Losing Functions without Gaining Data
– another look at defunctionalisation

Neil Mitchell [*]
University of York, UK
ndmitchell@gmail.com

Colin Runciman
University of York, UK
colin@cs.york.ac.uk

Abstract

We describe a 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 is remarkably successful.

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

General Terms languages, transformation

Keywords Haskell, defunctionalisation, firstification

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 2008) or termination (Sereni 2007).

Example 1

Consider this definition of incList:

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

map :: (α → β) → [α] → [β]
map f []       = []
map f (x : xs) = f x : map f xs
```

The definition of incList has higher-order features. The expression $(+1)$ 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 longer (excluding the library function map), 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, this 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.

There are two significant limitations to our method:

1. The transformation can *reduce sharing*, causing the resulting program to be less efficient. Therefore our defunctionalisation method is not appropriate as a stage in compilation. But it works well when used as a preliminary stage in program analysis, effectively making first-order analyses applicable to higher-order programs: examples include analyses for safe pattern-matching and for termination.
2. The transformation is *not complete*. In some programs there may be residual higher-order expressions. However, the possible occurrences of such residual expressions can be characterised, and mild restrictions guarantee first-order results. In practice, our method is very often completely successful: for example defunctionalisation is complete for over 90% of the nofib benchmark programs.

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 2008), which checks for potential pattern-match errors in Haskell. Catch is a first-order analysis, and without a defunctionalisation method we wouldn't be able to apply Catch to real programs.

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 (§3, §4). Our method makes use of standard transformation steps, but with precise restrictions on their applicability.
- We show where higher-order elements may remain in a resultant program, and show that given certain restrictions we guarantee a first-order result (§6).

[*] This work was done while the first author was supported by an EPSRC PhD studentship

```
expr := λv → x              lambda abstraction
      | f xs                function application
      | c xs                constructor application
      | x xs                general application
      | v                   variable
      | let v = x in y      non-recursive let expression
      | case x of alts      case expression

alt  := c vs → x            case alternative

arityExpr [[λv → x]] = 1 + arityExpr x
arityExpr _          = 0
```

We let v range over locally defined variables, x and y over expressions, f over top-level function names and c over constructors.

Figure 1. Core Language.

- We identify restrictions which guarantee termination, but are not overly limiting (§7).
- We have implemented our method, and present measured results for much of the nofib benchmark suite (§8). Our method can deal with the complexities of a language like Haskell, including type classes, programs using continuation-passing style and monads.
- We show how to apply our results to existing analysis tools, using GHC's strictness analysis and Agda's termination checker as examples (§9).

2. Core Language

Our Core language is both pure and lazy. The expression type is given in Figure 1. A program is a mapping of function names to expressions, with a root function named main. The arity of a function is the result of applying arityExpr to its associated expression. 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. We allow full Haskell 98 data types, assuming a finite number of different constructors, each with a fixed arity.

The variable, case, application and lambda expressions are much as they would be in any Core language. We restrict ourselves to non-recursive let expressions. (Any recursive let expressions can be removed, with a possible increase in runtime complexity, using the methods described in (Mitchell 2008).) 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 using 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 and body f to denote the arity and body of function f. We use the function rhs to extract the expression on the right of a case alternative. We define the syntactic sugar f v = x to be equivalent to f = λv → x.

We assume that all Core programs are type correct. In particular we assume that when a program is evaluated a constructor application will never be the first argument of a general application, and a lambda expression will never be the subject of a case expression. All our transformations are semantics preserving, so maintain these two invariants.

Definition: A program is *higher-order* if it contains expressions which create or use functional values. An expression creates a functional value if it is a partially-applied function or a lambda expression which does not contribute to the arity of function definition. An expression uses a functional value if it is an over-applied function or a general application. □

Example 1 (revisited)

The original definition of incList is higher-order because it creates functional values with the partial applications of both map and (+). The original definition of map is higher-order because it uses functional values within a general application. In the defunctionalised version, the program is first-order. □

3. Our First-Order Reduction Method

Our method works by applying a set of rules non-deterministically until no further rules apply. The rules are grouped in to three categories:

Simplification: Many local simplification rules are used, most of which may be found in any optimising compiler (Peyton Jones and Santos 1994).

Inlining: Inlining is a standard technique in optimising compilers (Peyton Jones and Marlow 2002), and has been studied in depth. Inlining involves replacing an application of a function with the body of the function.

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). Specialisation involves generating a new function specialised with information about the functions arguments.

Each transformation has the possibility of removing some functional values, but the key contribution of this paper is *how they can be used together* – including which restrictions are necessary.

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.

3.1 Simplification

The simplification rules have two purposes: to remove some simple functional values, and to ensure a *normal form* so other rules can apply. The simplification rules are simple, and many are found in optimising compilers. All the rules are given in §4.1.

Example 2

