M. Höff, R. Vater, A. Maletti, A. Kühnemann, and J. Voigtländer

Tree transducer based program transformations for Haskell$^+$

(progress report until March 2001)

$MAC \circ TOP \subseteq MAC$
$TOP \circ MAC \subseteq MAC$
$ATT \subseteq MAC$
$ATT_{su} \circ ATT \subseteq ATT$
$ATT \circ TOP \subseteq ATT$
$MAC_{wsu} \subseteq ATT$
$MAC_{nc} \circ MAC_{wsu} \subseteq MAC$
Deforestation for MACs
$MAC \subseteq TOP \circ YIELD$

Contents

1 Introduction

This document is intended as a first introduction to the programs of the Haskell$^+$ group on implementing program transformations for functional programs (currently constituted by Armin Kühnemann, Rainer Vater, Janis Voigtländer and Andreas Maletti, previous work done by Christian Lescher and Matthias Höff). It is not self–contained as the program transformations are not described formally here, but references to the literature are given where appropriate. The system is based on the Haskell$^+$ language for extending Haskell by macro attribute grammars (which are an extension of attribute grammars), described and implemented in [Les99]. This functional programming language is used for specifying different kinds of tree transducers as known from the literature (see [EV85, FV98, Küh98]). Several constructions for results from tree transducer theory are implemented in Haskell as transformations on Haskell$^+$ programs. There is as yet just a rudimentary user interface, so this document will focus on implementation issues by describing the involved Haskell modules in short and giving hints for further implementation work. For illustrating the implemented program transformations, we will give a simple example each, as well as a complex example at the end, which makes combined use of several transformations.
In Section 2, we describe the modules that build the environment for the program transformations, such as the basic datatype definitions and the user interface. In Section 3, we describe the Haskell modules that implement the program transformations. Section 4 presents an example where several transformation techniques are used.

2 Modules for "Infrastructure"

2.1 makefile

This is not a Haskell module, but a control file for the `make` command. By invocating "`make menu`" in the `/var/local/share/mag/hpg/` directory, dependencies are checked and if necessary, modules are recompiled and the executable file "`menu`" is built. Newly implemented transformations should be introduced into the `makefile` by including them in the `COMP` list.

Name:	`makefile`
Authors:	Janis Voigtländer
Purpose:	automated dependency check and compilation

2.2 SyntaxTr

This module contains the definition of algebraic datatypes that are used for the abstract syntax of Haskell$^+$ programs. All the program transformations work on these datatypes.

Name:	`syntaxtr.hs`
Authors:	Christian Lescher
Purpose:	definition of abstract syntax for Haskell$^+$

2.3 List_mag

This module implements the function `t_Prog` that turns the abstract syntax tree of a Haskell$^+$ program into a textual representation of this program.

Name:	`List_mag.hs`
Authors:	Rainer Vater
Purpose:	prettyprint abstract syntax tree to program text
Imports:	`SyntaxTr, Is_Type`

2.4 HPG

This module serves as an interface for collecting all the programs of the Haskell$^+$ group (including the Haskell$^+$ compiler, which is not described in this document). Every newly implemented program transformation should be made available through this module. The module also contains the function `comp_sumac_wsumac` that realizes the transformation **su-MAC** $\circ$ **wsu-MAC** $\to$ **MAC** by combining some of the other transformations (note that both macro tree transducers must be given with root symbols).

Name:	`HPG.hs`
Authors:	Matthias Höff and others
Purpose:	integrate modules of Haskell$^+$ group
Imports:	`SyntaxTr, Is_Type, List_mag,` all the transformation modules

2.5 Main

This module implements the user interface to the programs of the Haskell$^+$ group. It presents a textual menu as below with the possible choices which can be selected by a bit–encoded input number (not only the shown combinations are implemented):

```
                                 | 35|227|231| 55|107| 59|115| 47|255|547|551|
                                 |---|---|---|---|---|---|---|---|---|---|---|
  1 : hptree                     | x | x | x | x | x | x | x | x | x | x | x |
  2 : is_typ                     | x | x | x | x | x | x | x | x | x | x | x |
  4 : ATT                 -> MAC |   |   | x |   |   |   |   |   |   |   |   |
  8 : MAC    o TOP        -> MAC |   |   |   |   | x |   |   |   |   |   |   |
 12 : Deforestation              |   |   |   |   |   |   |   | x |   |   |   |
 16 : TOP    o MAC        -> MAC |   |   |   |   |   |   | x |   |   |   |   |
 20 : wsu-MAC             -> ATT |   |   |   | x |   |   |   |   |   |   |   |
 24 : ATT    o TOP        -> ATT |   |   |   |   |   | x |   |   |   |   |   |
 24 : su-ATT o ATT        -> ATT |   |   |   |   |   | x |   |   |   |   |   |
 28 : su-MAC o wsu-MAC -> MAC    |   |   |   |   |   |   |   |   | x |   |   |
 32 : showPrg                    | x | x | x | x | x | x | x | x | x | x | x |
 64 : Pruning                    |   | x | x |   | x |   | x |   | x |   |   |
128 : Copy Elimination           |   | x | x |   |   |   |   |   | x |   |   |
256 : hptrans                    |   |   |   |   |   |   |   |   |   |   |   |
512 : MAC   -> TOP    o YIELD    |   |   |   |   |   |   |   |   |   | x |   |
516 : nc-MAC o wsu-MAC -> MAC    |   |   |   |   |   |   |   |   |   |   | x |
```

