure termination of inlining. Given n functions, there can only be n^2 possible inlining steps, each for possibly many application sites.

7.4 Termination of Specialisation

The specialisation method, left unrestricted, also may not terminate.

Example 16

```
data Wrap a = Wrap (Wrap a)
            | Value a

f x = f (Wrap x)
main = f (Value head)
```

In the first iteration, the specialiser generates a version of f specialised for the argument Value head. In the second iteration it would specialise for Wrap (Value head), then in the third with Wrap (Wrap (Value head)). Specialisation would generate an infinite number of specialisations of f. □

To ensure we only specialise a finite number of times we use a homeomorphic embedding (Leuschel 2002), given in Figure 6. The homeomorphic embedding $\trianglelefteq$ is a well-binary relation, meaning there are no infinite admissible sequences. A sequence $s_1, s_2 \ldots$ is admissible if there are no $i < j$ such that $s_i \trianglelefteq s_j$. This property only holds provided all elements are expressions over a finite alphabet.

For each function, we associate a set S, of expressions. After generating a template t, we only specialise with that template if $\forall s \in S \bullet \neg(s \trianglelefteq t)$. If S is already admissible, then the sequence S with t added at the end is still admissible. After specialising with a template we add that template to the set S associated with that expression. When we create a new function based on a template, we copy the S associated with the function in which the specialisation is performed.

One of the conditions for termination of homeomorphic embedding is that there is only a finite alphabet. To ensure this condition, we first consider all variables and literals to be equivalent. However, this is not sufficient. During the process of specialisation we create new functions, and these new functions are new symbols in our language. So we only use function names from the original input program. Every template has a correspondence with an expression in the original program. We perform the homeomorphic embedding test only after transforming all templates into their original equivalent expression.

Example 16 (revisited)

Using homeomorphic embedding, we again generate the specialised variant of f (Value head). Next we would generate the template f (Wrap (Value head)). However, f (Value head) $\trianglelefteq$ f (Wrap (Value head)), so the new template would not be used. □

While homeomorphic embedding is sufficient to achieve full defunctionalisation in most simple examples, there are examples where it terminates prematurely.

Example 17

```
main y = f (λx → x) y
f x y = fst (x, f x y) y
```

Here we first generate a specialised variant of f (λx → x) y. If we call the specialised variant f′, we have:

```
f′ y = fst (λx → x, f′ y) y
```

Note that the recursive call to f has also been specialised. We now attempt to generate a specialised variant of fst, using the template fst (λx → x, f′ y) y. Unfortunately, this template is an embedding of the template we used for f′, so we do not specialise and the program remains higher-order. But if we did permit a further specialisation, we would obtain the first-order equivalent:

```
f′ y = fst′ y y
fst′ y1 y2 = y2
```

□

This example may look slightly obscure, but similar situations occur commonly with the standard translation of dictionaries. Often, classes have default methods, which call other methods in the same class. These recursive class calls often pass dictionaries, embedding the original caller even though no recursion actually happens.

To alleviate this problem, instead of storing one set S, we store a sequence of sets, $S_1 \ldots S_n$ – where n is a small positive number, constant for the duration of the program. Instead of adding to the set S, we now add to the lowest set S_i where adding the element will not violate the admissible sequence. Each of the sets S_i is still finite, and there are a finite number (n) of them, so termination is maintained.

By default our defunctionalisation program uses 8 sets. In the results table given in §8, we have given the minimum possible value of n to remove all lambda expressions within each program.

7.5 Termination as a Whole

Given an initial program, the arity raising, inlining and specialisation stages will each apply a finite number of times. The simplification stage is terminating on its own, and will be invoked a finite

Table 1. Results of defunctionalisation on the nofib suite.

Name is the name of the program; **Bound** is the numeric bound used for termination (see §7.4); **HO Create** the number of lambda expressions and under-applied functions, first in the input program and then in the output program; **HO Use** the number of application expressions and over-applied functions; **Time** the execution time of our method in seconds; **Size** the change in the program size measured as the number of nodes in the abstract syntax tree.