```
one = (λx → x) 1
```

The simplification rule (lam-app) transforms this function to:

```
one = let x = 1 in x
```
□

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

Example 3

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

The simplification rule (let-lam) 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 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 4

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

eqInt = (primEqInt, primNeqInt)
```

Both components of the eqInt pair, 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 rules can now make the program first-order, using the rule (case-con) from §4.1.

```
main = primEqInt 1 2
```
□

3.3 Specialisation

We use specialisation to remove lambda expressions that are arguments of function applications. Specialisation creates alternative function definitions where some information is known about the arguments. In effect, some arguments are passed at transformation time.

Example 5

```
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 now eliminated all functional values. □

3.4 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 established transformations, total 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 not introducing any data types we avoid introducing an interpreter, which can be a bottleneck for subsequent analysis. By composing our transformation out of existing transformations, none of which introduces data types, we can easily ensure that our 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 always be applied after our transformation. Applying our method first will cause Reynolds' method to introduce fewer additional data types and generate a smaller interpreter.

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. It is possible to inline the let binding to give f x + f x, but this expression evaluates f x twice. This transformation is valid in Haskell due to referential transparency, and will preserve both semantics and termination, but may increase the amount of work performed. In an impure or strict language, such as ML (Milner et al. 1997), this transformation may change the semantics 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. However, we will avoid the loss of sharing where possible, so the program remains closer to the original.

Minimize the size of the program. A smaller program is likely to be faster for any subsequent analysis. Previous work has speculated that there may be a substantial increase in code-size after defunctionalisation (Chin and Darlington 1996).

Make the transformation fast. The implementation must be sufficiently fast to permit proper evaluation. Ideally, when combined with a subsequent analysis phase, the defunctionalisation should not take an excessive proportion of the runtime.

4. Method in Detail

This section gives a set of rules, which are applied non-deterministically. Many programs require a combination of rules to be applied, for example, the initial incList example requires simplification and specialisation rules.

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.

4.1 Simplification

The simplification rules aim to move lambda expressions upwards, and introduce lambdas for partially applied functions. The rules include 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. We believe that the combination of rules from Figures 2 and 3 are confluent.

$(\mathsf{x}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}}$
$\quad\Rightarrow \mathsf{x}\ \overline{\mathsf{xs}}\ \overline{\mathsf{ys}}$ (app-app)

$(\mathsf{f}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}}$
$\quad\Rightarrow \mathsf{f}\ \overline{\mathsf{xs}}\ \overline{\mathsf{ys}}$ (fun-app)

$(\lambda\mathsf{v} \rightarrow \mathsf{x})\ \mathsf{y}$
$\quad\Rightarrow \mathbf{let}\ \mathsf{v} = \mathsf{y}\ \mathbf{in}\ \mathsf{x}$ (lam-app)

$(\mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \mathsf{y})\ \mathsf{z}$
$\quad\Rightarrow \mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \mathsf{y}\ \mathsf{z}$ (let-app)

$(\mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \{\mathsf{p}_1 \rightarrow \mathsf{y}_1; \ldots; \mathsf{p}_n \rightarrow \mathsf{y}_n\})\ \mathsf{z}$
$\quad\Rightarrow \mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \{\mathsf{p}_1 \rightarrow \mathsf{y}_1\ \mathsf{z}; \ldots; \mathsf{p}_n \rightarrow \mathsf{y}_n\ \mathsf{z}\}$ (case-app)