Name:	`Main.hs`
Authors:	Matthias Höff, additions by Janis Voigtländer
Purpose:	menu interface to programs of Haskell$^+$ group
Imports:	`HPG, TrAuxFns`

3 Modules for Transformations

3.1 TrAuxFns

This module contains common functions used in the transformations. Most functions work on abstract syntax trees of Haskell+ programs, extracting information such as types, attribute lists, rules, root and nullary symbols, etc. There are also functions for building reappearing constructs (such as `makeVariables` and `makePattern`) and functions to work with several kinds of list structures from `SyntaxTr` as well as for conversion between these and Haskell lists. Of special importance are the `getRules, norm` and associated functions that provide tree transducer rules in a unique form (e.g., abstracted from chosen variable names).

The module also contains a new datatype `MacExp` for representing right–hand sides of macro tree transducers, together with operations that work on it (such as `toMacRules`). This datatype was introduced for the `Deforest` and `MAC_ATT` transformations and seems to be easier to use than the original `RuleRhs` and `RhsExp` representation (see `rhs2re` and `re2rhs`) as it abstracts away from some redundancies that are not of importance for macro tree transducers[1]. There is also a collection of general auxiliary functions that might be useful in further implementations.

Name:	`TrAuxFns.hs`
Authors:	Matthias Höff and Janis Voigtländer
Purpose:	auxiliary functions for transformations
Imports:	`SyntaxTr`

3.2 Is_Type

This module is used to check several properties of tree transducers. There are functions that detect which `MagBlock`s actually contain tree transducers, as well as to classify tree transducers and check for certain properties. The classification is determined by one of the following attributes: `"Mat","Mac","Att","Top"`, for macro–attributed, macro, attributed and top–down tree transducers, respectively. The properties implemented so far, are the following:

`"Su"`	weakly single–use & noncopying, see [Küh00]
`"Ylin"`	linear in context parameters (noncopying)
`"Xlin"`	linear in recursion argument
`"Xlin_undef"`	for attributed tree transducers, `Xlin` is undefined
`"Ynd"`	non–deleting in context parameters
`"Xnd"`	non–deleting in recursion argument
`"Prod"`	producing, see [Höf99]
`"Tl"`	treeless
`"Wp"`	well–presented for macro tree transducers, see [FV98]
`"Swp"`	super well–presented, also see [FV98]
`"Wsu"`	weakly single–use (single–use displayed for Atts), see [Küh00]

[1] e.g., the `RuleRhs` representation of the right–hand side `rev x1 (A y1)` is parsed as:
`RuleRhsLowerIdpp "rev" (RhsExpListOList (RhsExpListPair (RhsExpListSingle (RhsExpLowerId "x1")) (RhsExpCompound (CompoundTuple (RuleRhsCommaListSingle (RuleRhsUpperIdpp "A" (RhsExpListOList (RhsExpListSingle (RhsExpLowerId "y1")))))))));`
as `MacExp` it simply reads: `Func "rev" [Var "x1",Constr "A" [Var "y1"]]`

Name: Is_Type.hs
Authors: Rainer Vater, "Wp" and "Swp" added by Janis Voigtländer, "Wsu" added by Andreas Maletti
Purpose: check for types and properties of tree transducers
Imports: SyntaxTr
Example:

```
begindata Data
  data Root = Root List
  data List = A List | B List | E
enddata

beginmag Ex [Att,Mac,Mat,Prod,Su,Swp,Tl,Top,Wp,Xlin,Xnd,Ylin,Ynd]
  input Data
  syn ex
  ex (A x) = B (ex x)
  ex (B x) = A (ex x)
  ex E = E
endmag

beginmag Rev1 [Mac,Mat,Su,Swp,Wp,Wsu,Xlin,Xnd,Ylin,Ynd]
  input Data
  root Root
  syn s_rev1 ::  Root -> List
  syn rev ::  List -> List -> List
  s_rev1 (Root x) = rev x E
  rev (A x) y = rev x (A y)
  rev (B x) y = rev x (B y)
  rev E y = y
endmag

beginmag Rev2 [Att,Mat,Su,Xlin_undef,Xnd,Ylin,Ynd]
  input Data
  root Root
  syn s_rev2 ::  Root -> List
  syn a ::  List -> List
  inh b ::  List -> List
  t@(Root t1) in
    s_rev2 t = a t1
    b t1 = E
  t@(A t1) in
    a t = a t1
    b t1 = A (b t)
  t@(B t1) in
    a t = a t1
    b t1 = B (b t)
  t@E in
    a t = b t
endmag
```

3.3 TOPoMAC

This module implements the function `compTopMac` which composes a top–down and a macro tree transducer to a new macro tree transducer by following a direct construction (not mentioned in the literature so far, see [Küh99]), in contrast to the indirect construction via $YIELD$–functions in [EV85]. For this transformation to work correctly, the second tree transducer must not be given with root symbol. That´s why for the example a variant `Rev` of Rev1 (from page 4) without the function s_rev1 was used.