Name	Bound	HO Create		HO Use		Time	Size
bernouilli	4	240	0	190	2	0.4	-32%
digits-of-e1	4	217	0	153	2	0.3	-35%
digits-of-e2	5	236	0	198	3	0.4	-32%
exp3_8	4	232	0	154	2	0.3	-39%
gen_regexps	7	116	0	69	0	0.1	-31%
integrate	4	348	0	358	2	0.8	-38%
paraffins	4	360	0	351	2	0.7	-53%
primes	4	217	0	148	2	0.2	-38%
queens	4	217	0	146	2	0.3	-38%
rfib	4	338	0	355	2	0.8	-31%
tak	4	212	0	145	4	0.3	-40%
wheel-sieve1	4	224	0	151	2	0.3	-35%
wheel-sieve2	4	224	0	162	2	0.3	-36%
x2n1	4	345	0	385	2	0.8	-57%
…plus 35 tests from the spectral suite…							
Minimum	2	60	0	46	0	0.1	-78%
Maximum	14	437	1	449	100	1.0	15%
Average	4.8	237	0.06	202	3.6	0.4	-33%

number of times, so will also terminate. Therefore, when combined, the stages will terminate.

8. Results

8.1 Benchmark Tests

We have tested our method with programs drawn from the nofib benchmark suite, and the results are given in Table 1. Looking at the input Core programs, we see many sources of functional values.

- Type classes create dictionaries which are implemented as tuples of functions.
- The monadic bind operation is higher-order.
- The IO data type is implemented as a function.
- The Haskell Show type class uses continuation-passing style extensively.
- List comprehensions in Yhc are desugared to continuations. There are other translations which require less functional value manipulations (Wadler 1987; Coutts et al. 2007).

We have tested all 14 programs from the imaginary section (each represented in the table), and 35 out of the 47 tests in the spectral section. The remaining 12 programs in the spectral section do not compile using the Yhc compiler, mainly due to missing or incomplete libraries. After applying our defunctionalisation method, only 3 programs remain higher-order. We first discuss the residual higher-order programs, then make some observations about each of the columns in the table.

8.2 Higher-Order Residues

The three programs with residual higher-order expressions are as follows:

Example 18

The integer program passes functional values to the primitive seq, using the following function:

```
seqlist [] = return ()
seqlist (x : xs) = x `seq` seqlist xs
```

This function is invoked with the IO monad, so the return () expression is a functional value. It is impossible to remove this functional value without having access to the implementation of the seq primitive. □

Example 19

The pretty and constraints programs both apply a functional value to an expression that evaluates to $\bot$. The case in *pretty* comes from the fragment:

```
type Pretty = Int → Bool → PrettyRep

ppBesides :: [Pretty] → Pretty
ppBesides = foldr1 f nil
```

Here ppBesides evaluates to $\bot$ if the input list to foldr1 is []. The $\bot$ value will be of type Pretty, and will be given further arguments, which can be functional arguments. In reality, the code ensures that the input list is never [], so the program will never fail with this error. □

8.3 Termination Bound (Bound in Table 1)

The termination bound used varies from 2 to 11 for the sample programs. If we exclude the integer program, which is complicated by the primitive operations on functional values, the highest bound is 8. Most programs have a termination bound of 4. There is no apparent relation between the size of a program and the termination bound.

8.4 Creating of Functional Values (HO Create in Table 1)

We use Yhc generated programs as input, which have been lambda lifted (Johnsson 1985), so contain no lambda expressions. The residual program has no partial application, only lambda expressions. Most programs in our test suite start with hundreds of partial applications, but only 3 residual programs contain lambda expressions.

For the purposes of testing defunctionalisation, we have worked on unmodified Yhc libraries, including all the low-level detail. For example, readFile in Yhc is implemented in terms of file handles and pointer operations. Most analysis operations work on an abstracted view of the program, which would reduce the number and complexity of functional values.

8.5 Uses of Functional Values (HO Use in Table 1)

While very few programs have residual functional values, a substantial number make use of applications to a non-function/non-constructor, and use over-application of functions. In most cases these result from supplying error calls with additional arguments, typically related to the desugaring of **do** notation and pattern matching within Yhc.

8.6 Execution Time (Time in Table 1)

The timing results were all measured on a 1.2GHz laptop, running GHC 6.8.2 (The GHC Team 2007). The longest execution time was only one second, with the average time under half a second. The programs requiring most time made use of floating point numbers, suggesting that library code requires most effort to defunctionalise. If abstractions were given for library methods, the execution time would drop substantially.