$\mathbf{case}\ \mathsf{c}\ \overline{\mathsf{xs}}\ \mathbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \mathsf{y}; \ldots\}$
$\quad\Rightarrow \mathbf{let}\ \overline{\mathsf{vs}} = \overline{\mathsf{xs}}\ \mathbf{in}\ \mathsf{y}$ (case-con)

$\mathbf{case}\ (\mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \mathsf{y})\ \mathbf{of}\ \overline{\mathsf{alts}}$
$\quad\Rightarrow \mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ (\mathbf{case}\ \mathsf{y}\ \mathbf{of}\ \overline{\mathsf{alts}})$ (case-let)

$\mathbf{case}\ (\mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \mathsf{y}; \ldots\})\ \mathbf{of}\ \overline{\mathsf{alts}}$
$\quad\Rightarrow \mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \mathbf{case}\ \mathsf{y}\ \mathbf{of}\ \overline{\mathsf{alts}}; \ldots\}$ (case-case)

$\mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \lambda\mathsf{v} \rightarrow \mathsf{y}; \ldots\}$
$\quad\Rightarrow \lambda\mathsf{z} \rightarrow \mathbf{case}\ \mathsf{x}\ \mathbf{of}$ (case-lam)
$\qquad\qquad \{\ldots\mathsf{z}; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow (\lambda\mathsf{v} \rightarrow \mathsf{y})\ \mathsf{z}; \ldots\mathsf{z}\}$

$\mathsf{f}\ \overline{\mathsf{xs}}$
$\quad\Rightarrow \lambda\mathsf{v} \rightarrow \mathsf{f}\ \overline{\mathsf{xs}}\ \mathsf{v}$ (eta)
$\quad \mathbf{where}\ \mathsf{arity}\ \mathsf{f} > \mathsf{length}\ \overline{\mathsf{xs}}$

Figure 2. Standard Core simplification rules.

$\mathbf{let}\ \mathsf{v} = (\lambda\mathsf{w} \rightarrow \mathsf{x})\ \mathbf{in}\ \mathsf{y}$
$\quad\Rightarrow \mathsf{y}\ [\mathsf{v}\ /\ \lambda\mathsf{w} \rightarrow \mathsf{x}]$ (bind-lam)

$\mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \mathsf{y}$
$\quad\Rightarrow \mathsf{y}\ [\mathsf{v}\ /\ \mathsf{x}]$ (bind-box)
$\quad$ **where** x is a boxed lambda (see §4.2)

$\mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \lambda\mathsf{w} \rightarrow \mathsf{y}$
$\quad\Rightarrow \lambda\mathsf{w} \rightarrow \mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \mathsf{y}$ (let-lam)

Figure 3. Lambda Simplification rules.

4.1.1 Lambda Introduction

The (eta) rule inserts lambdas in preference to partial applications, 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 6

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

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

Here the function applications of $(\circ)$, not and odd are all partially applied. Three lambda expressions can be inserted using the (eta) rule:

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

Now all three function applications are fully-applied. The (eta) rule replaces partial application with lambda expressions, making functional values more explicit, which permits other transformations. □

4.1.2 Lambda Movement

The (bind-lam) rule inlines a lambda bound in a let expression. The (let-lam) rule can be responsible for a reduction in sharing:

Example 7

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

$\mathsf{main}\ \mathsf{xs} = \mathsf{map}\ (\mathsf{f}\ 1)\ \mathsf{xs}$

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

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

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

4.2 Inlining

We use inlining of top-level functions as the first stage in the removal of functional values stored within a constructor – for example Just $(\lambda\mathsf{x} \rightarrow \mathsf{x})$. To eliminate a functional value stored inside a constructor we eliminate the containing constructor by making it the subject of a case expression and using the (case-con) rule. We move the constructor towards the case expression using inlining.

We refer to expressions that evaluate to functional values inside constructors 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 body of the function definition is a boxed lambda; (2) the function application occurs as the subject of a case expression.

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

Example 8

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

$[\lambda\mathsf{x} \rightarrow \mathsf{x}]$
$\mathsf{Just}\ [\lambda\mathsf{x} \rightarrow \mathsf{x}]$
$\mathbf{let}\ \mathsf{y} = 1\ \mathbf{in}\ [\lambda\mathsf{x} \rightarrow \mathsf{x}]$
$[\mathsf{Nothing}, \mathsf{Just}\ (\lambda\mathsf{x} \rightarrow \mathsf{x})]$