Name:	TOPoMAC.hs
Authors:	Matthias Höff
Purpose:	transformation **TOP** ∘ **MAC** → **MAC**
Imports:	SyntaxTr, TrAuxFns
Example:	see below

```
{- replace compositions by one function call:
  rev (ex x) y1 = ex_rev x y1 -}
beginmag Ex_o_Rev [Mac]
  input Data
  syn ex_rev ::  List -> List -> List
  ex_rev (A x1) y1 = (ex_rev x1 (B y1))
  ex_rev (B x1) y1 = (ex_rev x1 (A y1))
  ex_rev E y1 = y1
endmag
```

3.4 MACoTOP

This module implements the function compMacTop which composes a macro and a top–down tree transducer to a new macro tree transducer by following the classical transformation from the proof of $MAC \circ TOP \subseteq MAC$ as given, e.g., in [EV85]. Note that as well as in the transformation **TOP** ∘ **MAC** → **MAC**, the second tree transducer must not be given with root symbol.

Name:	MACoTOP.hs
Authors:	Matthias Höff
Purpose:	transformation **MAC** ∘ **TOP** → **MAC**
Imports:	SyntaxTr, TrAuxFns
Example:	see below

```
{- replace compositions by one function call:
  ex (rev x y1) = rev_ex x (ex y1) -}
beginmag Rev_o_Ex [Mac]
  input Data
  syn rev_ex ::  List -> List -> List
  rev_ex (A x1) y1 = rev_ex x1 (B (y1 ))
  rev_ex (B x1) y1 = rev_ex x1 (A (y1 ))
  rev_ex E y1 = y1
endmag
```

3.5 MAC_ATT

This module implements the function wsuMacToAtt which transforms one or all weakly single–use macro tree transducers (replacing them) to attributed tree transducers by following a simplified version (without check for "important parameter positions") of the construction described in [Höf98]. If the original weakly single–use macro tree transducer was additionally noncopying (property "Ylin"), then the resulting attributed tree transducer will be single–use. Note that this transformation only works on macro tree transducers specified with root symbol.

Name:	`MAC_ATT.hs`
Authors:	Janis Voigtländer
Purpose:	transformation **su-MAC** $\rightarrow$ **ATT**
Imports:	`SyntaxTr, TrAuxFns`
Example:	

```
beginmag Rev1 [Att,Su]
  input Data
  root Root
  inh y1_rev
  syn s_rev1
  syn rev
  x@(Root x1) in
    y1_rev x1 = E
    s_rev1 x = rev x1
  x@(A x1) in
    y1_rev x1 = A (y1_rev x)
    rev x = rev x1
  x@(B x1) in
    y1_rev x1 = B (y1_rev x)
    rev x = rev x1
  x@E in
    rev x = y1_rev x
endmag
```

3.6 Deforest

This module implements the function `deforest` which applies the technique of deforestation to two macro tree transducers and a composition expression by using the special version of deforestation for macro tree transducers as described in [Küh99, Höf99] (in contrast to the original deforestation algorithm for treeless programs from [Wad90]). The resulting function definitions are not given in a `MagBlock`, because in general the result of deforestation is not a macro tree transducer. (Instead, the new datatype `MacExp` is used. An interesting detail of implementation is the use of a state monad for accumulating the function definitions.)

Name:	`Deforest.hs`
Authors:	Janis Voigtländer
Purpose:	**deforestation** for macro tree transducers
Imports:	`SyntaxTr, TrAuxFns`
Example:	

```
beginmag App [Mac,Mat,Su,Swp,Tl,Wp,Wsu,Xlin,Xnd,Ylin,Ynd]
  input Data
  syn app ::  List -> List -> List
  app (A x) y = A (app x y)
  app (B x) y = B (app x y)
  app E y = y
endmag

{- replace (app (app z0 z1) z2) with (app_app z0 z1 z2) , where:  -}
app_ (E) y1 = y1
app_ (B x1) y1 = (B (app_ x1 y1))
app_ (A x1) y1 = (A (app_ x1 y1))

app_app (E) y1 y2 = (app_ y1 y2)
app_app (B x1) y1 y2 = (B (app_app x1 y1 y2))
app_app (A x1) y1 y2 = (A (app_app x1 y1 y2))
```

3.7 ATToATT

This module implements the function `compAttAtt` which composes two attributed tree transducers by following the construction described in [Küh98] (based on composition results for language morphisms and attribute coupled grammars, cf. [Gan83, GG84, Gie88]). This construction does not work with arbitrary attributed tree transducers. It is restricted to the cases where either the first attributed tree transducer has the single–use property or the second one is a top–down tree transducer. Due to the fact that a top–down tree transducer may be specified without a root symbol, there are several tricky cases in the implementation. Especially, if the first attributed tree transducer is given as a top–down tree transducer without root symbol, an artificial root symbol `"No_Root"` is created and a new datatype with same name is added to the corresponding `DataBlock`. The construction will **not** work, if there is more than one initial synthesized attribute at a root symbol.

