Deforestation of Functional Programs

François Dupressoir
under the supervision of Robin Cockett

Spring-Summer 2007

This document reports the work done at the University of Calgary, Canada for partial fulfillment of first year Master at the École Normale Supérieure de Lyon, France.

Deforestation of Functional Programs

François Dupressoir

The need for formal proofs of programs made it necessary to produce high-level programming languages based on mathematical theory. But this abstraction came with its cost, particularly in terms of performance.

During this traineeship, we were especially interested in the extra cost induced at execution time by the allocation and destruction of temporary intermediate results. Indeed, functional programs especially, make great use of a *compositional style*, using lists, or other datatypes to glue functions together, thus inducing the fore mentioned extra cost.

The goal of *deforestation*, as first presented by Philip WADLER in 1988, is to compute *at compile-time* the interesting function compositions, removing by that mean the need for an intermediate result, and the overhead due to its building and destruction.

Since then, a lot of techniques have been formally developed to effectively perform the deforestation. Other techniques, that had been used before without noticing this side-effect were proved to perform the transformation. Yet, only a few compilers include this optimization, often in weaker forms, as most of these techniques can only be applied in particular cases that require a costly analysis of the source program.

During this traineeship, I reviewed the existing deforestation techniques and tried to establish formal links between them, highlighting the cases where they are equivalent and those where they behave in sensibly different ways.

Contents

Introduction

Working Environment

The internship this document reports has been carried out at the Programming Languages Lab, at the University of Calgary, under the supervision of Robin Cockett. It was to be, at first, the natural continuation -and end- of a project I had to wrap up too quickly to our taste: the CHOP project. CHOP -the report of which is given in appendix B- aimed at studying the problem of deforestation and the already proposed solutions to eventually program a deforesting compiler for a simple functional language. But even at the end of the first project, it appeared that the topic was too big to be studied in the four month I had left.

Initial State

The report in appendix B contains all the early work, that was done between January and April 2007. However, some of its content will be repeated -and sometimes corrected- in this report, since it sometimes is necessary to go back to previous work to explain new ideas. Yet, I think it is important to read appendix B to get an idea of the state in which the study was at the beginning of the internship, as well as get an idea of what deforestation really is about.

The Categorical Approach

At the end of April, when the CHOP report was written, all "traditional" approaches but tree transducers looked like they had reached a dead end, and even tree transducers were

limited, ironically, by the vast variety of their names, classes and presentations. This is why we decided, before going further ahead, to take a closer look at the categorical approaches that were mentioned in [Jür03]. This gave rise to a lot of discussions on category theory, a summary of which is given in appendix A. Although I did not use their content for studying categorical and monadic transducers, these lectures helped me get a better and deeper understanding of all other approaches to deforestation. Actually, maybe should I write "broader" instead of "deeper" as seeing the problem from a categorical point of view lifts it to a point where the approach you take to solve does not matter anymore, and all of them appear to be the same.

The Ultimate Goal

And this is what showed up as a real goal to this internship: trying to connect the dots between all the techniques that we had encountered during our studies of deforestation. After having spent a semester looking at deforestation mostly through optimizing goggles, as shown in appendix B, and briefly recalled in section 1, I first started the internship time by studying more in depth tree transducers and their composition properties, as I will do in section 2. I then went back to Cockett's proof theoretical approach, as will be done in section 3, to eventually try and lift the problem to a more categorical point of view to try and link up the things that had not been linked before (section 4).

Chapter 1

Deforestation as an optimization

1.1 Wadler's Deforestation Algorithm

A full understanding of this first section might require the reading of appendix B and [Wad90]. Yet, it is possible to get the main ideas without doing so.
In [Wad90], Philip WADLER put together the basic idea of deforestation: remove intermediate data structures from modular functional programs to improve their performance. At the same time, he introduced the first deforestation technique: his Deforestation Algorithm. This section won't get into the details of it, and, especially, won't define the *treeless form*, that requires too much language background that has not enough to do with the goal of my internship to be here.

1.1.1 Wadler's Deforestation Theorem

One important thing that WADLER does in [Wad90] is giving the following theorem.

Theorem 1 ***Deforestation Theorem***
Every function written as a composition of functions in treeless form can be rewritten without loss of efficiency as a single function in treeless form.

This theorem, and its formulation, paved the way to all subsequent and parallel work on deforestation, by giving the process a goal: rewrite, at compile time, all compositions of functions as a single function, possibly without loss of efficiency, it being measured in calls to constructors and `case` constructs.

1.1.2 Wadler's Deforestation Algorithm

With this optimization goal in mind, WADLER designed a transformation system, for which the rules are given in figure 1.1.
These rules, as underlined in the original article, are not sufficient to define a working

(1) $T\,[[v]] \quad = v$

(2) $T\,[[c\ t_1\ \ldots\ t_k]] \quad = c\ T\,[[t_1]]\ \ldots\ T\,[[t_k]]$

(3) $T\,[[f\ t_1\ \ldots\ t_k]] \quad = T\,[[t[t_1/v_1, \ldots, t_k/v_k]]]$
where f is defined
by $f\ v_1\ \ldots\ v_k = t$

(4) $T\,[[\texttt{case}\ v\ \texttt{of}\ p_1\ \rightarrow\ t_1 \mid \ldots \mid p_n\ \rightarrow\ t_n]]$
$= \texttt{case}\ v\ \texttt{of}$
$p_1\ \rightarrow\ T\,[[\ t_1]]$
$\mid p_n\ \rightarrow\ T\,[[\ t_n]]$

(5) $T\,[[\texttt{case}\ c\ t_1\ \ldots\ t_k\ \texttt{of}\ p'_1\ \rightarrow\ t'_1 \mid \ldots \mid p'_n\ \rightarrow\ t'_n]]$
$= T\,[[\ t'_i[t_1/v_1, \ldots, t_k/v_k]]]$
where $p'_i = c\ v_1\ \ldots\ v_k$

(6) $T\,[[\texttt{case}\ f\ t_1\ \ldots\ t_k\ \texttt{of}\ p'_1\ \rightarrow\ t'_1 \mid \ldots \mid p'_n\ \rightarrow\ t'_n]]$
$= T[[\texttt{case}\ t[t_1/v_1, \ldots, t_k/v_k]\ \texttt{of}$
$p'_1\ \rightarrow\ t'_1$
$\mid p'_n\ \rightarrow\ t'_n]]$

(7) $T[[\texttt{case}$
$(\texttt{case}\ t_0\ \texttt{of}\ \ p_1\ \rightarrow\ t_1\ \mid \ldots \mid p_m\ \rightarrow\ t_m)$
$\texttt{of}\ \ p'_1\ \rightarrow\ t'_1 \mid \ldots \mid p'_n\ \rightarrow\ t'_n]]$
$= T[[\texttt{case}\ t_0\ \texttt{of}$
$p_1\ \rightarrow\ (\texttt{case}\ t_1\ \texttt{of}\ \ p'_1\ \rightarrow\ t'_1 \mid p'n\ \rightarrow\ t'_n)$
$\mid \ldots$
$\mid p_m\ \rightarrow\ (\texttt{case}\ t_m\ \texttt{of}\ p'_1\ \rightarrow\ t'_1 \mid p'_n\ \rightarrow\ t'_n)]]$

Figure 1.1: The rules for WADLER's Deforestation Algorithm.

algorithm, as applying them to a recursive function leads to an infinite computation. Yet, it is proved that such an infinite computation is cyclic, and that reaching one cycle allows us to terminate the transformation by introducing one single deforested version of the function, completely defined by the transformation steps we unwound.

This way of looking for cyclic computations and tying knots to terminate them is another important part in many deforestation techniques. Thus, the idea of cycles will be encountered again throughout this report.

1.2 Short-cut fusion

But WADLER's algorithm was not efficient and stable enough to be used as an effective optimization technique: the prior analysis to make sure the involved functions are in treeless form, and the ghost of non-termination lurking behind every computation were too scary. Yet, the idea of unfolding the program using transformation rules and folding it again as a single function, that was used in WADLER's algorithm, gave other ideas to one of his students.

1.2.1 The *foldr/build* rule

These ideas are that, instead of computing *all* function composition, we can have a finite set of rules allowing to perform deforestation at a lower-level. It was first applied to lists, and implemented as *short cut fusion* in the Gnu haskell Compiler, by Andrew GILL ([GLP93, Gil96]). The basic algorithm relies on a correct rewriting of basic list consuming functions using the `foldr` function, list producing functions using the `build` function, and to use a simple deforestation rule to eliminate the composition of `foldr` and `build`: the *foldr/build* rule. Definitions for `foldr` and `build`, as well as the *foldr/build* rule are given in figure 1.2. In fact, it is really a "Deforestation Technique", in the

```
foldr x k z =
    case x of
       Nil -> z
     | Cons(x,xs) -> k x (foldr xs k z)

build g = g (\ x,xs.Cons(x,xs)) []

foldr (build g) k z = g k z
```

Figure 1.2: The basics of short-cut fusion.

sense that it rewrites the composition of `foldr` and `build` as one single function: the

identity.

1.2.2 Generalizing the *foldr/build* rule

Category theory, and especially categorical type theory, led to hope for a generalized version of short cut fusion, that would use the universality of `fold` (the unique *catamorphism* associated with a recursive datatype) to remove any intermediate datatype using a simili-*foldr/build* rule for each datatype.
There is, however, a problem with these rules, as highlighted in [Gil96]: it needs to be proved. GILL's initial argument was based on WADLER's free theorems, that are known to hold for lists. But nothing makes them hold for arbitrary initial datatypes in Haskell or other high-level languages. Yet, lots of research have been done on short cut fusion, and some results have been proved: in [Joh01], Patricia JOHANN proves that the *cata/augment* rule (the general pendant of *foldr/build*) holds for languages providing higher-order polymorphic functions and fixed-point recursion.

Chapter 2

Deforestation as Syntactic Composition of Tree Transducers[1]

Tree transducers and their compositional properties have been studied long before Wadler thought of deforestation. Yet, the first time the two concepts were mentioned together was in 1998. In [Küh98], syntactic composition of tree transducers is used to remove intermediate data structures from a functional program, opening the way to a much wider application of the principle, using all available results on composition of tree transducers. The following chapter gives a quick survey of modular tree transducers, which are the most interesting ones to express functional languages, before explaining what we tried to do with them. A survey of all tree transducers and of their interesting composition properties is given in appendix C. It is important to note that, in a tree transduction-driven approach to deforestation, we are only allowed to work with functions that have at most one recursive argument (or active argument). This forbids functions such as `zip` that consumes its two arguments at the same time, but allows to write them using mutual recursion.

2.1 Modular tree transducers

These tree transducers were introduced by Joost Engelfriet in [EV91]. In the article, he defines them and compares them to other classes of tree transducers (see appendix

[1] Most of the technical aspects of this chapter was presented at the Colgate University, during Foundational Methods in Computer Science 2007 in Hamilton, New York (for more information, visit http://cs.colgate.edu/faculty/mulry/FMCS2007/FMCS2007.html).

C), but he also proves that modular tree transducers can compute all (total) primitive recursive tree functions. We will see later that a tree function is a function that operates on an initial (recursive,inductive) datatype.

2.1.1 Definitions

Preliminaries

Modular Tree Transducers

Definition 1 *A modular tree transducer is a tuple* $M = (Q, \mathrm{mod}, \Delta, [\Sigma], q_0, R)$ *where* Q *and* Δ *are ranked alphabets,* mod *is a mapping* $Q \to \mathbb{N}*$, $q_0 \in Q$ *such that* $\mathrm{mod}(q) = 1$, $[\Sigma] = (\Sigma_1, \ldots, \Sigma_r)$ *with* $\Sigma_i \subseteq \Delta$ *and* $r = \mathrm{rank}_Q(q_0)$, *and* R *is a finite set such that, for all* $q \in Q^{(r+1)}$ *and for every* $\delta \in \Delta^{(k)}$, *there is exactly one rule of the form* $q(\delta(x_1, \ldots, x_k), y_1, \ldots, y_r) \to \xi$ *in* R, *with* $\xi \in T_{Q \bigcup \Delta}(X \bigcup Y)$ *such that:*

1. *for every* $p \in Q$ *occurring in* ξ, $\mathrm{mod}(p) \geq \mathrm{mod}(q)$.
2. *if* $p(\xi_1, \ldots, \xi_u)$ *occurs in* ξ *with* $p \in Q^{(u)}$, *and* $\mathrm{mod}(p) = \mathrm{mod}(q)$, *then* $\xi_1 \in \{x_1, \ldots, x_k\}$.

If we chose to replace "there is exactly one rule of the form" by "there is at most one rule of the form" in the above definition, we would obtain a class of tree transducers that computes all partial primitive recursive tree functions.

Some Research Ideas on Modular Tree Transducers. It would be interesting to study non-deterministic modular tree transducers. It could also be interesting to study the effects of allowing ξ_1 in condition 2 to be a context parameter.

2.1.2 Going Further

Figure 2.1 shows an example of a modular tree transducer. It is interesting to study it with a programmer's eyes and see how close the expression of R is to the way the program itself would be written in Haskell, the type information being provided by the given Σ and Δ.

$$M_{nrev} = \left(\left\{ \mathtt{rev}^{(1)}, \mathtt{app}^{(2)} \right\}, \begin{pmatrix} \mathtt{rev} \mapsto 1 \\ \mathtt{app} \mapsto 2 \end{pmatrix}, \mathtt{rev}, [\Sigma], \Sigma, R \right) \text{ where } \Sigma = \left\{ cons^{(2)}, nil^{(0)} \right\}$$

$$\text{and } R = \left\{ \begin{array}{ll} \mathtt{app}(cons(x_1, x_2), y) & \rightarrow cons(x_1, \mathtt{app}(x_2, y)) \\ \mathtt{app}(nil, y) & \rightarrow y \\ \mathtt{rev}(cons(x_1, x_2)) & \rightarrow \mathtt{app}(\mathtt{rev}(x_2), cons(x_1, nil)) \\ \mathtt{rev}(nil) & \rightarrow nil \end{array} \right\}$$

Figure 2.1: An example of modular tree transducer: naïve `rev`

2.2 Context tree transducers

One of my main concerns while I was studying tree transducers was to make them fit with WADLER's deforestation algorithm. I was particularly interested in trying to formally define a class of tree transducers that would contain exactly programs in treeless form. I realized afterwards that, as being in treeless form is a property depending on the function definition and not on the function itself, this is not possible. This section, however, contains the result of that first try. It is easy to prove, using bag orders, that the underlying rewriting system is noetherian and confluent, and thus to define the semantics of such tree transducers the way it is done for most classes of tree transducers in [FV98].

Definition 2 *Let Q and Δ be ranked alphabets and m, $n \geq 0$. The set of right-hand sides RHS(Q, Δ, m, n) over Q and Δ with m variables and n parameters is the smallest set rhs$\subseteq T_{Q \cup \Delta}(X_m \cup Y_n)$ satisfying the following three conditions:*

1. *$Y_n \subseteq rhs$.*
2. *For $\delta \in \Delta_k$ with $k \geq 0$ and $\xi_1, \ldots, \xi_k \in rhs$, $\delta(\xi_1, \ldots, \xi_k) \in rhs$.*
3. *For $q \in Q_{k+1}$ with $k \geq 0$, $z \in X_m \bigcup Y_n$ and $(z_i)_{1 \leq i \leq k} \in X_m \bigcup Y_n \backslash \{z\}$ such that $z_i = z_j \in Y_n \Rightarrow i = j$, $q(z, z_1, \ldots, z_k) \in rhs$.*

Definition 3 *A context tree transducer M is a tuple $(Q, \Sigma, \Delta, q_{in}, R)$ consisting of three ranked alphabets Q, Σ and Δ of states, input, and output symbols, respectively, with $\forall q \in Q, rank_Q q \geq 1$, an initial state q_{in} of rank 1, and a finite set of rules R such that: $\forall q \in Q_{n+1}, \sigma \in \Sigma_m, \exists! rhs_{q,\sigma} \in RHS(Q, \Delta, m, n)$ $q(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_n) \rightarrow rhs_{q,\sigma}$.*

I have not proved any composition properties on these yet, although I hope that it is stable for syntactic composition. This would be a sign that the class of functions computed by these tree transducers could, finally, be the class of functions that have a treeless form. It might be interesting to conduct further research on this, starting by characterizing functions that have a treeless form (we already know that all functions that can be written as a composition of functions in treeless form have a treeless form... are they the only ones?).

