Computer Science at Kent

Transformation in HaRe

Chau Nguyen-Viet

Technical Report No. 21-04
December 2004

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK

Transformation in HaRe

Chau Nguyen-Viet
Computing Laboratory
University of Kent

December 9, 2004

Abstract

HaRe [**?**] is a system developed at the University of Kent Computing Laboratory to support refactoring in Haskell. We also want HaRe to be an open platform to support general Haskell program transformation so it can be used by other researchers in the field. This paper demonstrates the facilities HaRe provides for program transformation by implementing a deforestation transformation as a case study.

1 Introduction

Two desirable characteristics of a program are lucidity and efficiency. Unfortunately, one is usually gained at the expense of the other. A clean and understandable program is essential for maintainability and scalability in a software developer's view. At execution level, the only thing that matters is the efficiency of the program. Ideally, we want to have a general transformation that converts clear, but inefficient code written by developers, to an efficient but possibly obscure code to execute. Although a single solution does not exist there are many different techniques to autonomously optimise a program. One of them is deforestation, a transformation that eliminates intermediate data structures. In this paper, we present a partial implementation of the warm fusion deforestation proposed by Launchbury and Sheard in [**?**]. In doing so, we show that HaRe can be used effectively as an open platform for developing program transformations for Haskell. Readers can also find in this paper some general information about the HaRe API.

2 Background

2.1 Deforestation

Haskell programs often contains many intermediate data structures, which are used in the computation process but does not constitute a part of the result. For example in Figure **??** we define 3

```
evens n
        | n == 0     = []
        | n mod 2    == 0 =  n : evens (n-2)
        | otherwise  = evens (n-1)

sum []     = 0
sum (x:xs) = x + sum xs

sumEvens n = sum (evens n )
```

Figure 1: Example Haskell functions

```
sumEvens n
          | n == 0      = 0
          | n 'mod' 2   == 0 = n + sumEvens (n-2)
          | otherwise   = sumEvens (n-1)
```

Figure 2: Example functions transformed

functions.

Whenever the function `sumEvens` is called, an integer list is created by the `evens` function. This list serves as an intermediate list and is not part of the result. List creation in Haskell is expensive in both memory usage and computation.

A more efficient version of `sumEvens` is shown in Figure **??**; no list is produced in this version of `sumEvens`. However the first version is more intuitive and easier to understand. Also the first version posses a level of modularity, an important characteristic in software development and evolution.

Deforestation is a family of transformations that convert program written in the style of first version to a more efficient program like the second version of the `sumEvens` function. In most cases, a deforestation transformation searches for a function, which consists of two parts: the first one produces some data and the second consumes that data. In our example the first is the `evens` function and the second is the `sum` function. If possible, the transformation will merge the two functions into one, eliminating the intermediate data structure.

2.2 Warm fusion

There are several known deforestation algorithms. In this paper we build a (partial) implementation of the warm fusion algorithm presented by Launchbury and Sheard in [**?**] as an example of building a transformation using HaRe. Warm fusion is a combination of the cheap deforestation of Gill [**?**, **?**] and the fold promotion theory of Sheard and Fegaras [**?**]. This section contains a brief discussion of the algorithm; more details about warm fusion can be found in [**?**]. The main idea of the algorithm is that many list manipulation functions can be written in term of the standard list consuming function `foldr` and standard list producing function `build`. The function `foldr` is defined thus

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z (x:xs) = f x (foldr f z xs)
foldr f z []     = z
```

The `build` function abstracts the constructors inside the list and is defined by

```
build g = g (:) []
```

Many standard list manipulation functions can be rewritten using `foldr` and `build` . For example,

```
map f xs = build(\c n -> foldr(\x ys -> c (f x) ys) n xs))

sum xs = foldr (+) 0 xs
```

If a function uses `foldr` and `build` we can apply the following rule :