In order to gain acceptable speed, we perform a number of optimisations over the algorithm presented in §4:

- We transform functions in an order determined by a topological sort with respect to the call-graph.
- We delay the transform of dictionary components, as these will often be eliminated.
- We fuse the inlining, arity raising and simplification stages.
- We track the arity and boxed lambda status of each function.

8.7 Program Size (Size in Table 1)

We measured program size by taking the number of nodes in the abstract syntax tree. On average the size of the resultant program is smaller by 33%. The reason for the decrease in program size is due to eliminating dictionaries holding references to unnecessary code.

9. Related Work

Reynolds style defunctionalisation (Reynolds 1972) is the seminal method for generating a first-order equivalent of a higher-order program.

Example 20

```
map f x = case x of
              []       → []
              (y : ys) → f y : map f ys
```

Reynolds method works by creating a data type to represent all values that f may take anywhere in the whole program. For instance, it might be:

```
data Function = Head | Tail

apply Head x = head x
apply Tail  x = tail  x

map f x = case x of
              []     → []
              y : ys → apply f y : map f ys
```

Now all calls to map head are replaced by map Head. □

Reynolds method works on all programs. Defunctionalised code is still type safe, but type checking would require a dependently typed language. Others have proposed variants of Reynolds' method that are type safe in the simply typed lambda calculus (Bell et al. 1997), and within a polymorphic type system (Pottier and Gauthier 2004).

The method is complete, removing all possible higher-order functions, and preserves space behaviour. The disadvantage is that the transformation essentially embeds a mini-interpreter for the original program into the new program. The control flow is complicated by the extra level of indirection and in practice the apply interpreter is a bottleneck for analysis.

Reynolds method has been used as a tool in program calculation (Danvy and Nielsen 2001; Hutton and Wright 2006), often as a mechanism for removing introduced continuations. Another use of Reynolds' method is for optimisation (Boquist and Johnsson 1996; Meacham 2008), allowing flow control information to be recovered without the complexity of higher-order transformation.

Removing Functional Values. The closest work to ours is Chin and Darlington (1996), which itself is similar to that of Nelan (1991). They define a higher-order removal method, with similar goals of removing functional values from a program. Their work shares some of the simplification rules, the arity raising and function specialisation. Despite these commonalities, there are big differences between their method and ours.

- Their method makes use of the types of expressions, information that must be maintained and extended to work with additional type systems.
- Their method does not have an inline step, or any notion of boxed lambdas. Functional values within constructors are ignored. The authors suggest the use of deforestation (Wadler 1988) to help remove them, but deforestation transforms the program more than necessary, and still fails to eliminate many functional values.
- Their specialisation step only applies to lambda expressions, not lambdas within constructors.
- To ensure termination of the specialisation step, they never specialise a recursive function unless it has all functional arguments passed identically in all recursive calls. This restriction is satisfied by higher-order functions such as map, but fails in many other cases.

In addition, functional programs now use monads, IO continuations and type classes as a matter of course. Such features were still experimental when their work was done and they did not handle them. Our work can be seen as a successor to theirs, indeed we achieve most of the aims set out in their future work section. We have tried their examples, and can confirm that all of them are successfully handled by our system. Some of their observations and extensions apply equally to our work: for example, they suggest possible methods of removing accumulating functions such as in Example 14.

Partial Evaluation and Supercompilation The specialisation and inlining steps are taken from existing program optimisers, as is the termination strategy of homeomorphic embedding. A lot of program optimisers include some form of specialisation and so remove some higher-order functions, such as partial evaluation (Jones et al. 1993) and supercompilation (Turchin 1986). We have certainly benefited from ideas in both these areas in developing our algorithms. Our initial attempt at removing functional values involved modifying a supercompiler (Mitchell and Runciman 2008b). But the optimiser is not attempting to preserve correspondence to the original program, so will optimise all aspects of the program equally, instead of focusing on the higher-order elements. Overall, the results were poor.

10. Conclusions and Future Work

Higher-order functions are very useful, but may pose difficulties for certain types of analysis. Using the method we have described, it is possible to remove most functional values from most programs. A user can still write higher-order programs, but an analysis tool can work on equivalent first-order programs.