Chapter 3

Deforestation as Cut Elimination

This chapter assumes that the reader has some basic knowledge of categorical type theory. If that were not the case, reading appendix A now would be highly recommended.

3.1 Circular Proofs

Introduced by SANTOCANALE, this proof system is equivalent to KOZEN's algebra based system (see appendix A for a quick presentation of it, and [Coc01] for a quick proof of the equivalence). However, its intuitive presentation of induction (and co-induction) makes it a very useful tool to reason about proofs, and especially when they represent functional programs, that make extensive use of recursion (and, sometimes, co-recursion). A good example, such as the one given in figure 3.1, might be useful to get a first idea of what circular proofs are about. Figure 3.2 gives the associated term, as the corresponding term syntax will come in handy later. Note that the *cons* rule that is used in figure 3.1 comes from the datatype theory, and is the same as in KOZEN's system (the $fold$ rule is replaced by a circular proof of the $fold$, parametrized by the datatype's functor).

Of course, it is possible to represent mutual recursion using circular proofs, as shown by figures 3.3 and 3.4.

$$\frac{\dfrac{}{List(A) \vdash List(A)}\,id}{\mathbf{1}, List(A) \vdash List(A)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,id}{A, X, List(A) \vdash A} \quad \dfrac{}{A, X, List(A) \vdash List(A)}\,app}{A \times X, List(A) \vdash A \times List(A)}}{A \times X, List(A) \vdash \mathbf{1} + A \times List(A)}\,cons$$

$$\frac{\mathbf{1}, List(A) \vdash List(A) \qquad A \times X, List(A) \vdash List(A)}{\mathbf{1} + A \times X, List(A) \vdash List(A)}$$

$X := List(A) \mid app := xs' : X, ys' : List(A) \vdash (app\ xs'\ ys') : List(A)$

$xs : List(A), ys : List(A) \vdash (app\ xs\ ys) : List(A)$

Figure 3.1: A circular proof: `append`

```
μ app.{app xs ys=
         case xs of
             Nil          -> ys
           | Cons(x',xs') -> Cons(x',app(xs',ys))}
```

Figure 3.2: A circular term: `append`

3.1.1 Hunting Down Proof Equivalences

But, as underlined in [Coc01], having an idea of how circular proofs work and how to transform them into algebra-based rules and vice versa is not sufficient. To fully use them, as well as to prove the two proof systems are isomorphic, it is important to know what proof equivalences are, what makes two circular proofs equivalent. The method used in [Coc01], that is just quickly outlined here, is to study the cut elimination procedure for circular proofs, and then study the effects of cutting against an identity proof, to get an idea of what proof equivalences should be.

Cut Elimination

In proof theory, cut elimination is the process that rewrites a proof using the *cut* rule (figure 3.5) into one that does not use it. For most proof systems, this is possible, and we have algorithms (mostly thanks to Gentzen). Yet, defining a cut elimination procedure for a new system is somewhat tricky.

$$\frac{\dfrac{}{\mathbf{1} \vdash Bool}\, False \quad \dfrac{}{X \vdash Bool}\, even}{\mathbf{1} + X \vdash Bool} \qquad \frac{\dfrac{}{\mathbf{1} \vdash Bool}\, True \quad \dfrac{}{X \vdash Bool}\, odd}{\mathbf{1} + X \vdash Bool}$$

$$\frac{X \mid odd := X \vdash Bool \quad even := X \vdash Bool}{Nat \vdash Bool}$$

Figure 3.3: An example of mutual recursion using circular proofs: `odd`.

```
μodd.{odd x=
          case x of
              Zero     -> False
            | Succ(x') -> even(x')
       even y=
          case y of
              Zero     -> True
              Succ(y') -> odd(y')}
```

Figure 3.4: An example of circular term with mutual recursion: `odd`

Cut Elimination in Circular Proofs

Indeed, there is a pretty intuitive idea of what cut elimination should be for circular proofs. The basic transformation formulas are given in figure 3.6. The idea is to "push" the box as far up as possible to be able to perform cut elimination under it. This is done by "unrolling" its innards, *unfolding* the proof. Of course, their is a drawback: nothing guarantees that the process will -and it does not in the general case- terminate. Yet, we can make one observation. When cutting against an identity proof, any proof resulting from the transformation is necessarily equal to the original one, and yet, we can unroll the proof to any depth. This gives us the possibility to *tie the knot* when we encounter an instance of the original proof. The rule for tying knots is given in figure 3.7, and is the basis of proof equivalences in the circular proof system.

$$\frac{\Gamma \vdash D \quad \Delta, D \vdash C}{\Gamma, \Delta \vdash C}\ cut$$

Figure 3.5: The *cut* rule.

3.2 Cut Elimination and Deforestation

In a certain respect, cut elimination on proofs is like deforestation on programs: you remove from the proof a rule that creates an intermediate proposition that is destroyed right away, when you can rewrite your proof without using this rule. In fact, when you consider proofs as programs, and especially when you consider circular proofs as programs, it becomes clear that deforestation is a form of cut elimination. In fact, it is the removal of what COCKETT calls *deforesting cuts* in [Coc01], cuts that create intermediate datatypes that are used right away, as shown in figure 3.8.

In [Coc01], on top of a theoretical explanation of why deforestation works, a new deforestation technique is sketched through cut elimination. It is basically an application of the rules from figures 3.6 and 3.7, and of standard cut elimination on algebra based proofs. This technique has been studied from a more practical point of view in the report given in appendix B. Figure 3.9 shows an example of cut elimination on a deforesting cut, and figure 3.10 shows the corresponding terms. The notation used here is simpler than the one used in [Coc01], or even appendix B, as we won't, in this report, care too much about the practical aspects of context and accumulating parameters.

Note the $\pi[L(A)]$ appearing in the second step in figure 3.9. We here make extensive use of the fact that the inside of the circular proof is parametric in the variable it binds (in the left upper corner). Also note, in the same figure, that the step labeled "classical cute elimination" does not eliminate the cut. This is due to the presence of the black box on top of the proof. The cut can be pushed as far up as we want, as soon as we unroll enough boxes.

In figure 3.10, the terms are shown in the same position as the corresponding proofs, to help making the parallel. At this time I can only hope they will be put on facing pages by LaTeX, which would help even more.

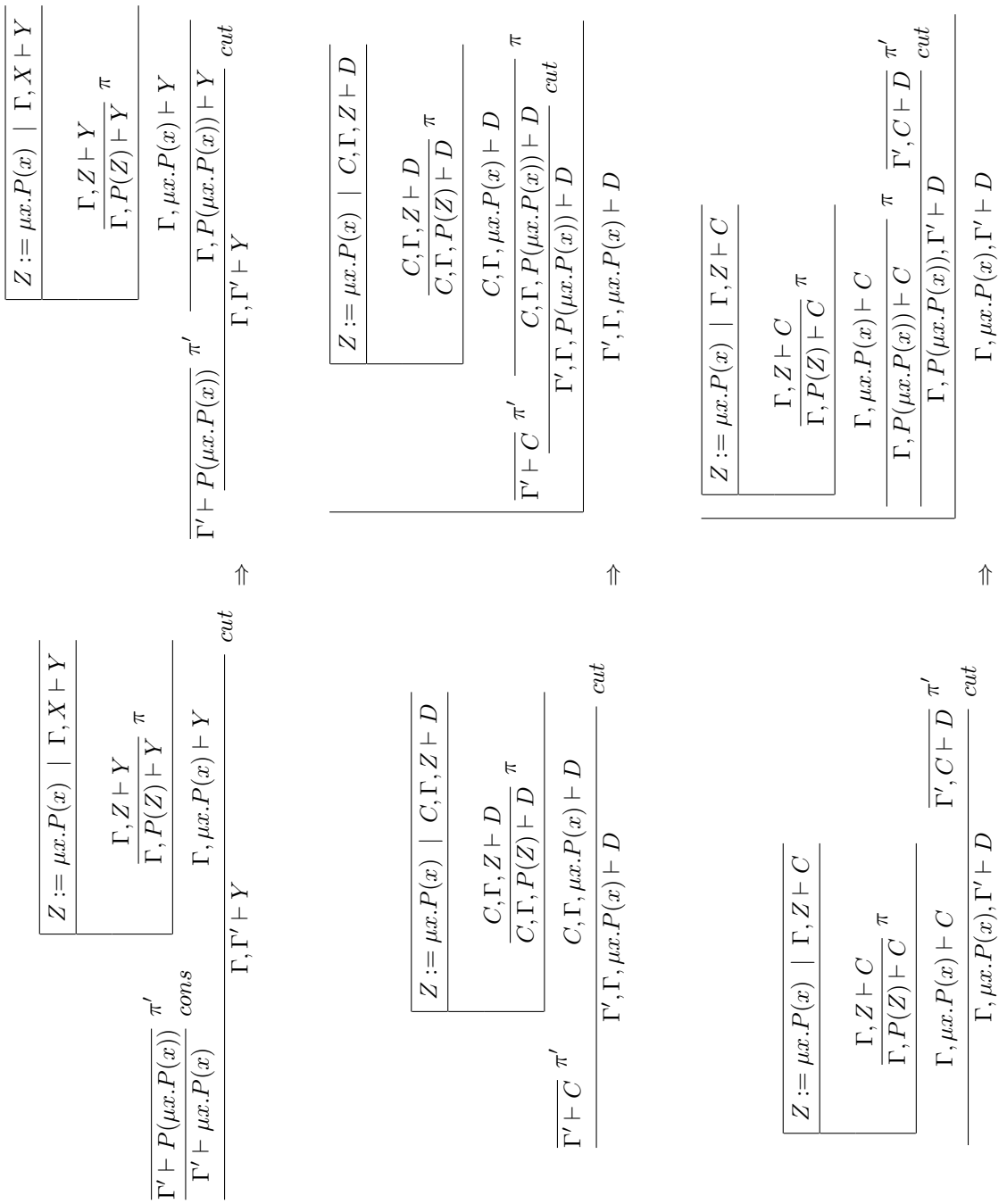


Figure 3.6: Cut elimination steps for circular proofs.

3.2.1 Wadler's Algorithm as Cut Elimination

So, the elimination of deforesting cuts in circular proofs correspond to the idea of deforestation. But does it correspond to the algorithm? At first glance, we could say yes:

$$\dfrac{\dfrac{\Pi}{\Gamma, \mu x.P(x) \vdash Y} = \quad \left|\begin{array}{l}\dfrac{\dfrac{\Pi}{\Gamma, \mu x.P(x) \vdash Y}}{\Gamma, P(\mu x.P(x)) \vdash Y}\,\pi \\ \hline \Gamma, \mu x.P(x) \vdash Y\end{array}\right.}{\dfrac{\Pi}{\Gamma, \mu x.P(x) \vdash Y} = \quad \left|\begin{array}{l}Z := \mu x.P(x) \;\mid\; \Gamma, Z \vdash Y \\ \hline \dfrac{\Gamma, Z \vdash Y}{\Gamma, P(Z) \vdash Y}\,\pi \\ \hline \Gamma, \mu x.P(x) \vdash Y\end{array}\right.}\,knot$$

Figure 3.7: *Knot tying* rule for circular proofs.

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash \mu x.P(x)} \quad \dfrac{\Pi'}{\Delta, \mu x.P(x) \vdash A}}{\Gamma, \Delta \vdash A}\,cut$$

Figure 3.8: Deforesting cut.

for the same initial program, we get approximately the same results. At second glance, however, we would say no: the transformations are not the same, and even the results sometimes differ. Yet, we can investigate further. The way I chose to conduct this investigation is by trying to represent each one of WADLER's transformations (see figure 1.1, page 4) on the level of proofs, using the basic steps of cut elimination for circular proofs, classical cut elimination, and the knot tying rule.

Rules (1), (2) and (4) are just "searching rules", going through the program to look for points of interest. In circular proofs, they could be replaced by some operation locating deforesting cuts.

Rule (3), however is a lot more interesting. It is easy to see that, in the case where t_1 is a function call, this rule corresponds to an application of the second cut elimination step (unfolding the right branch of the tree). Yet, in WADLER's system, functions are automatically unfolded anytime they are encountered, creating extra work as well as extra junk in the program instead of cleaning it up.

Rule (5) and (7) would be performed easily, at proof level, by standard cut elimination as it is done in the example given above.

Rule (6) unfolds the selector of a case construct. At proof level, it can be performed by the first cut elimination step, unfolding the proof on the left side of the *cut*.

Thus, all of WADLER's rules but some instances of rule (3) can be performed through cut

elimination. Moreover, specifically targeting deforesting cuts -as it is allowed by working at proof level- instead of mechanically unrolling all functions we encounter allows to get a cleaner final program.