```
foldr f z (build g) = g f z                                    (R1)
```

The 'cheap' deforestation algorithm, also called *short-cut deforestation*, rewrites all standard list manipulation functions in Haskell using `build` and `foldr`. It then inlines the definition of those functions in their call sites and if possible applies the rule `(R1)`. An obvious problem is that we can not expect Haskell coders to write their codes using the `foldr` and `build` function. We need to convert user defined functions to `foldr` and `build` form before we can fuse them using rule `(R1)`. The warn fusion algorithm can convert list producing function to `build` form and list consuming function to `foldr` form.

In this work, we only implemented the first part, which converts a general list producing function to a function that uses the `build` function. This task is solved by first searching for all list producing function in a program. If a function uses only the list constructor `:` and `[]` to construct the result we can abstract this function and replace `(:)`, `[]` by variables `c` and `n`. We do not however replace the old function definition but introduce a worker/wrapper pair for it. This technique is presented in [**?**]. For example, the `evens` function defined in Figure **??** satisfies all the requirements so a new worker/wrapper pair can be produced; this is shown in Figure **??**.

In the remainder of the program, all calls to `evens x` will be replaced by calls of the form

```
build (evensWorker x)
```

thus introducing the `build` function. After constructing a list of all list producing functions, the transformation will inline all the new definitions and try to apply rule `(R1)`.

```
evensWrapper = build evenWorker

evensWorker x c n
  | x == 0          = n
  | x 'mod' 2 ==    = c (x) (evensWorker (x-2) c n )
  | otherwise       = evensWorker (x-2) c n)
```

Figure 3: Worker and wrapper functions

3 Implementation in HaRe

3.1 HaRe overview

Built on top of the Programatica platform [**?**] and using Strafunski [**?**] for term traversal, HaRe provides a rich environment for developing term-rewriting program transformations in Haskell. A Haskell program is parsed and analysed by Programatica to build an abstract syntax tree. The Programmatica platform was built to provide developers a set of line commands to inspect Haskell programs. Programmatica defines a data type for each Haskell syntax term e.g. expression, operator, constant. Through HaRe, the syntax tree is accessible for transformation developer. He/she can then change any syntax phrase in the tree using low level Strafunski strategies or in most cases functions provided by the HaRe library of syntax manipulation functions. Strafunski provides programming support for generic traversal as useful for the implementation of program analyses and transformation components of language processors. Strafunski is based on the notion of a functional strategy. These are generic functions that can traverse into terms of any type while mixing type-specific and uniform behaviour.

One feature of the HaRe library functions that is crucial for the usability of the system is that the library functions preserve the layout of the modified code. This is a very desirable characteristic but also a non-trivial task. Using the Programatica infrastructure it is necessary to retain the token stream to keep comment and layout information; if the coder wants to modify a syntax phrase using Strafunski, he must also modify the token streams of the code in order to preserve the layout. The layout is preserved by the HaRe library functions.

3.2 The algorithm design

In the next section, we will explain the main steps of our implementation of warm fusion:

- Simplify function definition
- Abstract list producing function
- Search for fold/build pair
- Rewrite the foldr/build pair using fusion.

In Haskell, there are many different formats to describe a function including pattern matching, guards, cases. The simplification phase converts all function definitions into a standard form. The second phase checks if a function is list producing and introduces new worker/wrapper pair if so. The outcome of the second phase is a list of all functions that have been converted successfully. The third phase performs a top-down traversal through the syntax tree and find all foldr/build pairs. The final phase replaces these using the fusion rule `(R1)` in a top down traversal.

3.3 Implementation

Typically, a transformation developed in HaRe will be passed the name of the source code file as a parameter. The file name is then passed to the `parseSourceFile` function, which is provided by the HaRe API. This function returns pointers to the list of all defined functions, the export list, the abstract syntax tree and the token stream of the Haskell source file. The transformation then modifies the abstract syntax tree. If necessary the transformation can get the list of all files that import the current module for modification, since in general a transformation may affect each file in a project.

3.3.1 Simplification phase

This phase is as easy as it can get. The HaRe library provides a powerful function `simplifyDec` [**?**] which does exactly what we want. It converts all function definitions to a standard format using `case` expressions for all parameters matching functions and `if/else` for function definitions using guards.All functions that are already in `case` or `if/else` format will remain unchaged. For example

```
f 0 a =   a
f n a =  (f (n-1) a) * a
```

will be converted to

```
f x1 x2 = case (x1,x2) of
     (0, a) -> a
     (n, a) -> (f (n-1) a) * a
```

and

