A Combinator Library for Specifying Program Transformation

Tetsuo Yokoyama[†], Zhenjiang Hu[†,††], Masato Takeichi[†]
[†]Dept. of Mathematical Informatics
Graduate School of Information Science and Technology
The University of Tokyo
{tetsuo_yokoyama,hu,takeichi}@mist.i.u-tokyo.ac.jp
[††]PRESTO 21, Japan Science and Technology Agency

We present an embedded domain specific language for specifying program transformations. The language is implemented as a monadic combinator library in Haskell. The transformations are done at compile time using the mechanism of Template Haskell. The library provides a modular way to structure abstract and intuitive transformation strategies by higher-order matching and monadic programming.

1 Introduction

Program transformation bridges the gap between abstraction and efficiency. Glasgow Haskell Compiler (GHC) [1], enables users to write clear specification together with transformation rules, by transforming original codes into efficient ones automatically by rewriting at compile time [4] . However, the patterns in GHC are first order, and too weak to distill the parts of the structure of programs.

Higher-order patterns play an important role in program transformation [7], but they are rarely used in functional programming. In this paper, we design a monadic combinator library for program transformation in Haskell. The combinator library uses higher-order patterns as first-class values which can be passed as parameters, constructed by smaller ones in compositional way, returned as values, etc. As a result, our libraries provide more flexible binding than first-order ones, and enables more abstract and modular description of program transformation.

Since the combinator is embedded in Haskell, users do not need to learn other languages than Haskell, and enjoy all the benefits of the host language, such as type checking, module system, develop environment, etc. Thanks to the mechanism of Template Haskell [5], all the transformation specified using our libraries are expanded at compile time. Our library, together with all codes in this paper, has been tested on GHC 6.2.1.

2 A Combinator Library

As a usual construction of combinator libraries, we consider what data type our combinator convey, and design basic combinators.

2.1 Data Type of Program

We use meta-programming features to manipulate programs as values. Template Haskell provides a mechanism to handle abstract syntax trees of Haskell in Haskell itself. Enclosing brackets `[| |]` (quote) make programs abstract syntax tree whose type is `ExpQ` (= `Q Exp`), and the inverse operation is unquote described by a dollar `$`. For example, given a function to calculate the sum of a given list, `sum`, which has type[1] `[Int] -> Int`, `[| sum |]` has type `ExpQ`, whereas `$([| sum |])` has the same type as `sum`.

Unfortunately, type `ExpQ` is not enough for inner representation of programs. It needs two extra properties. Firstly, the inner representation of a program is a closure and should be represented as a tuple of an expression and an environment mapping from variables to closed expressions. Secondly, during program transformation, there is possibly more than one candidate program. Therefore, during program transformation, the data representation of program should be a list of closures.

[1] Strictly speaking, the type of function sum is $Num\ a \Rightarrow [a] \rightarrow a$ in Haskell. Here, for simplicity, we ignore type classes and polymorphism.

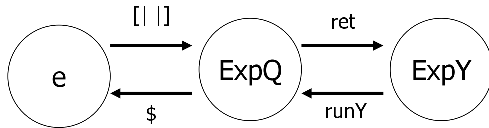

Fig. 1: Relationship of Types

<==	ExpQ -> ExpQ -> Y ()
==>	ExpQ -> ExpQ -> RuleY
<+	ExpY -> ExpY -> ExpY
>>	ExpY -> ExpY -> ExpY
casem	ExpQ -> [RuleY] -> ExpY

Fig. 2: Basic Combinators

Monad is a way to structure programming and provides such an easy treatment of program. One may construct a new monad, combining aspects of both operating lookup and update of environment and keeping track of a list of expressions. A more straightforward way is to define a combined monad consisting of smaller ones.

```
type ExpY = Y ExpQ
type Y e = StateT Subst (ListT Q) e
```

Here, `Subst` is a mapping from variables to closed expressions, and `StateT` and `ListT` are monad transformers which are defined in Haskell Hierarchical Libraries of GHC. Expanding the definition of the types `ExpY` gives

```
StateT (Subst -> ListT (Q [(ExpQ, Subst)]))
```

which means that the monad keeps track of a list of tuples of an expression (`ExpQ`) and an environment (`Subst`).

We use `ret` to lift `ExpQ` into `ExpY`. We use `runY` back to `ExpQ` from `ExpY`. The relationship of those types is summarized in Fig 1.

2.2 Basic Combinators

Our combinator has five basic constructs; match (`<==`), rule (`==>`), deterministic choice (`<+`), sequence (`>>`), and case-selection `casem`. Types of the basic combinators are summarized in Fig. 2.