3.3 Tree Transducers and Circular Proofs

Now that we have seen how closely related circular proofs and deforestation are, it is pretty natural to see if they have any relation with tree transducers, the other side of deforestation. And they do, actually, have at least a shapely closeness. Indeed, the rule of a modular tree transducer, on the one hand, and the terms associated with circular proofs are pretty much the same thing.
It could be just that they both represent a program, but there is more to it than just a closeness. We can consider each module as a box, all boxes being nested in one another, as modules are. Thus, circular proofs could be adapted and used as a graphical and intuitive representation for tree transducers as well.
However, their relationship, as stated intuitively, is -for now, at least- purely a morphological relationship. As far as deforestation is concerned, especially, there is a major difference between the two points of view: Wadler's deforestation algorithm and Cockett's cut elimination both work step by step, using program (resp. proof) isomorphisms to extract a loop and tie the knot, whereas syntactic composition of tree transducers works globally, introducing a bunch of new states and rewriting the whole tree transducer using them.
This difference was the major obstacle I hit during the internship, along with the impossibility to reason decently about the fusion rule (and especially the *build* and *augment* functions) due to its "almost-correctness" (see [Gil96] again for more about fusion).

$$\dfrac{\dfrac{\dfrac{}{X := L(A) \mid app := X, L(A) \vdash L(A)}}{\dfrac{X, L(A) \vdash L(A)}{\mathbf{1} + A \times X, ys : L(A) \vdash L(A)}\,\pi}}{xs : L(A), ys : L(A) \vdash \mathbf{L(A)}} \qquad \dfrac{\dfrac{\dfrac{}{X := L(A) \mid app := X, L(A) \vdash L(A)}}{\dfrac{X, L(A) \vdash L(A)}{\mathbf{1} + A \times X, zs : L(A) \vdash L(A)}\,\pi}}{\mathbf{L(A)}, zs : L(A) \vdash L(A)}$$

$$\dfrac{xs : L(A), ys : L(A) \vdash \mathbf{L(A)} \qquad \mathbf{L(A)}, zs : L(A) \vdash L(A)}{xs : L(A), ys : L(A), zs : L(A) \vdash (app\ (app\ xs\ ys)\ zs) : L(A)}\ cut$$

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{X := L(A) \mid app := X, L(A) \vdash L(A)}}{\dfrac{X, L(A) \vdash L(A)}{\mathbf{1} + A \times X, L(A) \vdash L(A)}\,\pi}}{\dfrac{L(A), L(A) \vdash L(A)}{\mathbf{1} + A \times L(A), L(A) \vdash L(A)}\,\pi[L(A)]} \qquad \dfrac{\dfrac{\dfrac{}{X := L(A) \mid app := X, L(A) \vdash L(A)}}{\dfrac{X, L(A) \vdash L(A)}{\mathbf{1} + A \times X, L(A) \vdash L(A)}\,\pi}}{L(A), L(A) \vdash L(A)}}{\mathbf{1} + A \times L(A), L(A), L(A) \vdash L(A)}\ cut$$

$$L(A), L(A), L(A) \vdash L(A)$$

$$\Downarrow$$

classical cut elimination

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{}{X \mid X, L(A) \vdash L(A)}}{\dfrac{X, L(A) \vdash L(A)}{\mathbf{1} + A \times X, L(A) \vdash L(A)}\,\pi}}{\mathbf{1}, L(A), L(A) \vdash L(A)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,id \qquad \dfrac{\Pi}{L(A), L(A), L(A) \vdash L(A)}}{A \times L(A), L(A), L(A) \vdash A \times L(A)}}{A \times L(A), L(A), L(A) \vdash \mathbf{1} + A \times L(A)}}{A \times L(A), L(A), L(A) \vdash L(A)}$$

$$\dfrac{\mathbf{1}, L(A), L(A) \vdash L(A) \qquad A \times L(A), L(A), L(A) \vdash L(A)}{\mathbf{1} + A \times L(A), L(A), L(A) \vdash L(A)}$$

$$L(A), L(A), L(A) \vdash L(A)$$

$$\Downarrow$$

knot tying

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{Y \mid Y, L(A) \vdash L(A)}}{\dfrac{Y, L(A) \vdash L(A)}{\mathbf{1} + A \times Y, L(A) \vdash L(A)}\,\pi}}{\mathbf{1}, L(A), L(A) \vdash L(A)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,id \qquad \dfrac{}{X, L(A), L(A) \vdash L(A)}\,app^2}{A \times X, L(A), L(A) \vdash A \times L(A)}}{A \times X, L(A), L(A) \vdash \mathbf{1} + A \times L(A)}}{A \times X, L(A), L(A) \vdash L(A)}}{\mathbf{1} + A \times X, L(A), L(A) \vdash L(A)}$$

$$\dfrac{X := L(A) \mid app^2 := X, L(A), L(A) \vdash L(A)}{xs : L(A), ys : L(A), zs : L(A) \vdash L(A)}$$

where π is the inside of the box in figure 3.1 and Π is the original cut-full proof.

Figure 3.9: Cut elimination on a circular proof.

```
app (app xs ys) zs
                                  ⇓
app (case xs of
         Nil        -> ys
       | Cons(x,xs) -> Cons(x,app xs ys)})
    zs
                                  ⇓
case xs of
    Nil        -> app ys zs
  | Cons(x,xs) -> Cons(x,app (app xs ys) zs)
                                  ⇓
μapp².{app² xs ys zs=
          case xs of
              Nil        -> app ys zs
            | Cons(x,xs) -> Cons(x,app² xs ys zs)}
```

Figure 3.10: Terms for figure 3.9.

Chapter 4

Deforestation in Calculational Form

With the introduction of circular proofs, we caught a glimpse of category theory. With the work done on tree transducers in [Jür03], and on short cut fusion in [MT95, Jür03], we have to dive into it. That I have done in appendix A. The following chapter just makes mention of these existing links between the deforestation approaches, without too many details.

4.1 Short Cut Fusion and Category Theory

In [MT95] is exposed and explained how the general $foldr/build$ rule can be seen as an application of a more general categorical theorem: the acid rain theorem.
It introduces two versions of this theorem, one concerning catamorphisms ($fold$), and the other one concerning anamorphisms (their dual). Then, Meijer studies them a little further to develop a deforestation algorithm based on *hylomorphisms* and justified categorically.

4.2 Tree Transducers and Category Theory

In [Jür03], the author establishes a relationship between tree transducers and category theory.

4.2.1 Algebraic Transducers

In a first run, he uses MEIJER and TAKANO's results about the acid rain theorem ([MT95]) to show how the syntactic composition of top-down tree transducers is equivalent to short-cut deforestation. Doing this, he defines *categorical transducers* ([JV04], called *algebraic* transducers in [Jür03]), that transforms an algebra into another. He also defines transformations between top-down tree transducers, and these algebraic transducers, allowing us to freely use results from both worlds.

4.2.2 Monadic Transducers

But going further than that, JÜRGENSEN defines another kind of transducers, working on monads: *monadic transducers*. Once again, he shows that their composition is short cut fusion and explains how to go back and forth from monadic transducers to top-down and macro tree transducers. Yet, there is a problem, as you will probably have noticed: macro tree transducers are not stable under composition. Thus, we can get "stuck" in monad-land after lifting two macro tree transducers and performing the fusion.
This is definitely promising future work. The main question is if monadic transducers can represent modular tree transducers or if they are limited to finite compositions of macro tree transducers, and the main objective is to define the necessary transformation steps to be able to use that powerful tool to its full extent.

Conclusion

Results

There are no real results to this internship. Maybe a few answers were given, a few diagrams were drawn that were never drawn before. But I have to say: there are more questions than answers at the end of four months of work.
The enormous bibliographic work I did on tree transducers revealed a few dead ends, that have never been re-explored, and especially the ever so present modular tree transducer, maybe saved for later. Their interest to deforestation is still unclear, but I think they deserve more attention.
As to deforestation itself, there is still no miracle technique. Short cut fusion is part of the GHC, and some sort of $unfold/fold$ technique has been implemented in another language. Both of them have their drawbacks, the most important one being that we don't know if and when they are going to go crazy or not terminate. The fact that it has been implemented before even being proved shows, however, that the performance gain is worth the risk. Gill's experimental measures of computation times sometimes gain many orders of magnitude. Allowing functional programs to get rid of so many memory accesses while keeping a clean and easy programming style would probably, and most definitely benefit them.

Future Work

Almost everything presented here is work in progress. There are some areas of tree transducer theory that are pretty much closed to new interesting results, but apart from that, there is much work to do on all aspects. Some of the work that has to be done has been mentioned in the corresponding chapters. Some of it has not. Here is an unsorted

list of things to do:

- Sort out properties of context tree transducers.
- Sort out a representation of the *foldr/build* rule
  1. as an instance of WADLER's deforestation algorithm.
  2. as cut elimination in a circular proof.
- Tidy up modular tree transducers and inter-modular composition properties (as in [KGK01] or even in the general case).
- Think about deforestation of co-inductive datatypes and the ethical problems it raises (`rev (rev l) = l`).
- Is short cut deforestation correct for Haskell? for other lazy languages?
- Deforestation and strict languages?
- ...(I probably forgot the most obvious ones and the ones I had in mind when I started typing the list)

Shrink Zone

This internship, although it is still work in progress, brought me more experience than anything before.
The fact that I was let almost completely free to go wherever I wanted on the project, that started with the sole and only reading of WADLER's article ([Wad90] motivated me to let my mind wander and ask so many stupid questions, that I ended up not being able to correctly structure my report.
Moreover, the interconnections between all the aspects of a single -and in appearance simple- problem, that later became the main topic of my research gave me a renewed interest for theory of programming language. The kind of interest that whispers: "Hey, even if you get bored of this with me, i still have that to offer."
Last, but not least, having been able to give a talk -even quick and technical- at a conference -even minor and very specialized- gave me the will to keep doing it. Meeting those researchers from all over the place, all having one language and one passion in common, made me decide it was my world... well, the one I wanted to be in.

In any case, I have to thank Robin for his support and the freedom, trust, and category theory lectures he gave me. I also have to thank Phil MULRY for organizing FMCS this year: Hamilton is a very fine place, and I had already been to Kananaskis... To all participants, thanks again for what you have done to me and for listening to my unrelated talk. Many thanks go to Sean NICHOLS, for the help he gave me when I was drowning in commutative diagrams, and to Nin TANG, who gave up his seat at FMCS -allowing me to go- because he hadn't understood anything last year... I should have listened to him.

Appendix A

Useful Category Theory: an Introduction to Datatypes.

This chapter presents only the notions and concepts that are necessary to the reading of this report. It does not pretend to be anything else but what it is: my understanding of the short lectures Robin gave me, and the very few things I picked up from MacLane, Awodey and Crole's books.

A.1 A Gentle Introduction

Categories are about structures (or *objects*) and relations (or *maps*) between them, without matter of what kind of objects, or what kind of maps. There are just some rules regarding maps: each object has to have an identity map, and maps compose correctly regarding objects and identities. More formally, we could give the following definition:

Definition 4 *A category $\mathcal{C}$ is specified by the following data:*

- *a collection $ob(\mathcal{C})$ of objects,*
- *a collection $map\mathcal{C})$ of morphisms,*
- *two operations, assigning to each morphism f its* domain $dom(f)$ *and its* codomain $cod(f)$. *We will use traditional function notations to express that typing ($f : dom(f) \to cod(f)$),*
- *a* composition *operator that associates with every pair (f, g) of morphisms such that $f : A \to B$ and $g : B \to C$ the morphism $g \circ f = gf = f; g : A \to C$, such that, when $f : A \to B$, $g : B \to C$ and $h : C \to D$,*

$$(hg)f = h(gf) = hgf$$

,

- *an operation associating, to each object A in $ob(\mathcal{C})$, its identity map id_A such that*

$$id_{cod(f)} \circ f = f$$

$$f \circ id_{dom(f)} = f$$

.

One important notion in category theory is duality. As it is pointless to introduce it here, I won't, but it is interesting to read about it.

Definition 5 *Let $\mathcal{C}$ be a category. An* initial object *in $\mathcal{C}$ is an object 0 such that, for every object A in $\mathcal{C}$, there is a unique map $!_A : 0 \to A$.*

A.1.1 Functors

We might want to transform a category into some other, for the sake of a proof, or just for the thrill. *Functors* allow to do it while preserving the categorical structure.

Definition 6 *A functor $F : \mathcal{C} \to \mathcal{D}$ is a pair (F_o, F_m) of functions such that*

$$F_o : ob(\mathcal{C}) \to ob(\mathcal{D})$$

$$F_m : map(\mathcal{C}) \to map(\mathcal{D})$$

$$f : A \to B \Rightarrow F\ f : F\ A \to F\ B$$

$$F(id_A) = id_{F\ A}$$

$$F(f; g) = F\ f; F\ g$$

Usually, all of F,F_o and F_m are noted F, as there is no possible confusion. It is easy to check that the image of a category through a functor is a category.

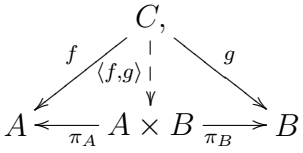
A.1.2 Products

Definition 7 *A* binary product *of objects A and B in a category $\mathcal{C}$ is:*

- *an object $A \times B$ of $\mathcal{C}$, together with*
- *two projection maps $\pi_A : A \times B \to A$ and $\pi_B : A \times B \to B$*

such that, for all given object C and maps $f : C \to A$ and $g : C \to B$ of $\mathcal{C}$, there is a unique morphism $\langle f, g\rangle : C \to A \times B$ for which $\pi_A \langle f, g\rangle = f$ and $\pi_B \langle f, g\rangle = g$. We will refer to the binary product as $A \times B$ instead of the full triplet.

The following commutative diagram shows a little more intuitively what is meant by the definition:



There exists a dual notion to products: coproducts. It is not important to introduce them here, as I will later only talk about functors and algebras, but it is good to know it exists. (coproducts are used to represent the "sum types").

A.1.3 Algebras

Definition 8 *Let $\mathcal{C}$ be a category, and $F : \mathcal{C} \to \mathcal{C}$ be a functor. An F-algebra is an object A of $\mathcal{C}$, called the* carrier*, together with a map $f : F\ A \to A$. A morphism of F algebras is a map $h : (A, f) \to (B, g)$ between the carriers such that*

$$\begin{array}{ccc} F(A) & \xrightarrow{F(h)} & F(B) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ A & \xrightarrow[h]{} & B \end{array}$$

commutes.

Note that, for any given endofunctor F of a category $\mathcal{C}$, the collection of F-algebras form a category $Alg_F(\mathcal{C})$.

Initial F-algebras

Definition 9 *An initial F-algebra is an F-algebra $(F^@, cons)$ such that, given any F-algebra (A, f), there is a unique map $fold_F(f)$ such that*

$$\begin{array}{ccc} F(F^@) & \xrightarrow{cons} & F^@ \\ {\scriptstyle F(fold_F(f))}\downarrow & & \downarrow{\scriptstyle fold_F(f)} \\ F(A) & \xrightarrow[f]{} & A \end{array}$$

commutes. Note that $(F^@, cons)$ is precisely an initial object in $Alg_F(\mathcal{C})$.
Initial F-algebras are also commonly called fixed points, as it is what they are in posets.

A.1.4 Combinators

Definition 10 *Let $\mathcal{C}$ be a category, and F and G be endofunctors of $\mathcal{C}$. A combinator c is an operator* $\dfrac{F\ X \xrightarrow{h} G\ X}{F'\ X \xrightarrow{c[h]} G'\ X}\ c$ *such that, in* $\dfrac{\begin{array}{ccc} F'\ X & \xrightarrow{c[h]} & G'\ X \\ {\scriptstyle F'\ X}\downarrow & & \downarrow{\scriptstyle G'\ X} \\ F'\ X' & \xrightarrow[c[h']]{} & G'\ X' \end{array}}{\begin{array}{ccc} F\ X & \xrightarrow{h} & G\ X \\ {\scriptstyle F\ X}\downarrow & & \downarrow{\scriptstyle G\ X} \\ F\ X' & \xrightarrow[h']{} & G\ X' \end{array}}\ c$ *, if the top diagram commutes, then the bottom one commutes, too.*

A.2 Categorical Datatypes

In this framework, a datatype is exactly an initial F-algebra for a given F. For example, lists are the fixed point of the functor $F(X) = \mathbf{1} + A \times X$ (here, $+$ is a coproduct and $\mathbf{1}$ is a terminal object, but this is unimportant to the argument).
For a given functor F, the initial F-algebra (initial objects are unique up to isomorphism) will be noted $\mu x.F(x)$. But this is just one point of view on datatypes. The second most common point of view is combinatorial.

A.2.1 Kozen rules

Let F be an endofunctor in a category $\mathcal{C}$, and $(F^@, cons)$ an arbitrary F-algebra. If there is a combinator $\dfrac{F\ A \xrightarrow{f} A}{F^@ \xrightarrow{fold[f]} A}\ fold$ such that $cons; fold[f] = F(fold[f]); f$ and $fold[cons] = id_{F^@}$, then $(F^@, cons)$ is an initial F-algebra (and thus, we have a new equivalent definition for a datatype).
From a type theoretical point of view, this gave the proof system known as the KOZEN rules, given in figure A.1. The advantage of that definition is that we have no need to check for any unicity (whereas we have to do so to prove that some F-algebra is initial). Yet, dealing with combinators might not be the easiest thing to do.

$$\frac{F\ A \xrightarrow{f} A}{\mu x.F(x) \xrightarrow{fold[f]} A}\ fold \qquad \frac{X \to F(\mu x.F(x))}{X \to \mu x.F(x)}\ cons$$

Figure A.1: KOZEN rules.

This is all the category theory that is needed here, even if more of it might prove itself useful. Yet, as already stated, this report is not intended to be lecture notes in category theory.