```
f a b | b == 0 = a
      | otherwise = ( f (n-1) a) * a
```

will be converted to

```
-- 'isListCons' takes an Exp and returns True
--  if it is an (:) list constructor
isListCons (HsCon (PNT (PN (UnQual ":")
             (G (PlainModule "Prelude") ":"_ )) _ _))    = True
isListCons _                                            = False


--'expHasCons' checks if an Exp is in the form :
        --  '(:) exp1 exp2' or ' exp1 : exp2'.
expHasCons (Exp (HsInfixApp _ cons _))    =  isListCons cons
expHasCons (Exp (HsApp (Exp (HsApp (Exp (HsId(cons))) exp1)) exp2))
                                          = isListCons cons
expHasCons (Exp(HsList _))                = True
expHasCons _                              = False
```

Figure 4: Recognising list producing functions

```
f a b = if (b==0) then 0 else (f (n-1) a ) * a
```

The `simplifyDec` function converts a single function definition. To convert all functions in a module we need to do a top-down traversal through its syntax tree to apply the `simplifyDec` function to all function definitions. This is realised by the function `simpDec`.

```
simpDec mod = fromJust (applyTP (full_tdTP (idTP `adhocTP` simplifyDecl)) mod)
```

The functions `applyTP`, `full_tdTP`, `idTP` and `adhocTP` are from Strafunski library. More information on Stranfunski can be found in [9] but basically what they do is: travel the syntax tree `mod` in top-down manner, if the term, or node/syntax entity, it visits, matches the type of the parameters of the `simplifyDec` function then applies the `simplifyDec` to the node and replace it with the function's result. The `simplifyDec` takes a definition declaration as it parameter thus the result of the right hand side expression is a new syntax tree with all the function definition rewritten into the unified format.

3.3.2 Abstracting list producing functions

This is the most difficult part of the transformation. Every function declared in the current module is examined. If its result is a list of some kind and it only uses the list constructor `(:)`, the empty list `[]` to construct the result list, the `(:)` operator must be used in the outermost position, then the function is qualified to be change to `build` form. In Figure **??** we show some of the functions used to recognize list production functions.

Once a list producing function is identified, a new worker function is introduced. The worker function is build from the old function with 2 new parameters `c` and `n`. On the right hand side

```
dup 0 a = []
dup n a = a : (dup (n-1) a)

-- after the first phase
dup x1 x2 = case (x1,x2) of
                       (0,a) -> []
                       (n,a) -> a : (dup (n-1) a)

-- the worker function introduced by phase 2
dup_worder x1 x2 c n = case (x1,x2) of
                       (0,a) -> n
                       (n,a) -> c a (dup (n-1) a)
```

Figure 5: Example of function transformation

```
-- 'isFold' checks if a syntax phrase represents the
--  standard foldr function.
    containsFoldr (Exp (HsApp (Exp (HsApp (Exp (HsApp fol f)) n)) exp))
                    = (isFold fol) && (elem exp workerList)
    containsFoldr _ = False
```

Figure 6: Searching for occurrences of `foldr`

of the worker definition all occurrences of `[]` will be replaced by `n` and the outermost `:` with `c`. Figure **??** shows a complete example of how the transformation works on a function:

At the end of the phase a list of all new worker functions with their names is produced:

```
-- the 'convertDec' converts a list producing function
--  to its corresponding worker
workerList = applyTU (full_tdTU (constTU[] 'adhocTU'  convertDec ) ) mod
```

The function list is then added to the current module by the `addDefDecl`[**?**] function from the HaRe library.

3.3.3 Searching for `foldr/build`

After the second phase, we have a list of all list producing function. We then search for all occurrences of `foldr` in the project space. If the third parameter of a `foldr` is a call of one of the function in our list, a fusion can take place. Figure **??** shows definition of a function that do this job: The search happens in all the modules that import the current module, together with the current module. We can get the list of those modules and their file names by another HaRe function `clientModsAndFiles` [**?**].