```
isBox ⟦c $\overline{xs}$⟧             = any isLambda $\overline{xs}$ ∨ any isBox $\overline{xs}$
isBox ⟦let v = x in y⟧      = isBox y
isBox ⟦case x of $\overline{alts}$⟧     = any (isBox ∘ rhs) $\overline{alts}$
isBox ⟦f $\overline{xs}$⟧              = isBox (fromLambda (body f))
isBox _                  = False

fromLambda ⟦λv → x⟧ = fromLambda x
fromLambda x         = x

isLambda ⟦λv → x⟧    = True
isLambda _           = False
```

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.

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

The following are *not* boxed lambdas:

```
λx → [x]
[id (λx → x)]
id [λx → x]
let v = [λx → x] in v
```

The final three expressions all *evaluate to* a boxed lambda, but are not themselves boxed lambdas. □

Definition: The inlining transformation is specified by:

```
case (f $\overline{xs}$) of $\overline{alts}$
  ⇒ case (y $\overline{xs}$) of $\overline{alts}$
  where
    y = body f
    If isBox (f $\overline{xs}$)  evaluates to True
```
□

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

4.3 Specialisation

For each application of a top-level function in which at least one argument has a lambda subexpression, 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 subexpressions are omitted, denoted by the • symbol. The process of specialisation proceeds as follows:

1. Find all function applications which need specialising, and generate templates (see §4.3.1).
2. Abstract templates, replacing some subexpressions with • (see §4.3.2).
3. For each template, generate a function definition specialised to that template (see §4.3.3).
4. For each expression matching a template, replace it with the generated function (see §4.3.4).

Example 9

```
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 (λx → x) xs. Next it abstracts the template to map (λx → 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. □

4.3.1 Generating Templates

The idea is to generate templates for all function applications which pass functional values. Given an expression e, a template is generated if: (1) e is a function application; and (2) at least one of the subexpressions of e is either a lambda or a boxed lambda (see §4.2). In all cases, the template generated is simply e.

Example 10

The following expressions generate templates:

```
id (λx → x)
map f [λx → x]
id (Just (λx → x + 1))
f (λv → v) True
```

□

4.3.2 Abstracting Templates

We perform abstraction to reduce the number of different templates required, by replacing non-functional expressions with •. For each subexpression e in a template, it can be replaced with • if the following two conditions hold:

1. e is not, and does not contain, any expressions which are either lambda expressions or boxed lambdas, e.g. we cannot substitute • for (λx → x) or (**let** y = λx → x **in** y).
2. None of the free variables in e are bound in the template, e.g. we cannot replace the expression f v with • in (**let** v = 1 **in** f v), as the variable v is bound within the template.

Example 11

Template	Abstract 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 these 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 •. □

4.3.3 Generating Functions

To generate a function from a template we first pick a unique function name. For each occurrence of • in the template a fresh variable is substituted. The body of the new function uses all the fresh variables as lambda variables, then is produced by inlining the outer function symbol of the template. If a previous specialisation has already generated a function for this template, we reuse the previous function.

Example 9 (revisited)

Consider the template map $(\lambda x \rightarrow x)$ •. Let v_1 be the fresh argument variable for the single • placeholder, and map_id be the function name:

```
map_id = λv₁ → map (λx → x) v₁
```

We inline the definition of map:

```
map_id = λv₁ → (λf xs → case xs of
                          []     → []
                          y : ys → f y : map f ys)
               (λx → x) v₁
```

After the simplification rules from Figure 3, we obtain:

```
map_id = λv₁ → let xs = v₁
               in  case xs of
                     []     → []
                     y : ys → y : map (λx → x) ys
```

□

4.3.4 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 • is passed as an argument.

Example 9 (continued)

```
map_id = λv₁ → let xs = v₁
               in  case xs of
                     []     → []
                     y : ys → y : map_id ys
```

We now have a first-order definition. □

4.4 Confluence

The transformations we have presented are not confluent. Consider the expression id $((\lambda x \rightarrow x)\ 1)$. We can either apply specialisation, or the (lam-app) rule. The first will involve the creation of an additional function definition, while the second will not.

We conjecture that the rules in each of the separate categories are confluent. In order to ensure a deterministic application of the rules we always favour rules first from the simplification stage, then the inlining stage, and finally the specialisation stage. By choosing the above order, we reduce the generation of auxiliary top-level functions, which should lead to a simpler result.

4.5 Primitive Functions

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

1. We define that a primitive application is *not* a boxed lambda, and has an arity derived from its type.
2. We restrict specialisation so that if the function to be specialised is a primitive, no template is generated. This restriction is necessary because specialisation requires inlining the function, which is not possible for a primitive.

These restrictions mean that some programs using primitive functions cannot be made first-order.

Example 12

```
main = seq (λx → x) 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 $\alpha \rightarrow (\text{IOError} \rightarrow \alpha) \rightarrow \alpha$, taking an exception handler as one of the arguments. □

4.6 Recovering Input Expressions

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

5. Examples

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

Example 13 (Inlining Boxed Lambdas)

An earlier version of our defunctionaliser inlined boxed lambdas everywhere they occurred. Inlining boxed lambdas means the isBox function does not have to examine the body of applied functions, and is therefore simpler. However, it was unable to cope with programs like this one:

```
main = map (λx → x 1) gen
gen = (λx → x) : gen
```

The gen function is both a boxed lambda and recursive. If we inlined gen initially 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:

```
nil = id
snoc x xs = λys → xs (x : ys)
list xs = xs []
```

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:

```
nil = []
snoc = (:)
list = reverse
```

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 required by the function-based solution. We suspect there are sometimes efficiency savings to storing data in closures, but not for snoc-based lists. □

6. Restricted Completeness

Our method would be *complete* if it made all programs first-order. In this section we give three conditions, which if met, ensure a program can be made first-order. We start by showing where expressions creating functional values can reside in a resultant program, then show that this information enables us to make programs first-order.

6.1 Proposition

After transformation, there will be no partial applications, and all lambda expressions will either contribute to the arity of a function definition or be unreachable (never be evaluated at runtime), provided:

1. The termination criteria do not curtail defunctionalisation (see §7).
2. No primitive function receives a functional argument, nor returns a functional result.
3. The main function has a type that ensures it neither receives a functional argument, nor returns a functional result.

We prove this proposition with a series of lemmas about the resultant program.

6.2 Lemmas

We define the root of a function to be its body after applying the fromLambda function from Figure 4. We define a higher-order lambda (HO lambda) to be a lambda expression that does not contribute to the arity of a function definition.

Lemma: *No partial applications*
The (eta) rule removes partial application, and at the end of the transformation, no further rules apply – therefore there can be no partial applications in the resultant program. □

Lemma: *The first argument of a general application must be a variable*
The rules (app-app), (fun-app), (lam-app), (let-app) and (case-app) mean the first argument to a general application must be a variable or a constructor application. All constructor applications are fully applied, and therefore cannot return a functional value, so type safety ensures they cannot be the first argument of an application. Therefore, the first argument of an application is a variable. □

Lemma: *A HO lambda may only occur in the following places: inside a HO lambda; as an argument to an application or a constructor*
A lambda cannot be the subject of a case expression as it would not be well typed. A lambda cannot be an argument to a function as it would be removed by specialisation. All other possible lambda positions are removed by the rules (lam-app), (case-lam), (bind-lam) and (let-lam). □

Lemma: *A boxed lambda may only occur in the following places: the root of a function; inside a HO lambda or boxed lambda; as an argument to an application*
Using the definition of isBox from Figure 4 to ignore expressions which are themselves boxed lambdas, the only possible locations of a boxed lambda not mentioned in the lemma are the binding of a let, the subject of a case, and as an argument to a function application. We remove the binding of a let with (bind-box) and the argument to a function application with specialisation.

To remove a boxed lambda from the subject of a case we observe that a boxed lambda must be a constructor application, a let expression, a case expression or a function application. The first three are removed with the rules (case-con), (case-let) and (case-case), the final one is removed by inlining. □

Lemma: *A boxed lambda must have a type that permits a functional value*
An expression must have a type that permits a functional value if any execution, choosing any alternative in a case expression, evaluates to a functional value. The base case of a boxed lambda is a constructor application to a lambda, which is a functional value. For let and case, the type of the expression is the type of the contained boxed lambda. The remaining case is if $((\lambda\overline{\mathsf{vs}} \rightarrow \mathsf{b})\ \overline{\mathsf{xs}})$ evaluates to a functional value. As b must be a boxed lambda, i.e. a constructor wrapping a lambda, any application and abstraction operations alone cannot remove the constructor, so cannot remove the functional value. □

Lemma: *A function whose root is a boxed lambda must be called from inside a HO lambda or as the argument of an application*
An application of a function whose root is a boxed lambda is itself a boxed lambda. Therefore the restrictions on where a boxed lambda can reside apply to applications of these functions. □

Lemma: *All HO lambdas are unreachable*
The main function cannot be a boxed lambda, as that would be a functional value, and is disallowed by restrictions on main. There remain only four possible locations for HO lambdas or boxed lambdas:

1. As an argument to an application $(\mathsf{v}\ \bullet)$.
2. As the body of a HO lambda $(\lambda\mathsf{v} \rightarrow \bullet)$.
3. Contained within a boxed lambda.
4. As the root of a function definition, whose applications are boxed lambdas.

None of these constructs binds a functional value to a variable, therefore in the first case v cannot be bound to a functional value. If v is not a functional value, then type checking means that v *must* evaluate to $\perp$, and $\bullet$ will never be evaluated. In the remaining three cases, the lambda or boxed lambda must ultimately be contained within an application whose variable evaluates to $\perp$ – and therefore will not be evaluated. □

Lemma: *There are no partial applications and all lambda expressions either contribute to the arity of a function definition or are unreachable*
By combining the lemmas that there are no partial applications and that all HO lambdas are unreachable. □

It is instructive to note that during the proof every rule has been used, and that the removal of any single rule would invalidate the proof. While this does not prove that each step is necessary, it does provide a motivation for each rule.

6.3 Residual Higher-Order Programs

The following programs all remain higher-order after applying our method, although none will actually create higher-order values at runtime.