Appendix B

The CHOP Project

The following is the report submitted as partial requirement for course CPSC503 in Winter 2007 at the University of Calgary. It contains a comparative study of different deforestation methods and introduces CHOP, a technique midway between Cockett's cut-elimination and Wadler's deforestation.

CHOP
Final Report

François Dupressoir
University of Calgary, Alberta - Department of Computer Science
École Normale Supérieure de Lyon, France - DMI

Abstract

Functional languages are often compared negatively to imperative languages on their performance. Yet, the clear programming style they allow makes them a good tool. This makes the problem of optimizing functional programs particularly important and motivated Phil Wadler's work on deforestation. Deforestation *is an optimization that removes intermediate data structures in functional programs at compile-time, avoiding unnecessary memory access and garbage collection. This paper reviews some of the methods used so far to perform deforestation before presenting a new method:* CHOP.

1. Introduction

Functional programs are often written in a very readable style that looks much more like what they are supposed to do at a high-level than to what they actually do at a low-level. But this facilitated writing and understanding has its cost, and the heavy use of intermediate data structures needed to "glue" those function compositions together require that the system builds and destroys heap objects at run-time. The idea behind *deforestation*, as Philip Wadler introduced it in [5], is to make the compiler transform an elegantly-written program into an efficient one, which removes building and destroying of intermediate structures. This optimization is especially useful for functional programs, as they make heavy use of the compositional style that makes them concise and clear.

1.1. This paper

This paper aims at reviewing the main methods used to study and perform deforestation on functional programs, as well as to present a different way of performing it: CHOP. This review will be made in section 3, along with a comparison of these methods on some examples. Then, in section 4, CHOP will be presented as formally as possible before being shown at work on the examples. Its limitations will then be analyzed, along with suggestions on how to solve them.

$$
\begin{aligned}
t ::= \;& v \mid \backslash v \;\rightarrow\; t \mid t_1\, t_2 \mid letrec\, f\, v \;=\; t \\
& \mid letrec\, f\, v\, [va_1, \ldots, va_n]\, [vc_1, \ldots, vc_m] \;=\; t \\
& \mid (t_1, \ldots, t_n) \mid \#^n_m\, t \mid @^n_m\, t \\
& \mid case\, t\, of \quad @^n_1\, t \;\rightarrow\; t_1 \mid \ldots \mid @^n_i\, t \;\rightarrow\; t_i
\end{aligned}
$$

Figure 1. The terms of the working language.

2. A language to work on.

2.1. Type system

A basic parametric type system is assumed, as is present in most functional languages. This allows the use of datatypes and polymorphism. As we would like to include coinductive datatypes in future work, a more elaborate type system will be required.

2.2. Term formation

The high-level grammar for terms is as shown on figure 1. The language has lambda-abstractions and `letrec` statements (which will be used as well to define non-recursive functions, but only functions, in a first approach). It also has general tuples with the associated general projectors ($\#^n_m$ extracts the mth value from a n-uple), and general sums with the associated injectors ($@^n_m$ injects the value as the mth element of a sum of n).

2.3. Semantics

As this article only aims at presenting a concept, the details of semantics are not crucial at this stage. In theory, this language is designed to be mappable on any existing language by changing its user-interface grammar and adding some front-end analysis to distinguish recursive arguments from other kinds of arguments.

3. Previous Work

3.1. Deforestation as an optimization

As stated in the introduction, the primary motivation for trying to automatically remove intermediate data structures from functional languages is to improve their execution in terms of time and space by removing unnecessary memory overhead. This is the motivation of Wadler's original article ([5]). Many subsequent articles on deforestation were written, improving on Wadler's initial ideas and leading to the integration of a derived version of deforestation in the *Glasgow Haskell Compiler* ([2]).

3.1.1 General deforestation: introduction.

Theorem 3.1 ***Deforestation Theorem.*** *Every composition of functions with treeless definitions can be effectively transformed to a single function with a treeless definition without loss of efficiency.*

In [5], with the theorem reproduced here as theorem 3.1, Philip Wadler proves that one can determine whether deforestation can be applied successfully to a program. To make the deforesting transformation effective, he also defines the seven transformation rules given in figure 2(these rewriting rules are given in the language defined in section 2, as is every other code sample in this article).

We can give a quick interpretation of these rules as follow :

1. The treeless form of a variable is a variable.

2. We build the treeless form of a constructor application by applying the constructor to the treeless forms of its arguments. This rule and rule (4) are exactly where the deforestation work is done, the others being used only to make the constructors explicit.

3. This rule expands function applications by inlining their body. This allows return data structures to be deforested, which would not be possible if the function were seen as a black box. However, recursive functions are expanded only once, and their reappearance allows to define their treeless form.

4. The treeless form of a `case` expression over a variable is the same case with branches taken in their treeless form.

(1) $T\,[\![v]\!] = v$

(2) $T\,[\![@^n_m(t_1, \ldots, t_n)]\!] = @^n_m(T\,[\![t_1]\!], \ldots, T\,[\![t_n]\!])$

(3) $T\,[\![f\ v\ [vA]\ [vC]]\!] = T\,[\![t[t_1/v, \ldots]]\!]$
where f is defined
by $f\ v\ [vA]\ [vC] = t$

(4) $T\,[\![\texttt{case}\ v\ \texttt{of}\ p_1 \rightarrow t_1 \mid \ldots \mid p_n \rightarrow t_n]\!]$
$= \texttt{case}\ v\ \texttt{of}$
$\quad p_1 \rightarrow T\,[\![t_1]\!]$
$\quad \mid p_n \rightarrow T\,[\![t_n]\!]$

(5) $T\,[\![\texttt{case}\ @^n_m(t_1, \ldots, t_n)\ \texttt{of}\ p_1 \rightarrow t'_1 \mid \ldots \mid p_n \rightarrow t'_n]\!]$
$= T\,[\![\, t'_i[t_1/v_1, \ldots, t_n/v_n]]\!]$
where $p_i = @^n_m(v_1, \ldots, v_k)$

(6) $T\,[\![\texttt{case}\ f\ t_1\ [vA]\ [vC]\ \texttt{of}\ p_1 \rightarrow t'_1 \mid \ldots \mid p_n \rightarrow t'_n]\!]$
$= \texttt{case}\ t[t_1/v_1, \ldots]\ \texttt{of}$
$\quad p_1 \rightarrow T\,[\![t_1]\!]$
$\quad \mid p_n \rightarrow T\,[\![t_n]\!]$

(7) $T[\![\texttt{case}$
$\quad (\texttt{case}\ t_0\ \texttt{of}\ p_1 \rightarrow t_1 \mid \ldots \mid p_n \rightarrow t_n)$
$\quad \texttt{of}\ p'_1 \rightarrow t'_1 \mid \ldots \mid p'_n \rightarrow t'_n]\!]$
$= T[\![\texttt{case}\ t_0\ \texttt{of}$
$\quad p_1 \rightarrow (\texttt{case}\ t_1\ \texttt{of}\ p_1 \rightarrow t'_1)$
$\quad \mid \ldots$
$\quad \mid p_n \rightarrow (\texttt{case}\ t_n\ \texttt{of}\ p_n \rightarrow t'_n]\!]$

Figure 2. Transformation rules for deforestation.

5. When the selector of a `case` expression is a constructor, the expression can be evaluated. It avoids the inspection of the heap object designated by the selector, thus removing that intermediate data structure.

6. This allows one to expand a function call when it is used as selector to a `case`. This is also a very important step in deforestation because it prevents us from unnecessarily inspecting the returned data.

7. This rule just allows us to distribute a `case` inside another `case`, which then allows one to apply one of the rules over `case` constructs that perform deforestation.

As stated, rules (3) and (6) can lead to an infinite unrolling of a recursive program. And so, by themselves, these rules don't terminate. Thus, the algorithm also checks for reappearance of any already seen pattern when it applies one of the two fore mentioned rules. When we find a call to a function (let us call it f) we have already encountered in the previous steps, we introduce a new treeless function (let us call it h) that satisfies : $h\ v_1\ \ldots\ v_n = T[\![f\ v_1\ \ldots\ v_n]\!]$ The Deforestation Theorem tells us there exists at least one such function if all functions involved, except the one we're trying to deforest, is *treeless*. This restriction is crucial in ensuring termination in Wadler's algorithm, but also significantly restricts the legal inputs to the algorithm.

3.1.2 Removing intermediate lists: (Short-Cut) Fusion

Subsequent to that article, many attempts were made to try and make the algorithm more general, or safer, but only some of these attempts actually led to an implementation. One of them was *short-cut fusion*. Fusion is the restriction of *deforestation* to intermediate lists, which are the most commonly used "glue" in functional programs.*Short-cut fusion* is an algorithm that performs *fusion* on any well-formed program, but does not remove every intermediate list and needs a lot more theoretical work to adapt it to general data structures. It was introduced in [3].

The algorithm in fact rewrites list-consuming functions using the *foldr* function and list-producing functions using the *build* function, then applying the *foldr/build* rule shown on figure 3 to their compositions.

Although [3] only considers intermediate lists, the *foldr/build* rule can be generalized to other inductive datatypes. Short-cut fusion has been implemented in the *Glasgow Haskell Compiler*, being one of the first deforestation algorithms to be put into practice. This allowed for a lot of feedback on the use and efficiency of the methods, as well as on the issues such an implementation raised (see [2]).

```
foldr xs [] [k;z] =
   case xs of
       Nil          → z
     | Cons (x,xs) →
              k x (foldr xs k z)


build g = g Cons Nil


foldr (build g) [] [k;z] = g k z
```

Figure 3. *foldr*, *build*, **and the** *foldr/build* **rule.**

3.2. Deforestation as cut-elimination: a link to proof theory

In [1], Wadler's technique for deforesting by trying to tie knots was linked to cut-elimination of the program, regarded as a proof. This reinforced the link between proofs and programs, and allowed for a better understanding of what was behind deforestation, as well as a more theoretical, but still down-to-earth deforestation algorithm.

3.2.1 The Curry-Howard correspondence: proofs as programs

The most important link between proof-theory and programs was achieved when Curry designed his simply-typed lambda-calculus. This link, known as the Curry-Howard correspondence, is rather obvious when comparing the rules of propositional logic presented in figure 4 and the type-checking rules for the simply-typed lambda-calculus shown on figure 5. We can see that types in the program world correspond to propositions in the proof world, and that programs map on proofs in the same way.

Moreover, to paste several proofs together, one uses the CUT rule. The cut-elimination procedure then tells one that one can normalize the global proof by removing the cuts, thus removing the interface between its different components and allowing them to interact more freely with each other. The above mentioned analogy between proofs and programs holds, here again: by inlining the function calls in a program, we remove the strict interface between two different components of the program and allow for a better interaction between them.

3.2.2 Deforesting by eliminating cuts

The method used in [1] is fairly similar to Wadler's for it is based on the unrolling of programs and function calls (eliminating cuts, still), but it differs in the way it does it. If the Deforestation Algorithm presented in [5] unroll innermost

$$\frac{}{\Gamma, \phi \vdash \phi}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \qquad \frac{\Gamma \vdash \phi \to \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}$$

$$\frac{\Gamma \vdash \phi \vee \psi \quad \Gamma, \phi \vdash \chi \quad \Gamma, \psi \vdash \chi}{\Gamma \vdash \chi}$$

Figure 4. The rules of propositional logic

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash Mx : \tau}$$

$$\frac{\Gamma \vdash x : \sigma_1 \quad \Gamma \vdash y : \sigma_2}{\Gamma \vdash (x, y) : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash (x, y) : \sigma_1 \times \sigma_2}{\Gamma \vdash x : \sigma_1} \qquad \frac{\Gamma \vdash (x, y) : \sigma_1 \times \sigma_2}{\Gamma \vdash y : \sigma_2}$$

$$\frac{\Gamma \vdash x : \sigma_1}{\Gamma \vdash \text{inl}\ x : \sigma_1 + \sigma_2} \qquad \frac{\Gamma \vdash y : \sigma_2}{\Gamma \vdash \text{inr}\ y : \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash M : \sigma_1 + \sigma_2 \quad \Gamma, x : \sigma_1 \vdash M_1 : \rho \quad \Gamma, y : \sigma_2 \vdash M_2 : \rho}{\Gamma \vdash \text{case}\ M\ \text{of}\ x \to M_1 | y \to M_2 : \rho}$$

Figure 5. The rules of simple type-inference

calls first, the proof theory based method only considers the first *active sequence*, that is, the first function call with an *active element* (a function call, a case or a recursive value) as its recursive argument and tries to resolve *demand* on those elements instead of unrolling them to reveal the internal demands and then solving them. Of course, such a method is restricted to functions that have only one recursive argument, but in practice, using mutual recursion, we can often write multi-argument functions as a set of single-argument ones. Moreover, considering that particular class of functions (functions with only one recursive argument) allows us to distinguish argument types between recursive, accumulating and context arguments. This piece of information is very useful to know *where* to apply the transformations first, as we only follow *active sequences*. As [1] goes into further details about proof theory, circular proofs and the transformation system -which is at the heart of CHOP and will be shown later on, the interested reader is invited to read it. This article will only show a few instances of the system applied to some example programs, to compare its way of working and its performances with the others.

3.3. Deforestation as composition of tree transducers

But even that link to proof theory has its drawbacks. The main one is that the transformations it describes are very high-level and we have nothing that allows us to understand what's going on concretely at the level of programming language processing during the transformation. And, even if it's not the first motivation of that paper, [4] provides a transformation system that describes the low-level interactions of function calls, and a high-level framework that makes proving the correctness of the algorithm more elegant and easy using tree transducers.

More exactly, it uses tree transducers, and the properties of their composition, to describe functional programs, and their composition, which corresponds to what Wadler called *deforestation*.

3.3.1 What is a tree transducer ?

Disclaimer The reader has here to be warned: this paper presents only a light version of what a tree transducer can be. Their real power needs some category theory voodoo to be fully understood and this is beyond the scope of this paper. So here will be presented only the so-called *top-down tree transducers*, which are powerful enough to describe some single argument programs. Although *top-down tree transducers* are quite restrictive, there exist other classes of tree transducers that do the same job with respect to other, more general, classes of programs. Jürgensen's main

achievement is, in [4], that he manages to define a more abstract class of transducers to unify the proofs and notations into a single framework: monadic transducers; they will not be treated here.

Top-down tree transducers

Definition 3.1 ***Ranked Alphabet.*** *A ranked alphabet Σ is a set of symbols annotated with their arities.*

$\Sigma^{(k)}$ denotes the subset of Σ that holds symbols of arity k.

Definition 3.2 ***Top-Down Tree Transducer.*** *A top-down tree transducer is a tuple $T = (Q, \Sigma, \Delta, q_0, R)$ where Q is the set of* states*, Σ is the* input *ranked* alphabet *, Δ is the* output *ranked* alphabet *, $q_0 \in Q$ is the* initial state*, and R is the* rule *. We have $R \subset \bigcup_{k\in\mathbb{N}_0} Q(\Sigma^{(k)}X_k) \times T_\Delta(QX_k)$ such that:*

$$\forall q \in Q.\forall k \in \mathbb{N}_0.\forall \sigma \in \Sigma^{(k)}.$$
$$\exists! rhs_{R,\sigma}\ q \in T_\Delta(QX_k).(q(\sigma(x_1, \ldots, x_k)), rhs_{R,\sigma}\ q) \in R$$

Note that this definition looks very much like the definition of a finite deterministic automaton...

3.3.2 Programs as tree transducers

Of course, having a definition does not tell us how to represent programs using tree transducers. Once again, top-down tree transducers can only be used to represent very simple programs.