```
multiply a b = foldr (+) 0 (dup a b)

-- after the transformation
multiply a b = dup_worker a b (+) 0
```

Figure 7: After the transformation

3.3.4 Fusion

In this implementation the `build` function actually never appears. We omit the inline phrase, which replaces all occurrences of a list producing function with its `build` form. The transformation simply converts all syntax phrases found in phase 3. An example is shown in Figure **??**. The most important functions used in the final phase are `replaceFold` and `convertFold`. The `replaceFold` function takes a module and abstract syntax tree, a list of list producing functions names and their workers names. It returns a new module with all the `foldr` functions rewritten.

```
replaceFold mod oldPNs newPNs
  = applyTP (full_tdTP (idTP `adhocTP` convertFold)) mod
```

The `convertFold` function converts a single occurrence of `foldr`

4 Conclusion and future directions

In this paper, we have showed that HaRe can be used effectively and effortlessly to develop a non-trivial transformation system. Although the system only uses a small part of the HaRe's library, it clearly demonstrates the use of HaRe in developing transformation system. The whole program has less then 200 lines of code suggesting the strength of the HaRe API. We hope after this work, more general transformations will be developed using HaRe.

Due to the short time avaiable for the project, I could not finish the implementation of second half of the warm fusion algorithm. A function that consumes a list should be converted to the `foldr` format. It should not be more difficult to implement this part than the fist half of the algorithm. We only need to write some additional term-rewrite functions similar to the `convertDec` function presented earlier.

When I started to work with the HaRe's group, HaRe was only a set of refactorings. Initially, the library was a flat file where some of the most frequent used functions by the refactoring programs were pulled into. The functions were written in a style that can be understood and used by internal group members with little comments. As the file quickly expanded, it is a good idea to seperate the file into different modules and write a proper documentation for the library. During the project, I and Huiqing Li [**?**] had writen the HaRe API [**?**] using Haddock and made various changes to the library to make it clearer and easier to use by external developers. In my own experience, most of the functions a developer may need to deal with any Haskell syntax phrase can be found

in the current library. But occationaly, a lower level function using Strafunski stategies is still needed.Since working with Stranfunki stategy is not intuitive nor easy, we would like to add more functions into the HaRe API so that the Strafunski library is completely hidden from the user. Also a short tutorial for HaRe beginer is very desirable.

5 Acknowledgments

I would like to thank the Nuffield Foundation for funding this project. The foundation has done a great job in supporing undergradute students doing research. The project gave me a lot of research experiences and more determination to pursue a career in academic research. I would like to thank the Funtional Programming groups at the University of Kent for their help during my project. Professor Simon Thompson was a great supervisor who not only gave me many ideas and suggestions for the project but also my first lessons in research. Huiqing Li was very generous and patient in answering all my questions about HaRe.

References

[1] `http://www.cs.kent.ac.uk/projects/refactor-fp/`

[2] John Launchbury and Tim Sheard. *Warm fusion: Deriving build-catas from recursive definitions*. In S.L. Peyton Jones, editor, *Functional programming Languages and Computer Architecture (FPCA'95)*. ACM Press, June 1995.

[3] Andy Gill. *Cheap deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1996.

[4] Andy Gill, John Launchbury and Simon L. Peyton Jones. *A short cut to deforestation*. In Arvind, editor, *Functional Programming Languages and Computer Architecture (FPCA'93)*, ACM press, 1993.

[5] Tim Sheard and Leonidas Fegaras. *A fold for all seasons*. In Arvind, editor, *Functional Programming and Computer Architecture (FPCA'93)*, Copenhagen, Denmark, 1993. ACM Press.

[6] Patricia Johann, Jeffrey R. Lewis and John Launchbury . *Warm fusion for the Masses: Detailing Virtual Data Structure Elimination in Fully Recursive Languages*. Technical Report, computer Science Department. Oregon Graduate Institute, 1998.

[7] Simon Peyton Jones and John Launchbury. *Unboxed values as first class citizen in a non-strict functional language*. In R.J.M. Hughes, editor, *Functional Programming and Computer Architecture (FPCA '91)*, volume 523 of Lecture Notes in Computer Science, Springer-Verlag, September 1991.

[8] `http://www.cse.ogi.edu/PacSoft/projects/programatica/`

[9] `http://www.cs.vu.nl/Strafunski/`

[10] The HaRe API (included in any HaRe release after August 2004)

[11] http://www.cs.kent.ac.uk/people/staff/hl/