```
[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))
```

Figure 5. Encoding of termination simplification.

Example 15

```
main = bottom (λx → x)
```

We use the expression bottom to indicate a computation that evaluates to ⊥ – either a call to error or a non-terminating computation. The function main will evaluate to ⊥, without ever evaluating the contained lambda expression. □

Example 16

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

In this example the lambda expression is never reached because the Just branch of the case expression is never taken. □

6.4 Transformation to First-Order

Our proposition means that, provided the three restrictions are met, replacing all lambda expressions in the resultant program with ⊥ will give an identical program. In addition, any uses of functional values are guaranteed to actually be operating on ⊥, as no functional values could have been created. Another way of viewing the proposition is that after transformation the program will be *first-order at runtime*, even if there are expressions that create or use functional values in the source program. Therefore, the following rewrites are valid:

$(\lambda v \rightarrow x) \Rightarrow \bot$ if not contributing to the arity of a function
$x\ \overline{xs} \Rightarrow x$
$f\ \overline{xs} \Rightarrow f\ (\mathsf{take}\ (\mathsf{arity}\ f)\ \overline{xs})$

After applying the (eta) rule and performing these rewrites, all programs are guaranteed to be first-order.

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 rules. 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. An encoding of a simplified version of the rules from Figures 2 and 3 is given in Figure 5. We have encoded rules by considering what type of expression is transformed by a rule. For example, the rule replacing (λv → x) y with **let** v = y **in** x is expressed as a rewrite replacing app(lam(x),y) with let(y,x). The names of binding variables within expressions have been ignored. To simplify the encoding, we have only considered applications with one argument. The rewrite rules are applied non-deterministically at any suitable location, so faithfully model the behaviour of our original rules.

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) 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 limit is intended to give a strong guarantee of termination, and will only rarely be necessary – hence the high bound.

7.2 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 as it would leave residual lambda expressions in programs such as Example 13. We first present a program which causes our method to fail to terminate, then our means of ensuring termination.

Example 17

```
data B α = B α
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 use sites within the definition of a function g, but only once per pair (f, g). In the previous 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 be at most n^2 inlining steps, each for possibly many application sites.

7.3 Termination of Specialisation

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

Example 18

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

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). The relation $x \trianglelefteq y$ indicates the expression x is an embedding of y. We can define $\trianglelefteq$ using the following rewrite rule:

$$emb = \{f(x_1, \ldots, x_n) \rightarrow x_i \mid 1 \leqslant i \leqslant n\}$$

Now $x \trianglelefteq y$ can be defined as $x \leftarrow^*_{emb} y$ (Baader and Nipkow 1998). The rule emb takes an expression, and replaces it with one of its immediate subexpressions. If repeated non-deterministic application of this rule to any subexpression transforms y to x, then $x \trianglelefteq y$. The intuition is that by removing some parts of y we obtain x, or that x is somehow "contained" within y.

Example 19

$$\begin{array}{rr} a \trianglelefteq a & b(a) \ntrianglelefteq a \\ a \trianglelefteq b(a) & a \ntrianglelefteq b(c) \\ c(a) \trianglelefteq c(b(a)) & d(a,a) \ntrianglelefteq d(b(a),c) \\ d(a,a) \trianglelefteq d(b(a),c(c(a))) & b(a,a) \ntrianglelefteq b(a,a,a) \end{array}$$

□

The homeomorphic embedding $\trianglelefteq$ is a well-quasi order, meaning that for every infinite sequence of expressions $e_1, e_2 \ldots$ over a finite alphabet, there exist indicies $i < j$ such that $e_i \trianglelefteq e_j$. This result is sometimes used in program optimisation to ensure an algorithm over expressions performs a bounded number of iterations, by stopping at iteration n once $\exists i \bullet 1 \leqslant i < n \wedge e_i \trianglelefteq e_n$ – for example by Jonsson and Nordlander (2009).

For each function definition, we associate a set of expressions S. After generating a template t, we only specialise with that template if $\forall s \in S \bullet s \ntrianglelefteq t$. After specialising an expression e with template t, we add t to the set S associated with the function definition containing e. When we generate a new function from a template, we copy the S associated with the function at the root of the template.