The *rule*'s form tells us how tree transducers and programs are related: the *states* represent the *functions*, the *input alphabet* [resp. *output alphabet*] represents the *input type* [resp. *output type*], the *initial state* represents the program's *entry point*, and the *rule* represent the *computation*.

The top-down tree transducer shown in figure 6 represents the `zigzag` program, taken from [4], and whose text can be found in example 8. Figure 7 shows another example, that needs a little more work to be transformed into a top-down tree transducer: the `tail` function.

3.3.3 Composing tree transducers

Having represented (simple) programs as (top-down) tree transducers, we now have to be able to compose them. Here are presented the methods and theorems for top-down tree transducers, again, as monadic transducers are more complex.

Definition 3.3 *Let $T_1 = (P, \Sigma, \Delta, p_0, R_1)$ and $T_2 = (Q, \Delta, \Gamma, q_0, R_2)$ be top-down tree transducers. Let T_2' be the tree transducer defined by: $T_2' = (Q, \Delta', \Gamma', q_0, R_2')$ where*

$$T_{zigzag} = (Q_{zigzag}, \Sigma_{zigzag}, \Delta_{zigzag}, q_{zigzag}, R_{zigzag})$$

where
$Q_{zigzag} = \{zig, zag\}$
$\Sigma_{zigzag} = \{\alpha^{(0)}, \sigma^{(2)}\}$
$\Delta_{zigzag} = \{N^{(0)}, A^{(1)}, B^{(1)}\}$
$q_{zigzag} = zig$

$$R_{zigzag} = \left\{ \begin{array}{lcl} zig\ \alpha & \mapsto & N, \\ zag\ \alpha & \mapsto & N, \\ zig\ \sigma(x_1, x_2) & \mapsto & A(zag\ x_1), \\ zag\ \sigma(x_1, x_2) & \mapsto & A(zig\ x_2) \end{array} \right\}$$

Figure 6. An example top-down tree transducer.

$$T_{tail} = (Q_{tail}, \Sigma_{tail}, \Delta_{tail}, q_{tail}, R_{tail})$$

where
$Q_{tail} = \{tail, id\}$
$\Sigma_{tail} = \{a^{(0)}, nil^{(0)}, cons^{(2)}\}$
$\Delta_{tail} = \{a^{(0)}, nil^{(0)}, cons^{(2)}\}$
$q_{tail} = tail$

$$R_{tail} = \left\{ \begin{array}{lcl} id\ a & \mapsto & a, \\ id\ nil & \mapsto & nil, \\ id\ cons(x_1, x_2) & \mapsto & cons(id(x_1), id(x_2)), \\ tail\ a & \mapsto & a, \\ tail\ nil & \mapsto & nil, \\ tail\ cons(x_1, x_2) & \mapsto & id(x_2), \end{array} \right\}$$

Figure 7. Another example top-down tree transducer.

$\Delta' = \Delta \uplus \{px^{(0)}\}_{p\in P, x\in X_r}$
$\Gamma' = \Gamma \uplus \{((q,p)x)^{(0)}\}_{q\in Q, p\in P, x\in X_r}$
$R_2' = R_2 \uplus \{q(px) \mapsto (q,p)x\}_{q\in Q, p\in P, x\in X_r}$
and
$r = \max_{\sigma\in\Sigma}(rank_\Sigma \sigma)$

then we define the ***syntactic composition*** *$T_2 \cdot T_1$ of T_2 and T_1 as the tree transducer defined by:*

$T_2 \cdot T_1 = (Q \times P, \Sigma, \Gamma, (q_0, p_0), R)$
where
$R = \{(q,p)(\sigma(x_1, \ldots, x_r)) \mapsto [\![T_2']\!]_q (rhs_{R_1,\sigma} p)\}_{q\in Q, p\in P, \sigma\in\Sigma}$
and $r = rank_\Sigma \sigma$

where $[\![T]\!]_q\ x$ is the ***computed tree transformation*** *of T starting from q, as defined in [4] (it is the result of the application of T to x in state q).*

This definition is rather complicated, and here is a short analysis of it. The first step to perform the syntactic composition is to modify T_2. This modification makes it able to consume T_1's half complete output and then run T_2 and T_1 on it *at the same time*. This clearly corresponds to the idea of deforestation, as it allows one to consume the elements of a tree as they are produced and not after the entire tree was built.

Moreover, [4] proves the following theorem :

Theorem 3.2 *The syntactic composition of two top-down tree transducers is a top-down tree transducer.*

This is similar to Wadler's *deforestation theorem* (3.1).

3.4. Examples

We now have defined four different kinds of transformations on programs. Let's compare them on some interesting examples, keeping in mind their limits:

- input to Wadler's **deforestation algorithm** has to be in *treeless form*, and the result is guaranteed to be an improvement;
- short-cut fusion can only work on intermediate lists and does not guarantee any improvement on its input. In order to have effective examples, we need to be able to rewrite our examples using `foldr` and `build`;
- Cockett's **cut-elimination** considers only programs with one, and only one *active argument*, and does not guarantee improvement but searches for improvements with no guarantee of reaching a fully deforested program, except when Wadler's algorithm, tree transducer composition or short-cut fusion do;

```
data tree =   Alpha
            | Sigma of (tree*tree)
data list = N | A of list | B of list

letrec zig t =
  case t of
      Alpha          -> N
    | Sigma(t1,t2)   -> A(zag t1)
letrec zag t =
  case t of
      Alpha          -> N
    | Sigma(t1,t2)   -> B(zig t2)

letrec bin l =
  case l of
      N     -> Alpha
    | A(l)  -> Sigma(bin l, bin l)
    | B(l)  -> Sigma(bin l, bin l)

let binzig t = bin (zig t)
```

Figure 8. The `zigzag`, `bin` and `binzig` programs.

- **syntactic composition of top-down tree transducers** considers only some simple single-argument programs, but guarantees termination and deforestation on them.

Here is a table summarizing the following examples and which algorithms are applied to them:

Program	WDA	SCF	CE	TDTT	CHOP
`binzig`				✓	✓
`append`	✓	✓	✓		✓
`rev`		✓	✓		✓

3.4.1 `binzig`

The example program shown on figure 8 is taken from [4]. It illustrates the use of intermediate lists in a single argument program. It is only used with tree transducers because short-cut fusion gives the same result as tree transducers (see [4]), cut-elimination gives the same result as CHOP, and the `bin` function is not in a treeless form and thus can't be deforested using Wadler's algorithm.

Tree transducers First, we translate the program into tree transducers. We must consider two top-down tree transducers : T_{zigzag}, as defined in figure 6, and which defines both `zig` and `zag`, and T_{bin}, defined in figure 9.

$T_{bin} = (Q_{bin}, \Sigma_{bin}, \Delta_{bin}, q_{bin}, R_{bin})$

where
$Q_{bin} = \{bin\}$
$\Sigma_{bin} = \{N^{(0)}, A^{(1)}, B^{(1)}\} = \Delta_{zigzag}$
$\Delta_{bin} = \{\alpha^{(0)}, \sigma^{(2)}\} \quad = Sigma_{zigzag}$
$q_{bin} = bin$

$$R_{bin} = \left\{ \begin{array}{lcl} bin\ N & \mapsto & \alpha, \\ bin\ A(x_1) & \mapsto & \sigma(bin\ x_1, bin\ x_1), \\ bin\ B(x_1) & \mapsto & \sigma(bin\ x_1, bin\ x_1) \end{array} \right\}$$

Figure 9. Tree transducer for `bin`.

We now transform T_{bin} into T'_{bin} according to definition 3.3: $T'_{bin} = (Q_{bin}, \Sigma'_{bin}, \Delta'_{bin}, q_{bin}, R'_{bin})$

where
$\Sigma'_{bin} = \{N^{(0)}, A^{(1)}, B^{(1)}, (zig\ x_1)^{(0)}, (zag\ x_1)^{(0)}\}$
$\Delta'_{bin} = \{\alpha^{(0)}, \sigma^{(2)}, (bin, zig)(x_1)^{(0)}, (bin, zag)(x_1)^{(0)}\}$

$$R_{bin} = \left\{ \begin{array}{lcl} bin\ N & \mapsto & \alpha, \\ bin\ A(x_1) & \mapsto & \sigma(bin\ x_1, bin\ x_1), \\ bin\ B(x_1) & \mapsto & \sigma(bin\ x_1, bin\ x_1) \\ bin\ (zig x_1) & \mapsto & (bin, zig)(x_1) \\ bin\ (zag x_1) & \mapsto & (bin, zag)(x_1) \end{array} \right\}$$

and we get the syntactic composition:
$T_{bin} \cdot T_{zigzag}$
$= (Q_{bin} \times Q_{zigzag}, \Sigma_{zigzag}, \Sigma_{zigzag}, (bin, zig), R)$ where

$$R = \left\{ \begin{array}{lcl} (bin, zig)\ \alpha & \mapsto & \alpha, \\ (bin, zag)\ \alpha & \mapsto & \alpha, \\ (bin, zig)\ \sigma(x_1, x_2) & \mapsto & \sigma((bin, zag)x_1, \\ & & \quad (bin, zag)x_1), \\ (bin, zag)\ \sigma(x_1, x_2) & \mapsto & \sigma((bin, zig)x_2, \\ & & \quad (bin, zig)x_2) \end{array} \right\}$$

which yields the following program:

```
letrec binzig t =
  case t of
      Alpha          -> Alpha
    | Sigma(t1,t2)   -> Sigma(binzag t1,
                              binzag t1)
letrec binzag t =
  case t of
      Alpha          -> Alpha
    | Sigma(t1,t2)   -> Sigma(binzig t2,
                              binzig t2)
```

in which the intermediate list is obviously never built and inspected.

3.4.2 Associativity of `append`

The `append` function appends two lists together. It is recursive on its first list argument, which makes calls

```
data list a =
       Nil
     | Cons of (a*list a)

letrec append x [] [y] =
  case x of
      Nil          -> y
    | Cons(x,xs)   ->
         Cons(x,(append xs y))
```

Figure 10. The `append` function.

to `append (append •1 •2) •3` traverse the first list twice unnecessarily. As, in real languages we might not be able to rewrite it in a more efficient way (for different reasons, like an incomplete call, or a function returning a function...), it is useful, and can speed things up, to transform that form into the equivalent `append •1 (append •2 •3)`. In other words, it is good to be able to prove the associativity of `append`.

Wadler's deforestation algorithm. We consider the code in figure 10 and we want to compute `T⟦append (append x y) z⟧`.

```
T[append (append x y) z]
 = case (append x y) of
      Nil          -> z
    | Cons(a,as)   -> Cons(a,append as z)
 = case (
    case x of
       Nil          -> y
     | Cons(x,xs)   ->
          Cons(x,append xs y)
  of
     Nil          -> z
   | Cons(a,as)   -> Cons(a,append as z)
 = case x of
      Nil          ->
        case y of
           Nil          -> z
         | Cons(a,as)   ->
             Cons(a,append as z)
    | Cons(x,xs)   ->
        case (Cons(x,append xs y)) of
           Nil          -> z
         | Cons(a,as)   ->
             Cons(a,append as z)
 = case x of
      Nil          ->
        case y of
           Nil          -> z
         | Cons(a,as)   ->
             Cons(a,append as z)
    | Cons(x,xs)   ->
        Cons(x,append (append xs y) z)
```

We now introduce a new function `f x y z` such that `f x y z = T⟦append (append x y) z⟧`.

```
T[append (append x y) z] =
  f x y z

letrec  f x y z =
  case x of
     Nil          ->
       case y of
          Nil          -> z
        | Cons(a,as)   ->
            Cons(a,append as z)
    | Cons(x,xs)   -> Cons(x,f xs y z)
```

This is the final form of the solution using this algorithm. We will see that distinguishing recursive arguments from context arguments allows a much cleaner form for the final solution:

Cut-elimination. We consider the active sequence `append (·append x y) z` and try to deforest it. We first unroll the inner call to satisfy the demand:

```
append (.append x y) z
 = append (
     case x of
        Nil          -> y
      | Cons(x,xs)   ->
         Cons(x,(append xs y))
   ) z
 = case x of
      Nil          -> append y z
    | Cons(x,xs)   ->
       append (Cons(x,(append xs y)))
              z
 = case x of
      Nil          -> append y z
    | Cons(x,xs)   ->
       Cons(x, append (append xs y) z)
```

We now *tie the knot*, replacing instances of `append (append)` with a function `f x [] [u=append y z]`:

```
letrec f x [] [u] =
  case x of
      Nil          -> u
    | Cons(x,xs)   -> Cons(x,appapp xs u)
```

We recognize that `f = append` and we just proved the associativity of `append`.

Short-cut fusion Before applying short-cut fusion to `append (append • •) •`, we need to rewrite `append` using `foldr` and `build` as defined in figure 3.

```
letrec append x y =
  build
   (\c n ->
     foldr x c (foldr y c n))

append (append x y) z =
  build
   (\c n ->
     foldr
       (build
         (\c' n' ->
           foldr x c' (foldr y c' n')))
     c (foldr z c n))
```

Then we can apply the *foldr/build* rule to deforest:

```
append (append x y) z
 =  build
     (\c n ->
      (\c' n' ->
        foldr x c' (foldr y c' n'))
      c (foldr z c n))
 = foldr x (:) (foldr y (:)
                (foldr z (:) []))
```

We inlined `build` on the last line, and we could also inline `foldr`, but the result gets really ugly and never gets as close to the associativity of `append` as the cut-elimination method, even though we can see it "through the `foldrs`".

3.4.3 Improving `rev`

Another classic example is improving the `rev` function presented in figure 11 into the one presented in figure 12. This optimization does actually save space AND time, but it introduces the use of an accumulating parameter that makes things more complicated as far as analyzing the function is concerned. Yet, this example shows why Wadler's *treeless form* lacks generality, as the definition given in figure 11 is not treeless (and neither is the one given in figure 12, which tells us not to try and run the Deforestation Algorithm here to avoid non-termination. Once again, the fact

```
letrec rev x =
  case x of
      Nil         -> Nil
    | Cons(x,xs) ->
        append (rev xs) (Cons(x,Nil))
```

where `append` is defined in figure 10.

Figure 11. A naive implementation of `rev`.

```
letrec rev' x [y] [] =
  case x of
      Nil         -> y
    | Cons(x,xs) -> rev' xs (Cons(x,y))

letrec rev x = rev' x []
```

where `append` is defined in figure 10.

Figure 12. An efficient implementation of `rev`.

that it does not make any difference between *active arguments* and other kinds of arguments makes it fail to perform a simple, yet very efficient optimization. We can't work with top-down tree transducers either because the definition in figure 11 uses `append`, which has two arguments.

Cut-elimination The definition in figure 11 has internal demand and we have to deforest `append (·rev x) z` to satisfy it.

```
append (.rev x) z
 = append
    (.case x of
         Nil         -> Nil
       | Cons(x,xs) ->
           append (rev xs)
                  (Cons(x,Nil)))
    z
 = case x of
       Nil         -> append Nil z
     | Cons(x,xs) ->
         append
           (.append (rev xs)
                    (Cons(x,Nil)))
           z
 = case x of
       Nil         -> z
     | Cons(x,xs) ->
         append (.rev xs)
                (append (Cons(x,Nil))
                        z)
 = case x of
       Nil         -> z
     | Cons(x,xs) ->
         append (.rev xs) (Cons(x,z))
```

where we recompute last result when we get to the active sequence `append (·append x y) z`. The last expression shows that we can tie the knot and write:

```
letrec apprev x [] [y] =
  case x of
      Nil         -> y
    | Cons(x,xs) ->
        apprev xs (Cons(x,y))

letrec rev x =
  case x of
      Nil         -> Nil
    | Cons(x,xs) ->
        apprev xs (Cons(x,Nil))
```

as we can't, in this system, make a difference between context arguments and accumulator arguments. Yet, this definition is pretty close to the one we were expecting, and certainly is as efficient, apart from one extra function call.