Our method has already found practical use within the Catch tool, and we hope it can be of benefit to others. We have released our tool, both as a command line program, and as a library that analysis programs can invoke. Currently the implementation is based around the core language from the Yhc compiler, which restricts our input programs to the Haskell 98 language. By making use of the GHC front end, we could deal with many language extensions.

Our method is whole program, requiring sources for all function definitions. This requirement both increases transformation time, and precludes the use of closed source libraries. We may be able to relax this requirement, precomputing first-order variants of libraries, or permitting some components of the program to be ignored.

We have developed our method for analysis, not performance. However, for many simple examples, the resultant program performs better than the original. By restricting rules that reduce sharing, our defunctionalisation method may be appropriate for integration into an optimising compiler.

The use of a numeric termination bound in the homeomorphic embedding is regrettable, but practically motivated. We need further research to determine if such a numeric bound is necessary, or if other measures could be used.

Many analysis methods, in fields such as strictness analysis and termination analysis, start out first-order and are gradually extended to work in a higher-order language. Defunctionalisation offers an alternative approach, instead of extending the analysis method, we transform the functional values away, enabling more analysis methods to work on a greater range of programs.

Appendix A

The following script can be supplied to the AProVe system (Giesl et al. 2006) to check termination of our simplification rules, as described in §7.1.

```
[x,y,z]
app(lam(x),y)     -> let(y,x)
app(case(x,y),z) -> case(x,app(y,z))
app(let(x,y),z)  -> let(x,app(y,z))
case(let(x,y),z) -> let(x,case(y,z))
case(con(x),y)    -> let(x,y)
case(x,lam(y))    -> lam(case(x,app(lam(y),var)))
let(lam(x),y)     -> lam(let(x,y))
```

References

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proc. ICFP '97*, pages 25–37. ACM, 1997.

Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Proc IFL '96*, volume 1268 of *LNCS*, pages 58–84. Springer-Verlag, 1996.

Wei-Ngan Chin and John Darlington. A higher-order removal method. *Lisp Symb. Comput.*, 9(4):287–322, 1996.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc ICFP '07*, pages 315–326. ACM Press, October 2007.

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proc. PPDP '01*, pages 162–174. ACM, 2001.

J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNCS*, pages 281–286. Springer–Verlag, 2006.

Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. *The Monad.Reader*, (7):45–61, April 2007.

John Hughes. A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.*, 22(3):141–144, 1986.

Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, February 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.

Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. FPCA '85*, pages 190–203. Springer-Verlag New York, Inc., 1985.

Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *Proc. PEPM '94*, pages 107–117. ACM Press, June 1994.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The essence of computation: complexity, analysis, transformation*, pages 379–403. Springer-Verlag, 2002.

John Meacham. jhc: John's haskell compiler. `http://repetae.net/john/computer/jhc/`, 2008.

Neil Mitchell and Colin Runciman. Not all patterns, but enough – an automatic verifier for partial but sufficient pattern matching. 2008a. Submitted to ICFP 2008.

Neil Mitchell and Colin Runciman. Supercompilation for core Haskell. In *Selected Papers from the Proceedings of IFL 2007*, 2008b. To appear.

George Nelan. *Firstification*. PhD thesis, Arizona State University, December 1991.

Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Simon Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proc. ICFP '07*, pages 327–337. ACM Press, October 2007.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12:393–434, July 2002.

Simon Peyton Jones and Andrés Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming Workshops in Computing*, pages 184–204. Springer-Verlag, 1994.

François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *Proc. POPL '04*, pages 89–98. ACM Press, 2004.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM '72*, pages 717–740. ACM Press, 1972.

Damien Sereni. Termination analysis and call graph construction for higher-order functional programs. In *Proc. ICFP '07*, pages 71–84. ACM, 2007.

The GHC Team. The GHC compiler, version 6.8.2. `http://www.haskell.org/ghc/`, December 2007.

Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc ESOP '88*, volume 300 of *LNCS*, pages 344–358. Berlin: Springer-Verlag, 1988.

Philip Wadler. List comprehensions. In Simon Peyton Jones, editor, *Implementation of Functional Programming Languages*. Prentice Hall, 1987.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.