One of the conditions for termination of homeomorphic embedding is that there must be a finite alphabet. To ensure this condition, we consider all variables to be equivalent. However, this is not sufficient. During the process of specialisation we generate new function names, and these names are new symbols in our alphabet. To keep the alphabet finite we only use function names from the original input program, relying on the equivalence of each template to an expression in the original program (§4.6). We perform the homeomorphic embedding test only after transforming all templates into their original equivalent expression.

Example 18 (revisited)

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

Forbidding homeomorphic embeddings in specialisation still allows full defunctionalisation in most simple examples, but there are examples where it terminates prematurely.

Example 20

```
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 frequently with the standard implementation of type classes as 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 invariant. Each of the sets S_i is still finite, and there are a finite number (n) of them, so termination is guaranteed.

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

7.4 Termination as a Whole

Given an initial program, inlining and specialisation rules will only apply a finite number of times. The simplification rules are terminating on their own, so when combined, all the rules will terminate.

8. Results

8.1 Benchmark Tests

We have tested our method with programs drawn from the nofib benchmark suite (Partain et al. 2008), and the results are given in Table 1. Looking at the input Core programs, we see many sources of functional values.

- Type classes 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 continuation-passing style. There are other translations which require fewer functional value manipulations (Coutts et al. 2007).

We have tested all 14 programs from the imaginary section of the nofib suite, 35 of the 47 spectral programs, and 17 of the 30 real programs. The remaining 25 programs do not compile using the Yhc compiler, mainly due to missing or incomplete libraries. After applying our defunctionalisation method, 4 programs are curtailed by the termination bound and 2 pass functional values to primitives. The remaining 60 programs can be transformed to first-order as described in §6.4. We first discuss the resultant programs which remain higher-order, then those which contain higher-order expressions but can be rewritten as first-order, then make some observations about each of the columns in the table.

8.2 Higher-Order Programs

All four programs curtailed by the termination bound are listed in Table 1. The lift program uses pretty-printing combinators, while

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 §7.3); **HO Create** is the number of under-applied functions and lambda expressions not contributing to the arity of a top-level function, first in the input program and then in the output program; **HO Use** is the number of over-applied functions and application expressions; **Time** is the execution time of our method in seconds; **Size** is the change in the program size measured by the number of lines of Core.

Name	Bound	HO Create		HO Use		Time	Size
Programs curtailed by a termination bound:							
cacheprof	8	611	44	686	40	1.8	2%
grep	8	129	9	108	22	0.8	40%
lift	8	187	123	175	125	1.2	-6%
prolog	8	308	301	203	137	1.1	-5%
All other programs:							
ansi	4	239	0	187	2	0.5	-29%
bernouilli	4	240	0	190	2	0.3	-32%
bspt	4	262	0	264	1	0.7	-22%
. . . plus 56 additional programs . . .							
sphere	4	343	0	366	2	0.7	-45%
symalg	5	402	0	453	64	1.0	-32%
x2n1	4	345	0	385	2	0.8	-57%
Summary of all 62 other programs:							
Minimum	2	60	0	46	0	0.1	-78%
Maximum	14	580	1	581	100	1.2	27%
Average	5	260	0	232	5	0.5	-30%

the other three programs use parser combinators. In all programs, the combinators are used to build up a functional value representing the action to perform, storing an unbounded amount of information inside the functional value, which therefore cannot be removed.

The remaining two higher-order programs are integer and maillist, both of which pass functional values to primitive functions. The maillist program calls the catch function (see §4.5). The integer program passes functional values to the seq primitive, 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.

8.3 First-Order Programs

Of the 66 programs tested, 60 can be made first-order using the rewrites given in §6.4. When looking at the resultant programs, 3 contain lambda expressions, and all but 5 contain expressions which could use functional values.

The pretty, constraints and mkhprog programs pass functional values to expressions that evaluate to $\perp$. The case in pretty comes from the fragment:

```
type Pretty = Int → Bool → PrettyRep

ppBesides :: [Pretty] → Pretty
ppBesides = foldr1 ppBeside
```

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

The vast majority of programs which have residual uses of functional values result from over-applying the error function, because Yhc generates such an expression when it desugars a pattern-match within a **do** expression.

8.4 Termination Bound

The termination bound required varies from 2 to 11 for the sample programs (see Bound in Table 1). 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.

8.5 Creation and Uses of Functional Values

We use Yhc generated programs as input. Yhc performs desugaring of the Haskell source code, introducing dictionaries of functions to implement type classes, and performing lambda lifting (Johnsson 1985). As a result the input programs have no lambda expressions, only partial application. Conversely, the (eta) rule from Figure 2 ensures resultant programs have no partial application, only lambda expressions. Most programs in our test suite start with hundreds of partial applications, but only 9 resultant programs contain lambda expressions (see HO Create in Table 1).

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 reduces the number and complexity of functional values.

8.6 Execution Time

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 just over one second, with the average time being half a second (see Time in Table 1). 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 method presented in §4. (1) We transform functions in an order determined by a topological sort with respect to the call-graph. (2) We delay the transformation of dictionary components, as these will often be eliminated. (3) We track the arity and boxed lambda status of each function.

8.7 Program Size

We measure program size by counting the number of lines of Core code, after a simple dead-code analysis to remove entirely unused function definitions. On average the size of the resultant program is smaller by 30% (see Size in Table 1). The decrease in program size is mainly due to the elimination of dictionaries holding references to unnecessary code. An optimising compiler will perform dictionary specialisation, and therefore is likely to also reduce program size. We do not claim that defunctionalisation reduces code size, merely hope to alleviate concerns raised by previous papers that it might cause an explosion in code size (Chin and Darlington 1996).

9. Higher-Order Analysis

In this section we show that our method can be used to improve the results of existing analysis operations. Our method is already used by the Catch tool (Mitchell and Runciman 2008), allowing a first-order pattern-match analysis to check higher-order programs. We now give examples of applying our method to strictness and termination analysis.

Example 21

GHC's demand analysis (The GHC Team 2007) is responsible for determining which arguments to a function are strict.

```
main :: Int → Int → Int
main x y = apply 10 (+x) y

apply :: Int → (α → α) → α → α
apply 0 f x = x
apply n f x = apply (n − 1) f (f x)
```

GHC's demand analysis reports that the main function is lazy in both arguments. By generating a first-order variant of main and then applying the demand analysis, we find that the argument y is strict. This strictness information can then be applied back to the original program. □

Example 22

The Agda compiler (Norell 2008) checks that each function is terminating, using an analysis taken from the Foetus termination checker (Abel 1998).

```
cons : (ℕ → List ℕ) → ℕ → List ℕ
cons f x = x :: f x

downFrom : ℕ → List ℕ
downFrom = cons f
  where f : ℕ → List ℕ
        f zero    = []
        f (suc x) = downFrom x
```

Agda's termination analysis reports that downFrom may not terminate. By generating a first-order variant and applying the termination analysis, we find that downFrom is terminating. □

No doubt there are other ways in which the above analysis methods could be improved, by extending and reworking the analysis machinery itself. But a big advantage of adding a preliminary defunctionalisation stage is that it is modular: the analysis is treated as a black box. A combination with Reynolds style defunctionalisation does not improve either analysis.

10. Related Work

10.1 Reynolds style defunctionalisation

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

Example 23

```
map f []       = []
map f (x : xs) = f x : map f xs
```

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 []       = []
map f (x : xs) = apply f x : map f xs
```

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 higher-order functions, and preserves space and time 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 the apply interpreter can be a bottleneck for analysis. Various analysis methods have been proposed to reduce the size of the apply function, by statically determining a safe subset of the possible functional values at a call site (Cejtin et al. 2000; Boquist and Johnsson 1996).

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 (Meacham 2008), allowing flow control information to be recovered without the complexity of higher-order transformation.

10.2 Removing Functional Values

The closest work to ours is by Chin and Darlington (1996), which itself is similar to that of Nelan (1991). They define a defunctionalisation method which removes some functional values without introducing data types. Their work shares some of the simplification rules, and includes a form of 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 has *no inlining* 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 outermost 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 Chin and Darlington developed their method and it 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.

10.3 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 method.

11. 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, allowing a first-order pattern-match analysis to be applied to real Haskell programs. It would be interesting to investigate the relative accuracy of higher-order analysis methods with and without defunctionalisation.

Our method works on whole programs, 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.

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.

References

Andreas Abel. foetus – Termination Checker for Simple Functional Programs. Programming Lab Report, July 1998.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

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.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Proc. ESOP '00*, volume 1782 of *LNCS*, pages 56–71. Springer–Verlang, 2000.

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*, 1(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.

Peter Jonsson and Johan Nordlander. Positive supercompilation for a higher order call-by-value language. In *POPL '09*, pages 277–288. ACM, 2009.

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.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

Neil Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, 2008.

Neil Mitchell and Colin Runciman. Not all patterns, but enough – an automatic verifier for partial but sufficient pattern matching. In *Proc. Haskell '08*, 2008.

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

Ulf Norell. Dependently typed programming in Agda. In *Lecture notes on Advanced Functional Programming*, 2008.

Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2008.

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