5Q–01 Recursion Removal and Introduction using Destructive Associativity

Kazuhiko Kakehi* Robert Glück** Yoshihiko Futamura**

* Graduate School of Science and Engineering, Waseda University
** School of Science and Engineering, Waseda University

1 Introduction

Recursive programs are often easy to write and reason about, while iterative representation is usually more efficient to execute. Transformation between recursive and iterative variants of a function is hence quite important in order to enjoy the benefits of both programming styles. This topic has been energetically researched for many years by several approaches, [1, 2, 3] for example, but translation from iteration to recursion or recursion removal from **append**-like functions seem rare.

To tackle with recursion removal toward functions constructing structures, pointer operations are focused in [4]. Following this, We found the way to give more flexibility, and it is possible to translate not only from recursion to iteration but for the other direction.

This presentation will first explain the basic idea 'prestructures' to enable these transformations in Section 2, and transformations themselves using examples in Section 3. We give a short note on an implementation idea of prestructures in Section 4, finally Section 5 concludes.

2 Basic Idea

Recursive data-types are constructed like:

$$\begin{array}{rcl} \mathit{IntList} & :: & \texttt{Nil} \\ & | & \texttt{Cons}[\mathit{Int}, \mathit{IntList}]. \end{array}$$

In this construction, IntList grows only from right to left, and decomposition from right is not allowed. Now we take an example **app**. Since it is defined following the data-type, an iterative solution for it is not observable.

$$\begin{array}{l} \texttt{app}(\texttt{x},\texttt{y}) \stackrel{\text{def}}{=} \\ \quad \texttt{case x of } \begin{array}{ll} \texttt{Nil} & \rightarrow \texttt{y} \\ \texttt{Cons}[\texttt{x1},\texttt{xs}] & \rightarrow \texttt{Cons}[\texttt{x1},\texttt{app}(\texttt{xs},\texttt{y})] \end{array} \end{array}$$

Our basic approach to relax this is to allow half-way construction of *IntList* using a destructive operation $\hookleftarrow_i$, an infix operator to put the right argument in the i-th position of the left constructor, and a nullary, dummy constructor _ (or `Dummy` explicitly) to fill the blank.

$$\begin{array}{rcl} \mathit{IntList} & :: & \texttt{Nil} \\ & | & [\mathit{Int}, \texttt{Dummy}]_{\texttt{Cons}} \hookleftarrow_2 \mathit{IntList} \end{array}$$

We call these unfinished prefix parts *prestructures*.

Since new constructions are done on the left of recursive calls, the result is created from left to right as the pattern variable is traversed. To follow this, these constructions have to be accumulated on the right of its accumulator in the tail-recursive definition. We use $\hookleftarrow_i$ also for the destructive operation to put the right argument in the i-th position of *the tail* of the left prestructure. Our convention enables to express this:

$$\begin{array}{l} \texttt{app}(\texttt{x},\texttt{y}) = \texttt{appi}(\texttt{x},\texttt{y},\texttt{Dummy}) \\ \texttt{appi}(\texttt{x},\texttt{y},\texttt{accl}) \stackrel{\text{def}}{=} \\ \quad \texttt{case x of} \\ \quad\quad \begin{array}{ll} \texttt{Nil} & \rightarrow \texttt{accl} \hookleftarrow_2 \texttt{y} \\ \texttt{Cons}[\texttt{x1},\texttt{xs}] & \rightarrow \texttt{appi}(\texttt{xs},\texttt{y},\texttt{accl} \hookleftarrow_2 [\texttt{x1},\_]_{\texttt{Cons}}) \end{array} \end{array}$$

In the literature associativity of **append** is often used for recursion removal or adding constructions on the right; it has sometimes disadvantages of worsening computation. Though the operation '$\hookleftarrow$' is destructive, this effects doesn't spread to other parts of programs under the assumption of no side-effects. We can therefore enjoy associativity of list construction without those disadvantages!

3 Transformation

In this section we explain the transformation using examples. Following Section 2, we assume functions without side-effects. We use $[\texttt{x1}]$ instead of $[\texttt{x1},\_]_{\texttt{Cons}}$ as their shorthands.