The essential construct is match

```
(<==) :: ExpQ -> ExpQ -> Y ()
pat <== term
```

which yields a substitution (match) that makes patterns (`pat`) and terms (`term`) to be equal.

For example, match

```
[| \a x -> $oplus a (bign x, sum x) |]
   <== [| \a x -> if a > sum x
                  then a : bign x
                  else bign x |]
```

yields substitution

```
{ $oplus := \x (b,s) ->
            if x > s then x : b else b }
```

Note that annotation `$` means unquote. Thus, the above match can be transformed into

```
{ oplus := [| \x (b,s) ->
          if x > s then x : b else b |] }
```

Function `$oplus` is a second-order pattern and to obtain the match we used deterministic higher-order matching [7].

A transformation rule is taking an expression and returns a list of closures.

```
type RuleY = ExpQ -> ExpY
```

A transformation rule is constructed by operator `(==>)`.

```
(==>) :: ExpQ -> ExpQ -> RuleY
(pat ==> body) term = do pat <== term
                         ret body
```

Here, function `ret` implicitly applies the match kept in monad to `body`. Using them, meta version of case is to be

```
casem :: ExpQ -> [RuleY] -> ExpY
casem sel (r:rs) =
  r sel <+ casem sel rs
```

Operator (`<+`) is deterministic choice. It returns the first argument if it is not empty. Otherwise, it returns the second argument.

For simplicity, we use long arrows (`<===`) and (`===>`). They are the same as short arrows except types are

```
module Bign where
import Prelude hiding (sum)

bign []     = []
bign (x:xs) =
    if x > sum xs then x : bign xs
                  else bign xs
sum []     = 0
sum (x:xs) = x + sum xs
```

Fig. 3: Specification

```
(<===) :: ExpQ -> ExpY -> Y ()
(===>) :: ExpQ -> ExpY -> RuleY
```

Sequencing of binding new environments can be realized by combining matches by operator (>>).

```
(pat1 <== term1) >> (pat2 <== term2)
```

which can be written as sequence of match using *do notation*.

```
do pat1 <== term1
   pat2 <== term2
```

3 An Application

3.1 Programming Program Transformation

Consider, for example, function `bign` defined in Fig. 3 that returns a list whose elements are bigger than the summation of the original following list. The inefficiency of the function is caused from function `sum` in the condition of the recursive case of function `bign`. Each iteration of function `bign`, function `sum` is computed. Thus the time complexity of function `bign` is proportional to the square of the size of the input list, i.e., ($O(n^2)$). But if both the result of function `bign` and also that of function `sum` are kept track of, the time complexity would be linear. *Tupling transformation* [2] is known to enable such transformation.

It is often the case that programmers write transformations of functional program on the back of an envelope, and they only write the result program. Therefore, a way to transform a program, i.e., the invention effort for efficiency which is especially useful for refactoring and improving efficiency of the program, is generously abandoned.

On the other side, we adopt the approach of *Calculation Carrying Program* [6] in which people write a clear program with transformation rules specifying how to make it efficient. Thus, program itself is also well documented program. Using our libraries, we can annotate the transformation strategies in a code as Fig. 4. The preprocessor that we implement transforms it into Template Haskell's code. The program represents a clear specification; if we assume that function `tupling` does nothing, the value `e` would be retrieved from `[| \x -> (bign x, sum x) |]` and the first element of the body would be `bign x`, which matches the specification of the transformation.

Thanks to the abstraction of higher-order patterns, we can write the transformation rules almost as it is. The formal definition of tupling transformation is

$$\frac{\begin{array}{lcl} h\ x & = & (f\ x, g\ x) \\ f\ [\,] & = & e_f \\ f\ (a:x) & = & a \oplus (f\ x, g\ x) \\ g\ [\,] & = & e_g \\ g\ (a:x) & = & a \otimes (f\ x, g\ x) \end{array}}{\begin{array}{l} h = \textbf{let}\ a \odot (x,y) = (a \oplus (x,y), a \otimes (x,y)) \\ \quad\ \textbf{in}\ foldr\ (\odot)\ (e_f, e_g) \end{array}}$$

and can be straightforwardly programmed in Fig. 5.