Name: `ATToATT.hs`
Authors: Matthias Höff, additions by Janis Voigtländer
Purpose: transformation **su-ATT** $\circ$ **ATT** $\rightarrow$ **ATT**,
resp., **ATT** $\circ$ **TOP** $\rightarrow$ **ATT**
Imports: `SyntaxTr, TrAuxFns`
Example:

```
{- replace compositions by one attribute call:
  s_rev2 (Root (s_rev2 (Root t))) = s_rev2_s_rev2 (Root t) -}
beginmag Rev2_o_Rev2 [Att]
  input Data
  root Root
  syn s_rev2_s_rev2 ::  Root -> List
  syn a_a ::  List -> List
  syn b_b ::  List -> List
  inh b_a ::  List -> List
  inh a_b ::  List -> List
  t@(Root x1) in
    s_rev2_s_rev2 t = a_a x1
    b_a x1 = b_b x1
    a_b x1 = E
  t@(A x1) in
    a_a t = a_a x1
    b_a x1 = b_a t
    a_b x1 = a_b t
    b_b t = A (b_b x1)
  t@(B x1) in
    a_a t = a_a x1
    b_a x1 = b_a t
    a_b x1 = a_b t
    b_b t = B (b_b x1)
  t@E in
    a_a t = b_a t
    b_b t = a_b t
endmag
```

3.8 ATT_MAC

This module implements the function `transAttMac` which transforms one or all attributed tree transducers (replacing them) to macro tree transducers by following the construction from the proof of $ATT \subseteq MAC$ as given, e.g., in [FV98].

Name:	`ATT_MAC.hs`
Authors:	Matthias Höff, corrections by Janis Voigtländer
Purpose:	transformation **ATT** → **MAC**
Imports:	`SyntaxTr, TrAuxFns`
Example:	

```
beginmag Rev2_o_Rev2 [Mac]
  input Data
  root Root
  syn b_b ::  List -> List -> List -> List
  syn a_a ::  List -> List -> List -> List
  syn s_rev2_s_rev2 ::  Root -> List
  s_rev2_s_rev2 (Root x1) = a_a x1 (b_b x1 E E) E
  a_a (A x1) y1 y2 = a_a x1 y1 y2
  a_a E y1 y2 = y1
  a_a (B x1) y1 y2 = a_a x1 y1 y2
  b_b E y1 y2 = y2
  b_b (A x1) y1 y2 = A (b_b x1 y1 y2)
  b_b (B x1) y1 y2 = B (b_b x1 y1 y2)
endmag
```

3.9 Prune

This module implements the "pruning" method for removing redundant context parameters of macro tree transducer states that might be introduced by some of the transformations. A fixed point construction is used to determine which context parameters will never (for no possible input tree) influence the output generated by a state. These context parameter positions are then removed from the definitions and all the right-hand sides are changed accordingly.

Name:	`Prune.hs`
Authors:	Janis Voigtländer
Purpose:	remove redundant context parameters from **MAC**s
Imports:	`SyntaxTr, TrAuxFns`
Example:	

```
{- replace function calls with pruned versions:
  b_b t y1 y2 = b_b_ t y2
  a_a t y1 y2 = a_a_ t y1 -}
beginmag Rev2_o_Rev2 [Mac]
  input Data
  root Root
  syn s_rev2_s_rev2
  syn a_a_ ::  List -> List -> List
  syn b_b_ ::  List -> List -> List
  s_rev2_s_rev2 (Root x1) = a_a_ x1 (b_b_ x1 E)
  a_a_ (A x1) y1 = a_a_ x1 y1
  a_a_ E y1 = y1
  a_a_ (B x1) y1 = a_a_ x1 y1
  b_b_ E y2 = y2
  b_b_ (A x1) y2 = A (b_b_ x1 y2)
  b_b_ (B x1) y2 = B (b_b_ x1 y2)
endmag
```

3.10 CopyElim

This module implements the method for eliminating copy–rules from macro tree transducers. A "copy–rule" occurs when some state will always, independently of its input tree, project on one of its context parameters. For simplicity reasons, the method is only implemented for states of rank two, i.e., states having exactly one context parameter[2]. The criterion for being a copy–rule is then simply that all right–hand sides of a state contain a) no constructors and b) recursive calls only to the state itself.
States that were detected to be occurrences of copy–rules are eliminated from the macro tree transducer and the right–hand sides of all rules are simplified accordingly.

