Program Transformation and Analysis

Assignment 3

Benjamin Brandt Ohrt, zpn492

May 22, 2019

1 Introduction

This the third of five weekly assignment in the course Program Transformation and Analysis (PAT) at Copenhagen University. The course professor is Robert Glück. The course is held in block 4, 2019.

In this assignment the focus is on the elimination of intermediate lists and trees and The Deforestation Algorithm of Philip Wadler.

1.1 Intermediate lists

"Intermediate lists has a cost at run time. If strict evaluation is used, the program requires space proportional to n." "Even under lazy evaluation each list element requires time to be allocated, to be examined, and to be deallocated." [1, p. 231]

Philip Wadler introduces a set of rules, which can eliminate the intermediate lists. Thus improving efficiency, because operations on list cells have been eliminated. Those rules can be found in fig. 4, 'Transitionmation rules for the Deforestation Algorithm' [1, p. 238].

1.2 Language

Let the language below be one, where the Philip Wadler's rules can be used. [1, p. 233]

t	::= v	variable
	::= c t_1 .. t_k	constructor application
	::= f t_1 .. t_k	function application
	::= case t_0 of p_1 : t_1 l .. p_n : t_n l	case term
p	::= c v_1 .. v_k	pattern

1.3 The Deforestation algorithm

The Deforestation algorithm is simply to apply the rules named in section 1.1 on possible terms of 1.2 [1, p. 236]

2 Assignment

Exercise 1

Show the transformation of function composition append (append xs ys) zs into the treeless program in Fig. 3 using the Deforestation Algorithm defined in Fig. 4. Present your transformation in the style of Fig. 5. Explain the steps and discuss shortly in what sense the composition is optimized. [1]

Let append be a function described in section 3. As shown below we apply the rules of the deforestation algorithm. If a transformation already has been visited a new function will be introduced to avoid infinite loops. Such transformation will occur.

```
Rule -
     T[[append (append xs ys) zs)]]

(3)  = T[[case (append xs ys) of
       nil: zs
       cons x xs: cons x (append xs zs)]]

(6)  = T[[case (
       case xs of
         nil: ys
         cons x xs: cons x (append xs ys)
             ) of
       nil: zs
       cons x xs: cons x (append xs zs)]]

(7)  = T[[case xs of
       nil: case ys of
         nil: zs
         cons x xs: cons x (append xs zs)
       cons x xs: case cons x (append xs ys) of
         nil: zs
         cons x xs: cons x (append xs zs)]]

(4)  = case xs of
       nil: T[[case ys of
```

```
          nil: zs
          cons x xs: cons x (append xs zs)]]
        cons x xs: T[[case cons x (append xs ys) of
          nil: zs
          cons x xs: cons x (append xs zs)]]

(4)   = case xs of
        nil: case ys of
          nil: T[[zs]]
          cons x xs: T[[cons x (append xs zs)]]
        cons x xs: T[[case cons x (append xs ys) of
          nil: zs
          cons x xs: cons x (append xs zs)]]

(5)   = case xs of
        nil: case ys of
          nil: T[[zs]]
          cons x xs: T[[cons x (append xs zs)]]
        cons x xs: T[[cons x (append (append xs ys) zs)]]

(1,2)= case xs of
        nil: case ys of
          nil: zs
          cons x xs: cons x T[[(append xs zs)]]
        cons x xs: cons x T[[(append (append xs ys) zs)]]

Let h0 xs ys zs = T[[(append (append xs ys) zs)]]

    = case xs of
        nil: case ys of
          nil: zs
          cons x xs: cons x T[[(append xs zs)]]
        cons x xs: cons x h0 xs ys zs

(3) = case xs of
        nil: case ys of
          nil: zs
          cons x xs: cons x T[[case xs of
```

```
                                  nil: zs
                                  cons x xs: cons x (append xs zs)]]
        cons x xs: cons x h0 xs ys zs

Let T[[append xs zs]] = T[[case xs of
                             nil: zs, cons x xs:
                             cons x (append xs zs)]]

Let h1 xs zs = T[[(append xs zs)]]

    = case xs of
        nil: h1 xs ys
        cons x xs: cons x h0 xs ys zs
```

```
h0 xs ys zs
where
h0 xs ys zs = case xs of
                nil: h1 ys zs
                cons x xs: cons x h0 xs ys zs

h1 xs ys = case xs of
             nil: ys
             cons x xs: cons x (append xs ys)
```