3.1 Recursion Removal—`mir`

Given a list of integers of *IntList*, it returns a *IntList*:

$$\begin{array}{l} \texttt{mir}(\texttt{x},\texttt{y}) \stackrel{\text{def}}{=} \texttt{case x of} \\ \quad \begin{array}{ll} \texttt{Nil} & \rightarrow \texttt{y} \\ \texttt{Cons}[\texttt{x1},\texttt{xs}] & \rightarrow \texttt{Cons}[\texttt{x1},\texttt{mir}(\texttt{xs},\texttt{Cons}[\texttt{x1},\texttt{y}])] \end{array} \end{array}$$

We make it a fully tail-recursive program.

First the definition is changed to use prestructures instead:

```
mir(x,y) =def case x of
    Nil          → y
    Cons[x1,xs] → [x1] ↩2 mir(xs,[x1] ↩2 y).
```

We prepare a new accumulator `accl` to collect constructions outside; when there is an outside construction on the *left* of recursive calls, it is accumulated inside on the *right* of the accumulator. We finally obtain the solution:

```
mir(x,y) = miri(x,y,Dummy)
miri(x,y,accl) =def case x of
    Nil          → accl ↩2 y
    Cons[x1,xs] → miri(xs,[x1] ↩2 y,
                       accl ↩2 [x1])
```

3.2 Recursion Introduction—`flat`

Given a tree structure *IntTree* as an input, `flat` returns a *IntList*:

```
IntTree :: Leaf[Int]
         | Node[IntTree,IntTree]
```

```
flat(x,y) =def
  case x of Leaf[i]      → Cons[i,y]
            Node[xl,xr] → flat(xl,flat(xr,y))
```

This is an, so to say, iterative solution using accumulation. We want to derive its recursive variant of `flat` without accumulation.

We need care for treating plural function calls appearing its definition. Among them, one is regarded as the *main* recursive call and the rest are dealt with as *forked* recursive calls. The out-most one is the main call in the iterative form.

First, the definition is changed in the manner of prestructures.

```
flat(x,y) =def
  case x of Leaf[i]      → [i] ↩2 y
            Node[xl,xr] → flat(xl,flat(xr,y))
```

Observing `y` is accumulating, `y` is put out with leaving `acc` initialized as `Dummy`, and we obtain definition of `flat1`:

```
flat(x,y)      = flat1(x,Dummy) ↩2 y
flat1(x,acc) =def case x of
  Leaf[i]      → [i] ↩2 acc
  Node[xl,xr] → flat1(xl,flat(xr,acc))
```

We again apply this rule to the inner forked call.

```
flat1(x,acc) =def case x of
  Leaf[i]      → [i] ↩2 acc
  Node[xl,xr] → flat1(xl,flat1(xr,Dummy) ↩2 acc)
```

Now that we see that the inner forked call is accumulated on the left of `acc`, it can be put out on the right of the main recursive call when eliminating `acc` from the definition of `flat1`:

```
flatr(x) =def case x of
    Leaf[i]      → [i]
    Node[xl,xr] → flatr(xl) ↩2 flat1(xr,Dummy)
```

Finally the remaining calls `flat1(x,Dummy)` are replaced by `flatr(x)`, and we obtain the final solution:

```
flat(x,y) = flatr(x) ↩2 y
flatr(x) =def case x of
    Leaf[i]      → [i]
    Node[xl,xr] → flatr(xl) ↩2 flatr(xr)
```

4 Implementation

Here a problem remains: how is this implemented in programming languages? As we have already mentioned, a destructive operation $\hookleftarrow_i$ puts the pointer of some (pre-)structure on the i-th position of the tail of a prestructure. This implies that it is enough to keep pointers top and tail elements of prestructures, or using circular lists, where taking and putting elements on the left and putting elements on the right are allowed and taking elements from the right cannot be done. There is no need to use double-linked lists to implement our idea.

5 Conclusion

We have presented simple ways to deal with removal and introduction of recursion using the idea of prestructures. We have seen that relaxing construction makes treatment of recursion quite easy.

Now that basic approaches are seen, formal transformaiton rules including treatments for `let`-expressions and the way for its implementation are left as future works.

References

[1] R. Bird. Notes on recursion elimination. *Comm. ACM*, 20(6):434–439, June 1977.

[2] P. G. Harrison and H. Khoshnevisan. A new approach to recursion removal. *Theoret. Comput. Sci.*, 93(1):91–113, Feb. 3, 1992.

[3] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In J. Lawall, editor, *PEPM'00*, pages 73–82, Jan. 2000.

[4] 二村, 大谷. 線形再帰プログラムからの再帰除去とその実際的効果. コンピュータソフトウェア, 15(3):38–49, May 1998.