Name:	`CopyElim.hs`
Authors:	Janis Voigtländer
Purpose:	eliminate copy–rules from **MAC**s
Imports:	`SyntaxTr, TrAuxFns`
Example:	see below

```
{- eliminate copy-rule function calls:
  a_a_ t y = y -}
beginmag Rev2_o_Rev2 [Mac]
  input Data
  root Root
  syn s_rev2_s_rev2
  syn b_b_ ::  List -> List -> List
  s_rev2_s_rev2 (Root x1) = b_b_ x1 E
  b_b_ E y1 = y1
  b_b_ (A x1) y1 = A (b_b_ x1 y1)
  b_b_ (B x1) y1 = B (b_b_ x1 y1)
endmag
```

3.11 DMAC_TOP

This module implements the decomposition of a macro tree transducer into a top-down tree transducer and a yield-function. The user is presented the choice, whether the yield-function is to be realized using either a macro tree transducer or an attributed tree transducer. The presence of a root symbol in the inputted macro tree transducer is preserved in the outputted top-down tree transducer. The implementation follows the construction given in [Küh98].

In contrast to the construction given in [Küh98], the implementation takes a slightly different approach. It does not alter the arity of the output symbols of the original macro tree transducer.

[2] this simplification is safe if the "pruning" method is applied before copy–rule elimination, because a state having more than one context parameter positions after "pruning" means that both context parameters influence the output for some input trees, so there can not be an occurrence of a copy–rule

Name: DMAC_TOP.lhs
Authors: Andreas Maletti
Purpose: decompose **MAC** into **TOP** and **YIELD**
Imports: SyntaxTr, TrAuxFns, Is_Type
Example:

```
{- replace function calls to
  rev x y1 by sub1 ( unary_rev x ) y1
  s_rev1 x by sub0 ( unary_s_rev1 x ) -}
begindata Data
  data Decomp_Root_List = Root_List Decomp_List List
  data Decomp_List = SUB1 Decomp_List Decomp_List | PI1 |
    A' Decomp_List | B' Decomp_List | E'
  data Root = Root List
  data List = A List | B List | E
enddata

beginmag Decomp_Rev1 [Att,Mac,Mat,Prod,Su,Swp,Tl,Top,Wp,...]
  input Data
  root Root
  syn unary_s_rev1 ::  Root -> Decomp_List
  syn unary_rev ::  List -> Decomp_List
  unary_s_rev1 (Root x1) = SUB1 (unary_rev x1) E'
  unary_rev (A x1) = SUB1 (unary_rev x1) (A' PI1)
  unary_rev (B x1) = SUB1 (unary_rev x1) (B' PI1)
  unary_rev E = PI1
endmag

beginmag Yield_Rev1 [Mac,Mat,Xlin,Xnd,Ylin]
  input Data
  syn sub0 ::  Decomp_List -> List
  syn sub1 ::  Decomp_List -> List -> List
  sub0 (SUB1 x1 x2) = sub1 x1 (sub0 x2)
  sub0 PI1 = E
  sub0 (A' x1) = A (sub0 x1)
  sub0 (B' x1) = B (sub0 x1)
  sub0 E' = E
  sub1 (SUB1 x1 x2) y1 = sub1 x1 (sub1 x2 y1)
  sub1 PI1 y1 = y1
  sub1 (A' x1) y1 = A (sub1 x1 y1)
  sub1 (B' x1) y1 = B (sub1 x1 y1)
  sub1 E' y1 = E
endmag
```

3.12 MACoMAC

This module implements the composition of a noncopying macro tree transducer with a weakly single–use macro tree transducer to gain another macro tree transducer according to the construction given in [Voi01].

Name:	`MACoMAC.lhs`
Authors:	Andreas Maletti
Purpose:	compose **nc-MAC** with **wsu-MAC** to get **MAC**
Imports:	`SyntaxTr, TrAuxFns`
Example:	see below

```
{- replacement of function calls:
  rev ( rev x1 y1 ) z1 by (comp_rev_o_rev x1 (rev y1
     (comp_1rev_o_1rev x1 El z1)) z1)
-}

beginmag Rev_o_Rev [Mac,Mat,Su,Wp,Wsu,Xlin,Xnd,Ylin]
  input Data
  syn comp_1rev_o_1rev ::  List -> List -> List -> List
  syn comp_rev_o_rev ::  List -> List -> List -> List
  comp_1rev_o_1rev (A x1) y1 y2 = A (comp_1rev_o_1rev x1 E y2)
  comp_1rev_o_1rev (B x1) y1 y2 = B (comp_1rev_o_1rev x1 E y2)
  comp_1rev_o_1rev E y1 y2 = y2
  comp_rev_o_rev (A x1) y1 y2 = comp_rev_o_rev x1 y1 y2
  comp_rev_o_rev (B x1) y1 y2 = comp_rev_o_rev x1 y1 y2
  comp_rev_o_rev E y1 y2 = y1
endmag
```

4 A complex example

Now we want to give an example for making combined use of the implemented program transformations. Consider the problem of changing in a list all elements of same parity (even or odd, counting from zero) such that the last element of the list is not in the set of changed positions. It is not immediate, how this could be done with, e.g., a macro tree transducer. However, using a modular style of programming, we might figure that the problem is easily solved by first reversing the list, then changing elements at positions one, three, five, ... and then reversing the resulting list again. In the following we show, how from this modular characterization a macro tree transducer can be derived.