Short-cut fusion We first have to rewrite the definition of `rev` using `build` and `foldr`.

```
letrec rev x =
  build
   (fun c n ->
      foldr x
            (fun x revxs ->
               build
                 (fun c n ->
                   foldr revxs
                         c
                         (c x n)))
            n)
```

Even though we now have a nice expression of `rev`, the transformation cannot be applied to it because we never run into an instance of `foldr (build •) • •`. This is quite expected since there is no way of rewriting the efficient definition of `rev` (figure 12) using `foldr` and `build` because of the accumulator. This is one of the most serious limitations of this transformation, as accumulating parameters are often used in functional programs, since they often allow *tail-recursion*.

4. The CHOP system

As the examples in section 3.4 show, neither of the above mentioned systems can be applied to every program we expect to be able to deforest. Limitations on the form of the input function, or even its type, make it hard to adapt them to more general cases, and yet, they are not different enough to be applied separately in order to take care of all the cases. Monadic transducers may be more generally applicable, but it is not clear how one chooses the monad appropriate to a given program.

4.1. Principles

Among the above mentioned systems, and apart from monadic tree transducers, the most general is Cockett's cut-elimination based program transformations (section 3.2). Thus, we chose to base CHOP on that system and to improve on it. We do so in two major steps:

- we first want to allow the distinction between accumulating arguments and context arguments;
- and then address problems inherent to the system, especially concerning mutual recursion and functions with more than one recursive argument (`zip`-like or worse...).

As mentioned before, CHOP is based on the system presented in [1], and it especially uses the ideas of *active sequences* and *knots*.

We define an *active sequence* as follows:

Definition 4.1 *An **active sequence** in a program is one of the following:*

1. *A function application where the active argument is a function application.*
2. *A function application where the active argument is a `case` construct.*
3. *A function application where the active argument is an active constructor (a constructor applied to a `case` or a function application).*
4. *A `case` construct where the selector is a function application.*
5. *A `case` construct where the selector is a `case` construct.*
6. *A `case` construct where the selector is a constructor application.*

At all point during the computation, we have a *history* holding already encountered *active sequences*, with the name of their arguments if they have not yet been tied in a knot, or the corresponding call to the replacing function when it is available.

During the computation, each variable that has a recursive type is assigned a *genealogy*, holding all its predecessors (and as we have control on the variable names appearing during the computation, we make sure they are unique -this may not be true on the examples shown in section 4.3.1).

The algorithm goes on as follows:

1. we look for the first (outermost) *active sequence* in the program we want to deforest. If, according to definition 4.1, it is of form:
   - 1, we check the *history* for an instance of it, with "parent arguments" (*i.e.* the arguments of the in-program instance are descendants of the arguments of the history instance).
     - If there exists one and it is solved, we replace the instance we found with the solution `f` held in the history. We then try to *roll-up* the program by trying to match the function we are in with `f` (it is a quite simple instance of the unification problem on the syntax trees). We finally go back to step 1.
     - If there exists one and it is not solved, go to step 2.
     - Otherwise, we add it to the *history*, instantiating the arguments with fresh variable names. We then consider it the new deforestation job and unroll the inner call to satisfy demand, going back to step 1;
   - 4 or 3, we unroll the function call and go back to step 1;
   - 6, we select the correct branch and rename the bound variables in it;
   - 2 or 5, we push the outermost *active element* inside the `case` branches, and go back to step 1. Note that we now have to be careful to make sure we reduce all branches before going on.
2. we now have a solution for the *active sequence* `f·g`, and we tie the knot by performing the following actions:
   - we introduce a new function `f` such that
     $$\texttt{f}\ \ \texttt{x}\ \ (\texttt{ac}_i)_{1\le i\le n}\ \ (\texttt{co}_i)_{1\le i\le m} = \texttt{f}\ (\texttt{g}\ \texttt{x}\ (\texttt{ac}_i)_{1\le i\le n_1}\ (\texttt{co}_i)_{1\le i\le m_1})\ (\texttt{ac}_i)_{n_1+1\le i\le n}\ \ (\texttt{co}_i)_{m_1+1\le i\le m}$$

It is easily defined with the code we unrolled between the first occurrence of the *active sequence* and the one we are currently treating.

We have to be careful when this occurs in a `case` branch, as we also have to replace instances of `f·g` in other branches, if applicable.

Once the function `f` is defined, we can affect it to `f·g` in the *history* . The knot is now tied and what we now want and can do is *tighten* it.

- in the definition of `f`, we check for any context argument (of `f`) that is built upon using a constructor or used by an internal call as an accumulating argument. If that happens, then this argument is in fact an accumulating argument. We then have to modify the definition of `f` to take this new piece of information into account.
- in the definition of `f`, we check for any function calls where all arguments are context arguments for `f`. Then this entire function call can be made context and any of its arguments that don't appear anywhere else in `f` can be removed from the definition of `f`.
- we are now done tightening the knot and we go back to the first deforestation job in queue (the one that made this knot appear), starting at step 1.

We are done when we can't find any active sequence and every item in the *history* is solved.

4.2. First Example

To explain more clearly what happens during a run of the above defined algorithm, here it is, run on the function defined in figure 13.

The first active sequence we encounter is `append (collect t) y` (the second argument is context and can be ignored in a first approach). We represent variable genealogy as a list attached to the variable, and start with the following history:

Active Sequence	#ac,#co	Syntactic Composition
`append·collect`	0,1	$\emptyset$

```
append (collect t) y
 = append
     (case t of
          Leaf a         -> Cons(a,Nil)
        | Node(t1.[t],t2.[t]) ->
             append (collect t1)
                    (collect t2))
     y
 = case t of
       Leaf a       ->
           append Cons(a,Nil) y
```

```
data tree a =
     Leaf of a
   | Node of (tree a*tree a)
data list a =
     Nil
   | Cons of (a*list a)

letrec collect t =
  case t of
       Leaf a       -> Cons(a,Nil)
     | Node(t1,t2) ->
          append (collect t1)
                 (collect t2)
```

where `append` is defined in figure 10.

Figure 13. The `collect` function.

```
      | Node(t1.[t],t2.[t]) ->
           append (append (collect t1)
                          (collect t2))
                  y
 = case t of
       Leaf a       -> Cons(a,y)
     | Node(t1.[t],t2.[t]) ->
           append (append (collect t1)
                          (collect t2))
                  y
```

At this point, the outermost active sequence this program holds is `append·append`, which is not yet in our history. This becomes the current deforestation job and is added to the history. It is then solved (this computation is done in section 4.3.1), and the result is put into the history as well

Active Sequence	#ac,#co	Syntactic Composition
`append·collect`	0,1	$\emptyset$
`append·append`	0,2	`append x (append y z)`

We now replace in the program:

```
append (collect t) y
 = case t of
       Leaf a       -> Cons(a,y)
     | Node(t1.[t],t2.[t]) ->
           append (collect t1)
                  (append (collect t2)
                          y)
```

We tie the knot as follows:

```
letrec appcol t [] [y] =
  case t of
      Leaf a       -> Cons(a,y)
    | Node(t1,t2) ->
         appcol t1
                (append (collect t2)
                        y)
```

And now comes the tightening phase. We first check for uses of context arguments as accumulators. We notice that `y` is one such argument and we immediately make it an accumulator. We can't do anymore tightening, we can add the new definition of `appcol` to the history and go back to the initial computation.

Active Sequence	#ac,#co	Syntactic Composition
`append·collect`	0,1	`appcol x y`
`append·append`	0,2	`append x (append y z)`

```
letrec collect t =
  case t of
      Leaf a       -> Cons(a,Nil)
    | Node(t1,t2) ->
         appcol t1 (collect t2)
```

We recognize in the definition of `collect` an instance of `appcol` with `y = Nil` (this is not actually given by any simple unification algorithm, but we use it only to show an example of rolling-up). We can then *roll-up* this definition and finally get:

```
letrec collect t = appcol t []
```

4.3. Achievements, issues and solutions

4.3.1 Achievements

To show what this system achieves, we'll just highlight the results of the algorithm on the examples studied in section 3.4.

The `binzig` function. We first run the algorithm on the `binzig` function, defined in figure 8. Our first active sequence is `bin·zig`.

Active Sequence	#ac,#co	Syntactic Composition
`bin·zig`	0,0	$\emptyset$

```
bin (zig t)
 = bin
    (case t of
         Alpha        -> N
       | Sigma(t1,t2) ->
            A(zag t1))
 = case t of
       Alpha        -> bin N
     | Sigma(t1,t2) ->
          bin (A(zag t1))
 = case t of
       Alpha        -> Alpha
     | Sigma(t1,t2) ->
         case (A(zag t1)) of
             N    -> Alpha
           | A(l) ->
             Sigma(bin l, bin l)
           | B(l) ->
             Sigma(bin l, bin l)
 = case t of
       Alpha        -> Alpha
     | Sigma(t1,t2) ->
          Sigma(bin (zag t1),
                bin (zag t1))
```

Active Sequence	#ac,#co	Syntactic Composition
`bin·zig`	0,0	$\emptyset$
`bin·zag`	0,0	$\emptyset$

```
bin (zag t)
 = bin
    (case t of
         Alpha        -> N
       | Sigma(t1,t2) ->
            B(zig t2))
 = case t of
       Alpha        -> bin N
     | Sigma(t1,t2) ->
          bin (B(zig t2))
 = case t of
       Alpha        -> Alpha
     | Sigma(t1,t2) ->
         case (B(zig t2)) of
             N    -> Alpha
           | A(l) ->
             Sigma(bin l, bin l)
           | B(l) ->
             Sigma(bin l, bin l)
 = case t of
       Alpha        -> Alpha
     | Sigma(t1,t2) ->
          Sigma(bin (zig t2),
                bin (zig t2))
```

We can tie the knot by assuming that we have a solution for `bin·zag` and then solving it. We also have to take care

to only tighten this double knot after tying both of them.

```
letrec binzig t =
  case t of
      Alpha         -> Alpha
    | Sigma(t1,t2) ->
          Sigma(binzag t1,
                binzag t1)

letrec binzag t =
  case t of
      Alpha         -> Alpha
    | Sigma(t1,t2) ->
          Sigma(binzig t2,
                binzig t2)
```

The tightening does not perform any new transformation and we now have the given solution, which is deforested, even if the calls appear in the `Sigma` constructor, because they are part of the returned solution and never ever used in a `case` or a function call again.

Associativity of `append`. This example introduces nothing new during the actual computation, but is much more interesting to study during its tightening phase. Just after we tied the knot, we get the following:

```
letrec appapp x [] [y;z] =
  case x of
      Nil         -> append y z
    | Cons(x,xs) ->
        Cons(x,appapp xs y z)
```

We now tighten the knot step by step:

```
letrec appapp2 x [] [w=append y z] =
  case x of
      Nil         -> w
    | Cons(x,xs) ->
        Cons(x,appapp xs w)

letrec appapp x [] [w] =
    append x w

append (append x y) z =
  append x (append y z)
```

Which is the result we expected.

Improving `rev`. This example only shows features that have already been demonstrated in earlier examples, and thus, it is useless to unroll it here.

```
letrec f x =
  case x of
     Nil -> 0
   | Cons(x1,Cons(x2,xs)) ->
       x1 + x2 + f (g xs)

letrec g x =
  case x of
     Nil -> Nil
   | Cons(x,Nil) -> Cons(x,Cons(x,Nil))
   | _ -> x
```

Figure 14. An example of asynchronous mutual recursion.

4.3.2 Asynchronous mutual recursion

As with any of the other algorithms, we might run indefinitely if we have a set of mutually recursive functions that don't consume their arguments at the same speed, as the ones shown in figure 14.

The solution is quite simple, and consists in transforming every function in the set to simple-step functions by performing the transformation shown on figure 15.

4.3.3 Functions with two and more recursive arguments

The most commonly used multi-argument function that actually performs recursion on all its arguments is `zip` (see figure 16) and its kind. The code given here uses a slightly modified version of the language defined in section 2, and we only hope that the transformation we defined above works with such a language. Indeed, the transformation's rationale is based on functions that have only one recursive argument, but no assumption is made when performing the transformation itself. We would then have to check both arguments for *active sequences* and certainly would see the number of entries in the history rise a lot depending on the use that's done of the functions.

5. Conclusion

This article, after reviewing some of the existing deforestation systems, has introduced the CHOP system, whose goal is to provide both a practical and implementable deforestation algorithm, and a theoretical background clear enough to ensure certain properties of the algorithm, including termination and correctness on general programs with general datatypes. All proofs still have to be provided, or


```
letrec f x ... =
  case x of
     Ci((Cj...)...) -> M1
   | Ck((Cj...)...) -> M2
   |            ...
   | Ci((Cl x))     -> Mn
```

where `Ci...Cl` are constructors
gets transformed into the set of functions:

```
letrec f x ... =
  case x of
     Ci x -> g x ...
   | Ck x -> h x ...

letrec g x ... =
  case x of
     Cj x -> g1 x ...
   | Cl x -> Mn

letrec h x ... =
     Cj x -> h1 x ...

...
```

Figure 15. A solution to asynchronous mutual recursion.

```
letrec zip x y [] [] =
  case x of
      Nil        -> Nil
    | Cons(x,xs) ->
        case y of
            Nil         -> Nil
          | Cons(y,ys) ->
              Cons((x,y),(zip1 xs ys)
```

Figure 16. The `zip` function.

at least adapted from the proofs in [1], since CHOP is very close to the system presented in that article.

5.1. Future Work

Tree transducers deserve to be studied more, as they do provide an excellent background to check the correctness of the algorithm, by comparing the result to the expected result. They also provide a classification of programs regarding their ability to be deforested, which is not a small thing.

Another feature that would allow great optimizations it to consider coinductive datatypes as well as inductive datatypes. This would hopefully, among other things allow the clean deforestation of programs such as `rev·rev`.

We might also want to expand CHOP more cleanly to take care of multi-argument functions such as `zip`.

References

[1] R. Cockett. Deforestation, program transformation, and cut-elimination. *Electronic Notes in Theoretical Computer Science*, 47:1–40, 2001.

[2] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1996.

[3] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. *FPCA*, pages 223–232, 1993.

[4] C. Jürgensen. *Categorical semantics and composition of tree transducers*. PhD thesis, Technischen Universität Dresden - Fakultät Informatik, 2003.

[5] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

Appendix C

Tree Transducers: a more complete study

C.1 The Modular Family

Mostly studied in Dresden, Germany, by Joost Engelfriet and his students, these tree transducers are very close to a functional programming language syntax and more or less correspond to the intuitive idea one may have of recursive functions.

C.1.1 Top-down tree transducers

Top-down tree transducers were introduced by Rounds in 1968 ([Rou68], not available). Top-down tree transducers were defined again by Engelfriet in 1975 ([EFV01]). The definition given here is the one that is given in [Jür03], and only considers total-deterministic top-down tree transducers.

Definition 11 *A top-down tree transducer is a tuple* $T = (Q, \Sigma, \Delta, q_0, R)$ *where* Q *is a finite set,* Σ *and* Δ *are ranked alphabets,* $q_0 \in Q$ *and* $R \subset Q(\Sigma(X)) \times T_\Delta(QX)$ *such that, for all* $k \in \mathbb{N}, q \in Q, \sigma \in \Sigma^{(k)}$*, there is one and only one* $\mathrm{rhs}_{q,\sigma} \in T_\Delta(QX_k)$ *such that* $q(\sigma(x_1, \ldots, x_k)) \to \mathrm{rhs}_{q,\sigma}$ *is in* R.