3.2 Evaluation

We demonstrate evaluation of the example of the previous subsection. Transformation consists of three parts: specification (Fig. 3), application of transformation (Fig. 4), and transformation rule (Fig. 5). In Fig. 4 line 3, we import module `ProgramTransformation` which contains our combinators. In line 15,

```
tupling [| \x -> (bign x, sum x) |]
```

is called. Function `tupling` is defined in Fig. 5. In line 7, meta variables `f` and `g` are bound as

```
{ $f := bign, $g := sum }
```

```
module Main where
import Prelude hiding (sum)
import ProgramTransformation
import Tupling
import Bign

fastbign =
 $(runY (
   let laws x = casem x [
                 [| sum [] |] ==> [| 0 |],
                 [| \x xs -> sum (x:xs) |] ==> [| \x xs -> x + sum xs |],
                 [| bign [] |] ==> [| [] |],
                 [| \x xs -> bign (x:xs) |] ==>
                    [| \x xs -> if x > sum xs then x : bign xs else bign xs |] ]
   in do e <- tupling laws [| \x -> (bign x, sum x) |]
         ret [| fst . $e |]))
```

Fig. 4: Application of Transformation

Similarly, line 8,9,10 bind `oplus`, `eg`, and `otimes`. In line 8, they are substituted. Then right hand side of the match become

```
[| \a x -> bign (a:x) |]
```

which is unfolded by `laws` defined in Fig. 4 and match appeared before is obtained. Similarly, line 10 becomes match

```
[| \a x -> $oplus a (bign x, sum x) |]
   <== [| \a x -> a + sum x |]
```

and returns substitution

```
{ $oplus := \x (b,s) -> x + s }
```

All the substitution obtained before is applied in line 11 and returns the result expression. Back to Fig. 4, it is substituted into `e`, function `runY` extracts `ExpQ` from `ExpY`, and it is unquoted by `$`, which is the result program. All the evaluation is done at compile time.

Fig. 6 shows time complexity of function `bign` and `fastbign`. The experimental environment is GHC 6.2, HYLO+GHC 5.04.3 and 2 CPU (PentiumIII 1.26G, memory 1024M). While Original `bign` is asymptotically proportional to the square of the input size, optimized one `fastbign` is linear. HYLO is an automatic fusion system, which eliminates unnecessary intermediate data structures [3]. Function `bign` is not optimized by fusion, but it needs to be applied to by tupling transformation. Our programmed program transformation optimized it, where HYLO does not contribute the efficiency.

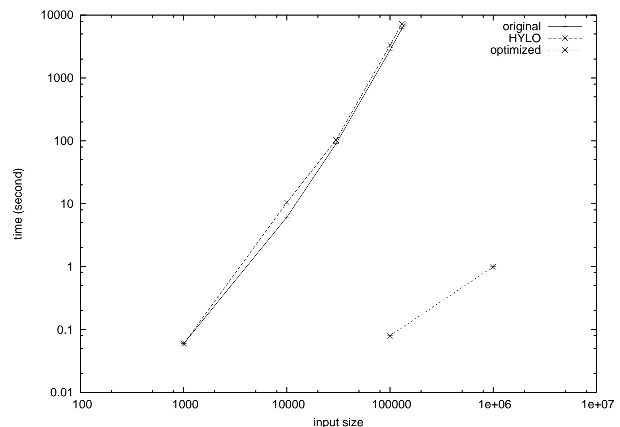

Fig. 6: Time Complexity of Function bign

```
module Tupling where
import Prelude hiding (sum)
import ProgramTransformation
import Bign

tupling laws [| \x -> ($f x,$g x) |] = do
    [| $ef |] <=== laws [| $f [] |]
    [| \a x -> $oplus   a ($f x, $g x) |] <=== laws [| \a x -> $f (a:x) |]
    [| $eg |] <=== laws [| $g [] |]
    [| \a x -> $otimes  a ($f x, $g x) |] <=== laws [| \a x -> $g (a:x) |]
    ret [| foldr (\y z -> ($oplus y z,$otimes y z)) ($ef,$eg) |]
```

Fig. 5: Transformation Rule

4 Conclusion

We present a monadic combinator library for specifying program transformations. The transformations are done at compile time using the mechanism of Template Haskell. The advantage of the library are abstraction and modularity by higher-order matching and monadic programming.

This library is in the framework of *Calculation Carrying Program* [6], in which user writes clear specification together with calculation specifying the intension of how to manipulate programs to be efficient. Our library is the realization of CCP embedded in generic purpose programming language.

References

[1] The glasgow haskell compiler. http://www.haskell.org/ghc.

[2] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

[3] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106, Le Bischenberg, France, February 1997. Chapman&Hall.

[4] Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, 2001.

[5] Tim Sheard and Simon L. Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop*, pages 1–16, Pittsburgh, Pennsylvania, May 2002.

[6] Masato Takeichi and Zhenjiang Hu. Calculation carrying programs: How to code program transformations (invited paper). In *International Sumposium on Principles of Software Evolution (ISPSE 2000)*, Kanazawa, Japan, November 2000. IEEE Press.

[7] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Deterministic higher-order patterns in program transformation. *LNCS 3018*, 2004.