We start with the following program, defining two tree transducers for reversing a list and for changing the odd elements (with positions one, three, five, ...) of a list to `A`:

```
begindata Data
  data Root = Root List
  data List = A List | B List | C List | E
enddata

beginmag Rev
  input Data
  root Root
  syn s_rev
  syn rev  :: List -> List -> List
  s_rev (Root t) = rev t E
  rev (A x) y = rev x (A y)
  rev (B x) y = rev x (B y)
  rev E     y = y
endmag

beginmag Even
  input Data
  root Root
  syn s_even
  syn even
  syn odd
  s_even (Root t) = even t
  even (A x) = A (odd x)
  even (B x) = B (odd x)
  even E     = E
  odd (A x)  = A (even x)
  odd (B x)  = A (even x)
  odd E      = E
endmag
```

□

Our problem can now be solved for some list `t` of type `List` by the expression:

`s_rev (Root (s_even (Root (s_rev (Root t))))).`

Since `Rev` is a single–use macro tree transducer and `Even` is a top–down tree transducer, both can be transformed to attributed tree transducers, yielding the following (partial, since data definitions are left out) program:

```
beginmag Rev [Att,Su]
  input Data
  root Root
  inh y1_rev
  syn s_rev
  syn rev
```

```
  x@(Root x1) in
    y1_rev x1   =  E
    s_rev x   =  rev x1
  x@(A x1) in
    y1_rev x1   =  A (y1_rev x)
    rev x   =  rev x1
  x@(B x1) in
    y1_rev x1   =  B (y1_rev x)
    rev x   =  rev x1
  x@E in
    rev x   =  y1_rev x
endmag

beginmag Even [Att,Su]
  input Data
  root Root
  syn s_even
  syn even
  syn odd
  x@(Root x1) in
    s_even x   =  even x1
  x@(A x1) in
    even x   =  A (odd x1)
    odd x   =  A (even x1)
  x@(B x1) in
    even x   =  B (odd x1)
    odd x   =  A (even x1)
  x@E in
    even x   =  E
    odd x   =  E
endmag
```

□

Applying the transformation for composing two attributed tree transducers, where the first one is single–use, to Rev and Even gives the new attributed tree transducer Rev_o_Even:

```
{- replace compositions by one attribute call:
  s_even (Root (s_rev (Root t))) = s_rev_s_even (Root t) -}
beginmag Rev_o_Even [Att]
  input Data
  root Root
  syn s_rev_s_even
  syn rev_even
  syn rev_odd
  inh y1_rev_even
  inh y1_rev_odd
  t@(Root x1) in
    s_rev_s_even t   =  rev_even x1
    y1_rev_odd x1   =  E
    y1_rev_even x1   =  E
  t@(A x1) in
    rev_odd t   =  rev_odd x1
    rev_even t   =  rev_even x1
    y1_rev_odd x1   =  A (y1_rev_even t)
    y1_rev_even x1   =  A (y1_rev_odd t)
  t@(B x1) in
    rev_odd t   =  rev_odd x1
    rev_even t   =  rev_even x1
    y1_rev_odd x1   =  A (y1_rev_even t)
    y1_rev_even x1   =  B (y1_rev_odd t)
  t@E in
```

```
    rev_odd t   =  y1_rev_odd t
    rev_even t   =  y1_rev_even t
endmag
```

□

This resulting attributed tree transducer is composed again with `Rev`, yielding the following tree transducer:

```
{- replace compositions by one attribute call:
  s_rev (Root (s_rev_s_even (Root t))) = s_rev_s_even_s_rev (Root t) -}
beginmag Rev_o_Even_o_Rev [Att]
  input Data
  root Root
  syn s_rev_s_even_s_rev
  syn rev_even_rev
  syn rev_odd_rev
  syn y1_rev_odd_y1_rev
  syn y1_rev_even_y1_rev
  inh y1_rev_odd_rev
  inh y1_rev_even_rev
  inh rev_even_y1_rev
  inh rev_odd_y1_rev
  t@(Root x1) in
    s_rev_s_even_s_rev t   =  rev_even_rev x1
    y1_rev_even_rev x1   =  y1_rev_even_y1_rev x1
    y1_rev_odd_rev x1   =  y1_rev_odd_y1_rev x1
    rev_odd_y1_rev x1   =  E
    rev_even_y1_rev x1   =  E
  t@(A x1) in
    rev_odd_rev t   =  rev_odd_rev x1
    rev_even_rev t   =  rev_even_rev x1
    y1_rev_even_rev x1   =  y1_rev_odd_rev t
    y1_rev_odd_rev x1   =  y1_rev_even_rev t
    rev_odd_y1_rev x1   =  rev_odd_y1_rev t
    rev_even_y1_rev x1   =  rev_even_y1_rev t
    y1_rev_even_y1_rev t   =  A (y1_rev_odd_y1_rev x1)
    y1_rev_odd_y1_rev t   =  A (y1_rev_even_y1_rev x1)
  t@(B x1) in
    rev_odd_rev t   =  rev_odd_rev x1
    rev_even_rev t   =  rev_even_rev x1
    y1_rev_even_rev x1   =  y1_rev_odd_rev t
    y1_rev_odd_rev x1   =  y1_rev_even_rev t
    rev_odd_y1_rev x1   =  rev_odd_y1_rev t
    rev_even_y1_rev x1   =  rev_even_y1_rev t
    y1_rev_even_y1_rev t   =  A (y1_rev_odd_y1_rev x1)
    y1_rev_odd_y1_rev t   =  B (y1_rev_even_y1_rev x1)
  t@E in
    rev_odd_rev t   =  y1_rev_odd_rev t
    rev_even_rev t   =  y1_rev_even_rev t
    y1_rev_even_y1_rev t   =  rev_even_y1_rev t
    y1_rev_odd_y1_rev t   =  rev_odd_y1_rev t
endmag
```