The class of functions that are computed by a top-down tree transducer is called TOP.

Composition Results

TOP is stable under syntactic composition of tree transducers (as per [EFV01]). Thus, it is stable by function composition.

C.1.2 Macro tree transducers

Macro tree transducers were first introduced in 1982 by Courcelle, under the name *primitive recursion scheme*([CFZ82a, CFZ82b]). They were then redefined by Engelfriet and studied more thoroughly in [EV85], where the following definition was given (for total deterministic transducers, here again)

Definition 12 *A macro tree transducer is a tuple* $M = (Q, \Sigma, \Delta, q_0, R)$ *where* Q, Σ *and* Δ *are ranked alphabets (with* $Q^{(0)} = \emptyset$*),* $q_0 \in Q^{(1)}$ *and* R *is a finite set containing, for all* $q \in Q$, $\sigma \in \Sigma$, *exactly one rule of the form* $q(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_n) \to t$ *where* $m = \text{rank}_\Sigma(\sigma), n + 1 = \text{rank}_Q(q)$ *and* $t \in \text{RHS}(Q, \Delta, m, n)$

Engelfriet of course also gave the following definition for the right-hand sides.

Definition 13 *Let* Q *and* Δ *be ranked alphabets and* $m, n \geq 0$. *The set of right-hand sides* $\text{RHS}(Q, \Delta, m, n)$ *over* Q *and* Δ *with* m *variables and* n *parameters is the smallest set* $\text{rhs} \subseteq T_{Q \bigcup \Delta}(X_m \bigcup Y_n)$ *such that:*

1. $Y_n \subseteq \text{rhs}$
2. *For* $\delta \in \Delta^{(k)}$ *with* $k \geq 0$ *and* $\xi_1, \ldots, \xi_k \in \text{rhs}$, $\delta(\xi_1, \ldots, \xi_k) \in \text{rhs}$
3. *For* $q \in Q^{(k+1)}$ *with* $k \geq 0$, $x \in X_m$ *and* $\xi_1, \ldots, \xi_k \in \text{rhs}$, $q(x, \xi_1, \ldots, \xi_k) \in \text{rhs}$.

The class of functions computed by macro tree transducer is called MAC.

Alternate definitions

There are quite a certain number of alternate definitions, depending on the goal of the study. In particular, some people prefer to allow the initial state to have an arbitrary rank and make the initial parameters explicit (see[Jür03], for example).

Composition Results

Although their computational power is much more important than that of top-down tree transducers, macro attributed do not have nice composition properties. Many composition results have been proved for subclasses of them, though. Some of them are related to attributed tree transducers (see section C.2.1).

C.1.3 Modular tree transducers

Introduced and studied by Engelfriet in 1991 ([EV91]), these transducers can compute all primitive recursive tree functions.

Definition 14 *A modular tree transducer is a tuple* $M = (Q, \mathrm{mod}, \Delta, [\Sigma], q_0, R)$ *where* Q *and* Δ *are ranked alphabets,* mod *is a mapping* $Q \to \mathbb{N}*$, $q_0 \in Q$ *such that* $\mathrm{mod}(q) = 1$, $[\Sigma] = (\Sigma_1, \ldots, \Sigma_r)$ *with* $\Sigma_i \subseteq \Delta$ *and* $r = \mathrm{rank}_Q(q_0)$, *and* R *is a finite set such that, for all* $q \in Q^{(r+1)}$ *and for every* $\delta \in \Delta^{(k)}$, *there is exactly one rule of the form* $q(\delta(x_1, \ldots, x_k), y_1, \ldots, y_r) \to \xi$ *in* R, *with* $\xi \in T_{Q \bigcup \Delta}(X \bigcup Y)$ *such that:*

1. *for every* $p \in Q$ *occurring in* ξ, $\mathrm{mod}(p) \geq \mathrm{mod}(q)$.
2. *if* $p(\xi_1, \ldots, \xi_u)$ *occurs in* ξ *with* $p \in Q^{(u)}$, *and* $\mathrm{mod}(p) = \mathrm{mod}(q)$, *then* $\xi_1 \in \{x_1, \ldots, x_k\}$.

Definition 15 *We can also define the following terminology:*

- *a modular tree transducer is* m*-ary if* $\mathrm{rank}_Q(q_0) = m$.
- *a tree transducer is* n*-modular if, for every* $q \in Q$, $\mathrm{mod}(q) \leq n$.

In this framework, we can give simpler definitions.

- a macro tree transducer is a unary 1-modular tree transducer.
- a top-down tree transducer is a macro tree transducer such that, for all $q \in Q$, $\mathrm{rank}_Q(q) = 1$.

The class of modular tree transducers is called MOD. The subclass of n-modular tree transducers is called $n - MOD$. The subclass of n-ary modular tree transducer is called MOD_n.

Composition Results

The composition results for modular tree transducers, although not interesting for deforestation, are interesting by themselves. They form a strict composition hierarchy: $\forall n, m, n - MOD \subsetneq n - MOD^m \subsetneq (n+1) - MOD$. This allows us to say that most useful functions in reality are in $2 - MOD$ as they are just compositions of macro tree transducers. This does not, however, give us a way of writing them as such.

C.2 The Attributed Family

These classes of tree transducers were mostly developped in Hungary by Zoltán Fülöp, in collaboration with Heiko Vogler and Armin Kühnemann in Germany.

C.2.1 Attributed tree transducers

Attributed tree transducers were directly inspired by context-free grammars, and more specifically attribute grammars. The first mention of them I found was in [Fül81], that I couldn't find and read. The definition given here is the one taken from [FV98].

Definition 16 *An attributed tree transducer is a tuple* $A = (Att, \Sigma, \Delta, a_0, R, E)$ *where* Att *is a set partitioned into the disjoint subsets* Att_{syn} *and* Att_{inh}, Σ *and* Δ *are ranked alphabets disjoint from* Att, $a_0 \in Att_{syn}$, R *is a set such that* $R = \bigcup_{\sigma \in \Sigma} R_\sigma$ *where each* R_σ *satisfies the following conditions:*

1. *for every* $a \in Att_{syn}$ *and* $\sigma \in \Sigma^{(k)}$ *with* $k \geq 0$, R_σ *contains exactly one rule of the form* $a(\pi) \to \xi$ *where* $\xi \in \mathrm{RHS}(Att_{syn}, Att_{inh}, \Delta, k)$.

2. *for every* $b \in Att_{inh}$ *and* $\sigma \in \Sigma^{(k)}$ *with* $k \geq 0$ *and* $1 \leq i \leq k$, R_σ *contains exactly one rule of the form* $b(\pi i) \to \xi$ *where* $\xi \in \mathrm{RHS}(Att_{syn}, Att_{inh}, \Delta, k)$

, and E *is a mapping* : $Att_{inh} \to T_\Delta$.

The right-hand sides are defined as follows:

Definition 17 *Let* A_s *and* A_i *be disjoint unary ranked alphabets,* Δ *be a ranked alphabet disjoint from* A_s *and* A_i, *and* $k \geq 0$ *be an integer. The set*
$\mathrm{RHS}(A_s, A_i, \Delta, k)$ *of right-hand sides over* A_s,A_i,Δ *and* k *is the smallest subset* rhs $\subseteq T_{A_s \bigcup A_i \bigcup \Delta}(\{\pi, \pi 1, \ldots, \pi k\})$ *satisfying the following conditions.*

1. *for every* $a \in A_s$ *and* $1 \leq i \ \leq k$, $a(\pi i) \in$ RHS

2. *for every* $b \in A_i$, $b(\pi) \in$ RHS

3. *for every* $\delta \in \Delta^{(l)}$ *with* $l \geq 0$ *and* $\xi_1, \ldots, \xi_l \in$ RHS, $\delta(\xi_1, \ldots, \xi_l) \in$ RHS.

Note that we can give yet another definition for top-down tree transducers, as it is obvious that for every attributed tree transducer with $Att_{inh} = \emptyset$, we can build an equivalent top-down tree transducer and vice versa.
The class of attributed tree transducers is called ATT.

Composition Results

There are no composition results per se, for attributed tree transducers. However, it is easy to show that $ATT \subsetneq MAC$. Thus, most composition results for subclasses of MAC are true in ATT, and some of them even need ATT to be proved.

C.2.2 Macro-attributed tree transducer

Macro-attributed tree transducers were introduced in [KV94] to mix the features of macro tree transducers and attributed tree transducers. The result is, unfortunately, not more powerful than any of the first two, as their composition hierarchies are entangled together.

Definition 18 *A macro attributed tree transducer is a tuple* $N = (Att, \Sigma, \Delta, a_0, R, E)$ *where*

- Att *is a ranked alphabet with elements of at least rank 1, and is partitioned into the disjoint alphabets* Att_{syn} *and* Att_{inh}.
- Σ *and* Δ *are ranked alphabets.*
- $a_0 \in Att_{syn}^{(r+1)}$ *sor some* $r \geq 0$.
- R *is the union* $\bigcup_{\sigma \in \Sigma} R_\sigma$ *of finite sets* R_σ *of rules satisfying the conditions:*
  1. *for every* $a \in Att_{syn}^{(n+1)}$ *with* $n \geq 0$ *and* $\sigma \in \Sigma^{(k)}$ *with* $k \geq 0$, R_σ *contains exactly one rule of the form* $a(\pi, y_1, \ldots, y_n) \to \xi$ *where* $\xi \in \mathrm{RHS}(Att_{syn}, Att_{inh}, \Delta, k, n)$.
  2. *for every* $b \in Att_{inh}^{(n+1)}$ *with* $n \geq 0$, $\sigma \in \Sigma^{(k)}$ *with* $k \geq 1$ *and* $1 \leq i \leq k$, R_σ *contains exactly one rule of the form* $b(\pi i, y_1, \ldots, y_n) \to \xi$ *where* $\xi \in \mathrm{RHS}(Att_{syn}, Att_{inh}, \Delta, k, n)$.
- $E = (E_1, E_2)$ *where* $E_1 \in (T_\Delta)^r$ *and* $E_2 : Att_{inh} \to T_\Delta(Y)$ *is a mapping such that for every* $b \in Att_{inh}^{(k+1)}$ *with* $k \geq 0$, $E_2(b) \in T_\Delta(Y_k)$.

The right-hand sides are defined as follows.

Definition 19 *Let* A_s *and* A_i *be disjoint ranked alphabets,* Δ *be a ranked alphabet disjoint from* A_s *and* A_i, *and* $k, n \geq 0$ *be integers. The set* $\mathrm{RHS}(A_s, A_i, \Delta, k, n)$ *of right-hand sides over* A_s,A_i,Δ, k *and* n *is the smallest subset* $\mathrm{rhs} \subseteq T_{A_s \bigcup A_i \bigcup \Delta}(\{\pi, \pi 1, \ldots, \pi k\} \bigcup Y_n)$ *satisfying the following conditions.*

1. *for every* $a \in A_s^{(l+1)}$ *with* $l \geq 0$, $1 \leq i \ \leq k$, *and* $\xi_1, \ldots, \xi_l \in \mathrm{RHS}$, $a(\pi i, \xi_1, \ldots, \xi_l) \in \mathrm{RHS}$
2. *for every* $b \in A_i^{(l+1)}$ *with* $l \geq 0$ *and* $\xi_1, \ldots, \xi_l \in \mathrm{RHS}$, $b(\pi, \xi_1, \ldots, \xi_l) \in \mathrm{RHS}$
3. *for every* $\delta \in \Delta^{(l)}$ *with* $l \geq 0$ *and* $\xi_1, \ldots, \xi_l \in \mathrm{RHS}$, $\delta(\xi_1, \ldots, \xi_l) \in \mathrm{RHS}$.
4. $Y_n \subseteq \mathrm{RHS}$.

The classe of functions computed by macro-attributed tree transducer is called MAT.

Composition Results

It has been proved that $MAT = ATT^2$. Thus, the composition hierarchies of MAT, ATT, and MAC are entwined together, as shown by figure C.2.

C.2.3 Higher-order attributed tree transducers

This class of tree transducer was introduced in 2001 by Thomas Noll and Heiko Vogler ([NV01]) to allow attributed tree transducers to modify their control (input) tree. They define the higher-order attributed tree transducers as follows.

Definition 20 *A higher-order attributed tree transducer is a tuple* $M = (Syn, Inh, a_0, tree, [\Sigma], \Delta, \delta_0, \alpha, R)$ *where*

- Syn *is a unary ranked alphabet.*
- Inh *and* Δ *are disjoint ranked alphabets, and they are disjoint from* Syn*.*
- $a_0 \in Syn$
- $\delta_0 \in \Delta^{(r+1)}$ *for some* $r \geq 0$*.*
- $\alpha \in \Delta^{(0)}$*.*
- $[\Sigma] = (\Sigma_1, \ldots, \Sigma_r)$ *such that every* $\Sigma_i \subseteq \Delta$*.*
- $R = \bigcup_{\delta \in \Delta\{\alpha\}} R_\delta$ *where every* R_δ *is a finite set of rules satisfying the following conditions. For every* $\delta \in \Delta^{(k)}$ $\{\alpha\}$ *with* $k \geq 0$*,*
  - *for every* $a \in Syn$*, there is exactly one rule in* R_δ *of the form* $\langle a, \epsilon \rangle \to \eta$*,*
  - *for every* $b \in Inh$ *and* $1 \leq j \leq k$*, there is exactly one rule of the form* $\langle b, j \rangle \to \eta$*,*

  where, in both cases, $\eta \in T_\Delta(Syn \times \{1, \ldots, k\} \bigcup Inh \times \{\epsilon\})$ *if* $\delta \neq \delta_0$ *and eta* $\in T_\Delta(Syn \times \{1, \ldots, k\})$ *if* $\delta = \delta_0$*.*

The class of functions computed by higher-order attributed tree transducers is called $HO - ATT$ and has not been studied enough yet to provide interesting results.

C.3 Inclusion Diagrams

Proofs for these inclusions can be found in:

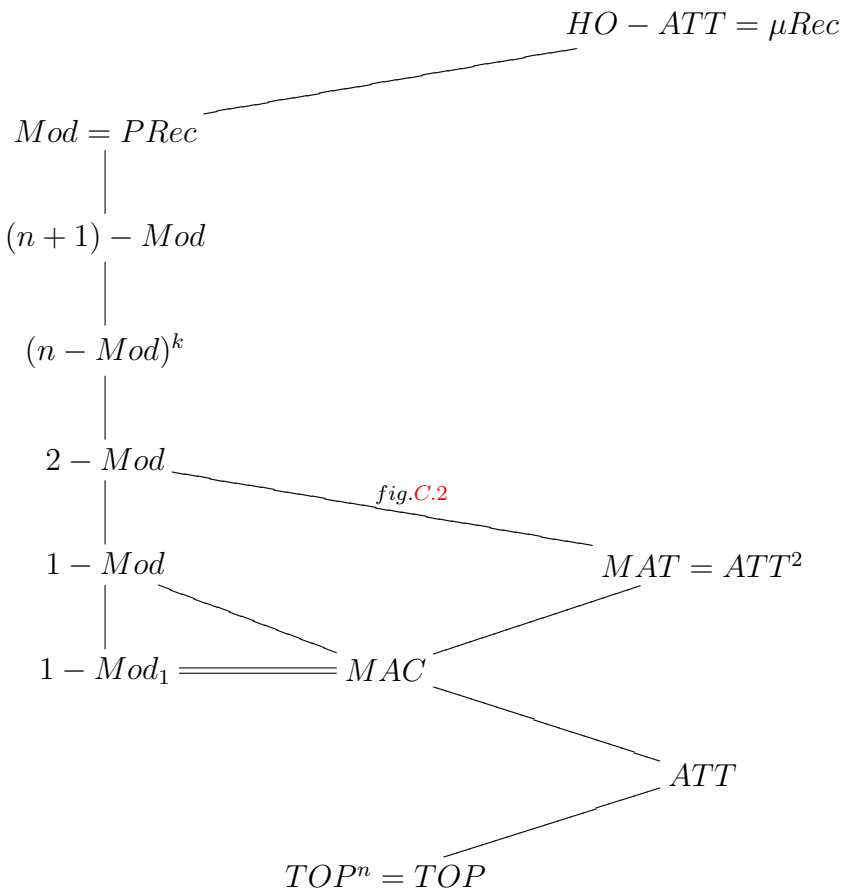


Figure C.1: General Hasse diagram for tree transductions classes.