Exercise 2

Apply the algorithm described in exercise 1 to append xs (append ys zs). What is the difference to the result in exercise 1?

For the transformation of append xs (append ys zs) i use the Deforestation algorithm described in [1].

```
Rule _
     T[[append xs (append ys zs)]]

(3)  = T[[case xs of
          nil: (append ys zs)
          cons x xs: cons x (append xs (append ys zs))]]

(4)  = case xs of
          nil: T[[(append ys zs)]]
          cons x xs: T[[cons x (append xs (append ys zs))]]

(3)  = case xs of
          nil: T[[case ys of
                    nil: zs
                    cons y ys: cons y (append ys zs)]]
          cons x xs: T[[cons x (append xs (append ys zs))]]

(2)  = case xs of
          nil: T[[case ys of
                    nil: zs
                    cons y ys: cons y (append ys zs)]]
          cons x xs: cons x T[[(append xs (append ys zs))]]

Let h0 xs ys zs = T[[append xs (append ys zs)]]

     = case xs of
          nil: T[[case ys of
                    nil: zs
                    cons y ys: cons y (append ys zs)]]
          cons x xs: cons x h0 xs ys zs
```

```
(4,2,1)  = case xs of
           nil: case ys of
                    nil: z
                    cons y ys: cons y T[[(append ys zs)]]
           cons x xs: cons x h0 xs ys zs

(3)  = case xs of
           nil: case ys of
                    nil: z
                    cons y ys:
                      cons y T[[
                               case ys of
                                 nil: zs
                                 cons y ys: cons y (append ys zs)]]
           cons x xs: cons x h0 xs ys zs

Infinite loop.
Let h1 ys zs = T[[append ys zs]]
Replace term of the nil branch.

    = case xs of
           nil: h1 ys zs
           cons x xs: cons x h0 xs ys zs
```

```
h0 xs ys zs
where
h0 xs ys zs = case xs of
                 nil: h1 ys zs
                 cons x: cons x h0 xs ys zs

h1 ys zs    = case ys of
                 nil: zs
                 cons y ys: cons y h1 ys zs
```

Exercise 3

Discuss the presentation structure of the article used by the author. What is the role of formalization and examples, what about the readability, and the like. In your opinion, can it be reason why the article is widely read and used?

I think the formalization and examples shows that these rules of elimination of intermediate lists and trees will work in general. Though is should be gvin that the functional programming language upholds the criteria of linear terms and tree less form [1].

For me it was a hard read. Well the introduction to intermediate lists and eliminating them was great and with somewhat good examples. But understanding the simple algorithms of flip and append took some time. Only because of knowledge of how they should work, helped understanding the general expressions of Cons and Branch. Another obstacle was my lack of understanding some of the transformation rules. For me it could have been nice with simple examples for each rule. That said the article both had explanation and full examples so i guess its my talents which is insufficient.

Again the reason why this article is widely read and used, is probably the use of a general language, which my guess is, it can fit in all functional languages.

3 Append

```
Append          : list a -> list a -> list a
Append xs ys =
  case xs of
    Nil         : ys
    Cons x xs   : Cons x (append xs ys)
```

References

[1] Wadler P., *Deforestation: transforming programs to eliminate trees. Theoretical Computer Science, 73(2): 231-248, 1990.* https://doi.org/10.1016/0304-3975(90)90147-A