□

In order to obtain a functional program, this attributed tree transducer can be transformed to a macro tree transducer:

```
beginmag Rev_o_Even_o_Rev [Mac]
  input Data
  root Root
  syn y1_rev_even_y1_rev :: List -> List -> List -> List -> List -> List
```

```
  syn y1_rev_odd_y1_rev :: List -> List -> List -> List -> List -> List
  syn rev_odd_rev :: List -> List -> List -> List -> List -> List
  syn rev_even_rev :: List -> List -> List -> List -> List -> List
  syn s_rev_s_even_s_rev
  s_rev_s_even_s_rev (Root x1)
                    = rev_even_rev x1
                        (y1_rev_odd_y1_rev x1 E (y1_rev_even_y1_rev x1 E E E E) E E)
                        (y1_rev_even_y1_rev x1 (y1_rev_odd_y1_rev x1 E E E E) E E E)
                                      E E
  rev_even_rev (A x1) y1 y2 y3 y4  =  rev_even_rev x1 y2 y1 y3 y4
  rev_even_rev (B x1) y1 y2 y3 y4  =  rev_even_rev x1 y2 y1 y3 y4
  rev_even_rev E y1 y2 y3 y4  =  y2
  rev_odd_rev (A x1) y1 y2 y3 y4  =  rev_odd_rev x1 y2 y1 y3 y4
  rev_odd_rev (B x1) y1 y2 y3 y4  =  rev_odd_rev x1 y2 y1 y3 y4
  rev_odd_rev E y1 y2 y3 y4  =  y1
  y1_rev_even_y1_rev (A x1) y1 y2 y3 y4  =  A (y1_rev_odd_y1_rev x1 y2 y1 y3 y4)
  y1_rev_even_y1_rev (B x1) y1 y2 y3 y4  =  A (y1_rev_odd_y1_rev x1 y2 y1 y3 y4)
  y1_rev_even_y1_rev E y1 y2 y3 y4  =  y3
  y1_rev_odd_y1_rev (A x1) y1 y2 y3 y4  =  A (y1_rev_even_y1_rev x1 y2 y1 y3 y4)
  y1_rev_odd_y1_rev (B x1) y1 y2 y3 y4  =  B (y1_rev_even_y1_rev x1 y2 y1 y3 y4)
  y1_rev_odd_y1_rev E y1 y2 y3 y4  =  y4
endmag
```

□

The comments generated so far in the source code, assure us that:

`s_rev (Root (s_even (Root (s_rev (Root t))))) = s_rev_s_even_s_rev (Root t).`

However, the constructed tree transducer is barely readable and obviously redundant in the sense that context parameters are carried around that are not used. Using the "pruning" method eliminates these redundancies and leads to the following program:

```
{- replace function calls with pruned versions:
  y1_rev_odd_y1_rev t y1 y2 y3 y4 = y1_rev_odd_y1_rev_ t y3 y4
  y1_rev_even_y1_rev t y1 y2 y3 y4 = y1_rev_even_y1_rev_ t y3 y4
  rev_odd_rev t y1 y2 y3 y4 = rev_odd_rev_ t y1 y2
  rev_even_rev t y1 y2 y3 y4 = rev_even_rev_ t y1 y2 -}
beginmag Rev_o_Even_o_Rev [Mac]
  input Data
  root Root
  syn s_rev_s_even_s_rev
  syn rev_even_rev_ :: List -> List -> List -> List
  syn rev_odd_rev_ :: List -> List -> List -> List
  syn y1_rev_even_y1_rev_ :: List -> List -> List -> List
  syn y1_rev_odd_y1_rev_ :: List -> List -> List -> List
  s_rev_s_even_s_rev (Root x1) = rev_even_rev_ x1 (y1_rev_odd_y1_rev_ x1 E E)
                                                  (y1_rev_even_y1_rev_ x1 E E)
  rev_even_rev_ (A x1) y1 y2  =  rev_even_rev_ x1 y2 y1
  rev_even_rev_ (B x1) y1 y2  =  rev_even_rev_ x1 y2 y1
  rev_even_rev_ E y1 y2  =  y2
  rev_odd_rev_ (A x1) y1 y2  =  rev_odd_rev_ x1 y2 y1
  rev_odd_rev_ (B x1) y1 y2  =  rev_odd_rev_ x1 y2 y1
  rev_odd_rev_ E y1 y2  =  y1
  y1_rev_even_y1_rev_ (A x1) y3 y4  =  A (y1_rev_odd_y1_rev_ x1 y3 y4)
  y1_rev_even_y1_rev_ (B x1) y3 y4  =  A (y1_rev_odd_y1_rev_ x1 y3 y4)
  y1_rev_even_y1_rev_ E y3 y4  =  y3
  y1_rev_odd_y1_rev_ (A x1) y3 y4  =  A (y1_rev_even_y1_rev_ x1 y3 y4)
  y1_rev_odd_y1_rev_ (B x1) y3 y4  =  B (y1_rev_even_y1_rev_ x1 y3 y4)
  y1_rev_odd_y1_rev_ E y3 y4  =  y4
endmag
```

□

Realizing that `rev_odd_rev_` is never called and giving shorter names to the functions as well as doing some formatting, finally gives the readable program:

```
beginmag Rev_o_Even_o_Rev [Mac]
  input Data
  root Root
  syn s_rev_s_even_s_rev
  syn y_even :: List -> List -> List -> List
  syn y_odd  :: List -> List -> List -> List
  syn rer    :: List -> List -> List -> List

  s_rev_s_even_s_rev (Root x1)  =  rer x1 (y_odd x1 E E) (y_even x1 E E)

  rer (A x1) y1 y2  =  rer x1 y2 y1
  rer (B x1) y1 y2  =  rer x1 y2 y1
  rer E      y1 y2  =  y2

  y_even (A x1) y1 y2  =  A (y_odd x1 y1 y2)
  y_even (B x1) y1 y2  =  A (y_odd x1 y1 y2)
  y_even E      y1 y2  =  y1

  y_odd (A x1) y1 y2  =  A (y_even x1 y1 y2)
  y_odd (B x1) y1 y2  =  B (y_even x1 y1 y2)
  y_odd E      y1 y2  =  y2
endmag
```

□

A macro tree transducer solving our initial problem has been created, which moreover is more efficient than the modular solution composing tree transducers, where our measure of efficiency is the number of reduction steps using call–by–need evaluation.
Given, for example, the input list `t = A (B (B E))`, call–by–need evaluation of the expression `s_rev (Root (s_even (Root (s_rev (Root t)))))` to `A (A (B E))` takes 15 steps, whereas the same value is obtained from `s_rev_s_even_s_rev (Root t)` with 9 reduction steps. In general, for an input with n symbols, the modular solution reduces in $3n + 3$ steps and our optimized macro tree transducer solution in just $2n + 1$ steps.

References

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31:71–145, 1985.

[FV98] Z. Fülöp and H. Vogler. *Syntax-directed semantics — Formal models based on tree transducers.* Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1998.

[Gan83] H. Ganzinger. Increasing modularity and language–independency in automatically generated compilers. *Sci. of Comput. Prog.*, 3:223–278, 1983.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN'84, Symposium on Compiler Construction, SIGPLAN Notices Vol.19, No.6*, pages 157–170, June 1984.

[Gie88] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

[Höf98] M. Höff. Vergleich der Berechnungsstärken von Klassen eingeschränkter Tree Transducer. Großer Beleg, Dresden University of Technology, 1998.

[Höf99] M. Höff. Vergleich von Verfahren zur Elimination von Zwischenergebnissen bei funktionalen Programmen. Master's thesis, Dresden University of Technology, September 1999.

[Küh98] A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology & Theoretical Computer Science, Chennai, India, Proceedings*, volume 1530 of *LNCS*, pages 146–157. Springer-Verlag, December 1998.

[Küh99] A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In A. Middeldorp and T. Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming, Tsukuba, Japan, Proceedings*, volume 1722 of *LNCS*, pages 114–130. Springer-Verlag, November 1999.

[Küh00] A. Kühnemann. Attribute grammars and program optimization. Script, Dresden University of Technology, Department of Computer Science, 2000.

[Les99] C. Lescher. Entwurf und Implementierung einer Eingabesprache für ein System zur Erzeugung syntaxgesteuerter Editoren. Großer Beleg, Dresden University of Technology, 1999.

[Voi01] J. Voigtländer. Composition of restricted macro tree transducers. Master's thesis, Dresden University of Technology, March 2001. *http://wwwtcs.inf.tu-dresden.de/~voigt/mscthesis.ps.gz.*

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoret. Comput. Sci.*, 73:231–248, 1990.