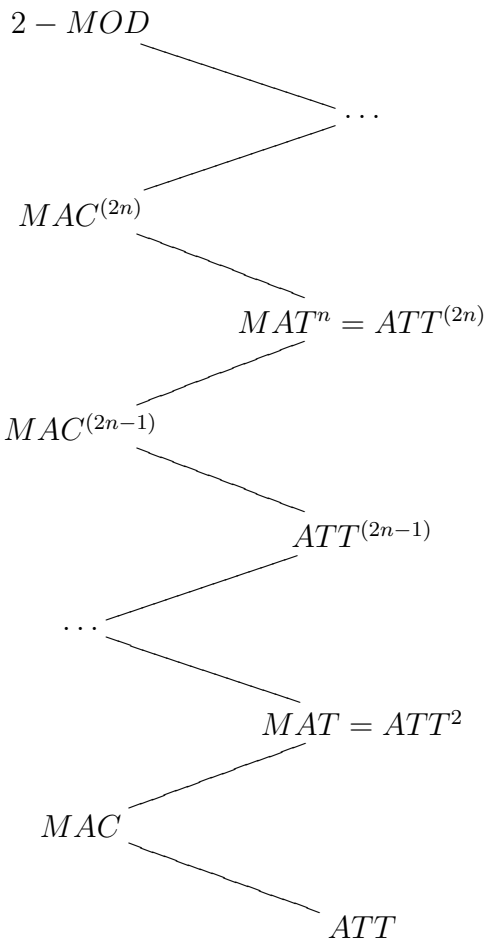


Figure C.2: The crossed composition hierarchy for MAC, MAT and ATT.

- [EV91] for all modular tree transducer inclusions.
- [NV01] for $HO - ATT = \mu Rec$
- [FV98] for all other inclusions. The book refers to the original proof.

In this diagram, I did not represent the composition hierarchy for MAC, ATT and MAT, but it is pretty easy to realize that we have a perfect alternation that is fully and properly included in $2 - MOD$.

Appendix D

Tree Transducers: Talk at FMCS'07

The following document is the presentation slides for my talk at Foundational Methods in Computer Science 2007 in Hamilton, NY (June 2007). It makes a quick survey of different useful results on tree transducers for deforestation and presents ideas for future studies.

A tree transducer approach to deforestation

François Dupressoir[1]
under the supervision of Robin Cockett[1]

[1] University of Calgary

Foundational Methods in Computer Science
Hamilton, New York
June 10, 2007

Deforestation

Objectives

- Reduce overhead due to intermediate data structures;
- Remove intermediate data structures;
- Compute intermediate data structure-free function compositions at compile-time.

Example

- `sum (map square (from 0 n))`
- ```
  letrec ssf n =
    case n of
        0 -> 0
      | _ -> n*n + ssf (n-1)
  ```

Deforestation Techniques

Wadler's Deforestation Algorithm computes function compositions compositions in *treeless form* by unfolding and refolding the function calls ([Wad90]).

Cockett's proof theoretic technique explains Wadler's algorithm using *circular proofs* ([Coc01]).

Short cut fusion removes intermediate lists from functions using *cata-* and *ana*-morphisms ([MT95],[GLP93],[Gil96]).

Tree transducers Allow representation of certain classes of programs. Their syntactic composition can represent the fusion of programs ([JV04]).

Algebraic and monadic transducers Allow representation of programs using categorical structures and generalizing fusion using the acid rain theorem ([Jür03]).

Tree transducers.

Advantages

Tree transducers:

- are close to the computation;
- are close to the program;
- handle mutual recursion in a natural way.

Two very different ideas

Modular tree transducers are based on the function's syntax.

Attributed tree transducers are based on the function's semantic interpretation.
  ```

Outline

A quick presentation

Definition

A *top-down tree transducer* is a tuple $(Q, \Sigma, \Delta, q_0, R)$ where Q is the finite set of functions, Σ and Δ are *ranked alphabets* (finite set of variables with arities), $q_0 \in Q$, and $R \subseteq T_{Q \cup \Sigma}(X) \times T_{Q \cup \Delta}(X)$ such that:
$\forall q \in Q, \sigma \in \Sigma^{(k)}, \exists! rhs_{q,\sigma} \in T_{Q \cup \Delta}(X).\ (q(\sigma(x_1, \ldots, x_k)), rhs_{q,\sigma}) \in R$

Intuition

$q \in Q \leftrightarrow$function
$\Sigma \leftrightarrow$input datatype
$\Delta \leftrightarrow$output datatype
$R \leftrightarrow$equations

Note

If $(lhs, rhs) \in R$, we will also write $lhs \to rhs \in R$ or $lhs \to_R rhs$ or $lhs \to rhs$.

Top-down tree transducers

Example: length of a list

$T_{len} = (\{\mathtt{len}\}, \{cons^{(2)}, nil^{(0)}\}, \{S^{(1)}, Z^{(0)}\}, \mathtt{len}, R_{len})$ where

$$R_{len} = \left\{ \begin{array}{ll} \mathtt{len}(cons(x_1, x_2)) & \to S(\mathtt{len}(x_2)); \\ \mathtt{len}(nil) & \to Z \end{array} \right\}$$

Example: doubling a natural number

$T_{dbl} = (\{\mathtt{dbl}\}, \{S^{(1)}, Z^{(0)}\}, \{S^{(1)}, Z^{(0)}\}, \mathtt{dbl}, R_{dbl})$ where

$$R_{dbl} = \left\{ \begin{array}{ll} \mathtt{dbl}(S(x_1)) & \to S(S(\mathtt{dbl}(x_1))); \\ \mathtt{dbl}(Z) & \to Z \end{array} \right\}$$

Top-Down tree transducers
Composition (1)

Definition

Let T_1 and T_2 be top-down tree transducers. Let T_2' be the top-down tree transducer defined by:

$$
\begin{aligned}
Q_2' &= Q_2 \\
\Sigma_2' &= \Sigma_2 \bigcup \left\{ (q(\sigma(x_1, \ldots, x_k)))^{(0)} | q \in Q_1, \sigma \in \Sigma_1^{(k)} \right\} \\
\Delta_2' &= \Delta_2 \bigcup \left\{ ((q_2, q_1)(\sigma_1(x_1, \ldots, x_k))^{(0)} | q_i \in Q_i, \sigma_1 \in \Sigma_1^{(k)} \right\} \\
R_2' &= R_2 \bigcup \{ (q_2(q_1(\sigma(x_1, \ldots, x_n))), (q_2, q_1)(\sigma(x_1, \ldots, x_n))) \}
\end{aligned}
$$

Top-Down tree transducers
Composition (2)

Definition

We can now define a new top-down tree transducer $T_2 \circ T_1$ as follows:

$$
\begin{aligned}
Q_{2\circ1} &= Q_2 \times Q_1 & \Sigma_{2\circ1} &= Sigma_1 \\
\Delta_{2\circ1} &= \Delta_2 & q_{0_{2\circ1}} &= (q_{0_2}, q_{0_1}) \\
R_{2\circ1} &= \left\{ ((q_2, q_1)(\sigma(x_1, \ldots, x_n), [\![M_2']\!]_{q_2} (rhs_{1,q_1,\sigma}) \right\}
\end{aligned}
$$

where $[\![M]\!]_q (e)$ represents the "natural" interpretation (semantics) of the tree transducer starting in state q with input e.

Intuition

- Intermediate substructures are consumed as soon as they are produced.
- The only structure that is actually built is the final result.

Top-Down tree transducers
Composition (3)

Example

$$
R'_{dbl} = \left\{ \begin{array}{ll}
\mathtt{dbl}(S(x_1)) & \rightarrow S(S(\mathtt{dbl}(x_1))); \\
\mathtt{dbl}(Z) & \rightarrow Z; \\
\mathtt{dbl}(\mathtt{len}(cons(x_1, x_2))) & \rightarrow (\mathtt{dbl}, \mathtt{len})(cons(x_1, x_2)); \\
\mathtt{dbllen}(nil) & \rightarrow (\mathtt{dbl}, \mathtt{len})(nil)
\end{array} \right\}
$$

and

$$
T_{dbl} \circ T_{len} = (\{(\mathtt{dbl}, \mathtt{len})\}, \{cons^{(2)}, nil^{(0)}\}, \{S^{(1)}, Z^{(0)}\}, (\mathtt{dbl}, \mathtt{len}), R_{dbl\circ len})
$$

where

$$
R_{dbl\circ len} = \left\{ \begin{array}{ll}
(\mathtt{dbl}, \mathtt{len})(cons(x_1, x_2)) & \rightarrow [\![T'_{dbl}]\!]_{\mathtt{dbl}} (S(\mathtt{length}(x_2))); \\
(\mathtt{dbl}, \mathtt{len})(nil) & \rightarrow [\![T'_{dbl}]\!]_{\mathtt{dbl}} (Z)
\end{array} \right\}
$$

Theorem

Top-down tree transducers are stable under composition.

Top-down tree transducers
Summary

- Good composition properties make them good candidates for deforestation ([Jür03]).
- Not powerful enough to express useful functions.

Objectives
We want to find a way of having more than one argument, and still have some interesting composition results.

Macro tree transducers
Definition: *Right hand sides*

Definition
We first define the set of *right hand sides* of a macro tree transducer as follows. The set $RHS(Q, \Delta, m, n)$ is the smallest set $\text{rhs} \subseteq T_{Q \cup \Delta}(X_m \bigcup Y_n)$, with X_m and Y_n some disjoint sets of variables, such that:

1. $Y_n \subseteq \text{rhs}$
2. $\forall k \in \mathbb{N}, x_i \in X_m, q \in Q^{(k+1)}, \xi_1, \ldots, \xi_k \in \text{rhs}.$ $q(x_i, \xi_1, \ldots, \xi_k) \in \text{rhs}$
3. $\forall k \in \mathbb{N}, \delta \in \Delta^{(k)}, \xi_1, \ldots, \xi_k \in \text{rhs}.$ $\delta(\xi_1, \ldots, \xi_k) \in \text{rhs}$

Intuition
Accumulation and recursive function calls in parameter positions.

Macro tree transducers
Definition

Definition
A *macro tree transducer* is a tuple $(Q, q_0, \Sigma, \Delta, R)$ where Q is a ranked alphabet, $q_0 \in Q$ with $\text{rank}_Q(q) = 1$, Δ and Σ are ranked alphabets, and $R \subseteq T_{Q \cup \Delta}(X \bigcup Y) \times T_{Q \cup \Delta}(X \bigcup Y)$ with $\forall q \in Q^{(r+1)}, \sigma \in \Delta^{(k)}, \exists! \text{rhs}_{q,\sigma} \in RHS(Q, \Delta, k, r).$ $(q(\sigma(x_1, \ldots, x_k), y_1, \ldots, y_r), \text{rhs}_{q,\sigma}) \in R.$

Intuition
Generalizes top-down tree transducers in a natural way:

$q \in Q$ $\leftrightarrow$function
q_0 $\leftrightarrow$root function (embeds the environment)
Δ $\leftrightarrow$all used datatypes
Σ_i $\leftrightarrow$input datatype for argument i
R $\leftrightarrow$functional equations

Macro tree transducers
Example

Appending lists

$M_{app} = \left(\left\{\mathtt{app}^{(2)}\right\}, \mathtt{app}, \left\{cons^{(2)}, nil^{(0)}\right\}, \left\{cons^{(2)}, nil^{(0)}\right\}, R_{app}\right)$

where $R = \left\{ \begin{array}{ll} \mathtt{app}(cons(x_1, x_2), y_1) & \rightarrow cons(x_1, \mathtt{app}(x_2, y_1)) \\ \mathtt{app}(nil, y_1) & \rightarrow y_1 \end{array} \right\}$

Macro tree transducers
Composition

- Macro tree transducers are not stable under composition.
- We want to find subclasses (restricted right hand sides) that have good composition properties.

Example

The composition of two *single-use* macro tree transducers is a macro tree transducer.

Let *MAC* and *TOP* represent the class of functions computable by macro and top-down tree transducers, respectively.

- we have $MAC \circ TOP = MAC = TOP \circ MAC$
- we can build the corresponding tree transducer by adapting the previous construction.

Macro tree transducers
Summary

- Powerful enough to represent usual functions.
- The composition properties require analysis.
- We want more general composition results.

Definition

A *modular tree transducer* is a tuple $(Q, \mathrm{mod}, q_0, [\Sigma], \Delta, R)$ where Q is a ranked alphabet with $\forall q \in Q, \mathrm{rank}(q) \geq 1$, mod is a function $Q \to \mathbb{N}$, $q_0 \in Q$ with $\mathrm{mod}(q_0) = 1$ and $R \subseteq T_{Q \cup \Delta}(X \bigcup Y) \times T_{Q \cup \Delta}(X \bigcup Y)$ with $\forall q \in Q^{(r+1)}, \sigma \in \Delta^{(k)}, \exists! \mathrm{rhs}_{q,\sigma} \in T_{Q \cup \Delta}(X \bigcup Y). (q(\sigma(\ldots), \ldots), \mathrm{rhs}_{q,\sigma}) \in R$.
Moreover, for all rules, the following conditions have to hold:

1. for every $p \in Q$ occuring in $\mathrm{rhs}_{q,\sigma}$, $\mathrm{mod}(p) \geq \mathrm{mod}(q)$
2. if $p(\xi_1, \ldots, \xi_{k'})$ occurs in $\mathrm{rhs}_{q,\sigma}$ with $\mathrm{mod}(p) = \mathrm{mod}(q)$ then $\xi_1 \in \{x_1, \ldots, x_{\mathrm{rank}(\sigma)}\}$

Intuition

- different levels of function definitions.
- only call functions defined at a higher level.
- (mutual) recursive calls occur only in parameter positions.

Naïve `rev`

$M_{nrev} = \left(\left\{\mathtt{rev}^{(1)}, \mathtt{app}^{(2)}\right\}, \begin{pmatrix} \mathtt{rev} \mapsto 1 \\ \mathtt{app} \mapsto 2 \end{pmatrix}, \mathtt{rev}, [\Sigma], \Sigma, R\right)$ where

$\Sigma = \left\{cons^{(2)}, nil^{(0)}\right\}$

and $R = \left\{ \begin{array}{ll} \mathtt{app}(cons(x_1, x_2), y) & \to cons(x_1, \mathtt{app}(x_2, y)) \\ \mathtt{app}(nil, y) & \to y \\ \mathtt{rev}(cons(x_1, x_2)) & \to \mathtt{app}(\mathtt{rev}(x_2), cons(x_1, nil)) \\ \mathtt{rev}(nil) & \to nil \end{array} \right\}$

- Strict composition hierarchy: $n - MOD \subsetneq (n+1) - MOD$ and $n - MOD \circ n - MOD \subsetneq (n+1) - MOD$ ([EV91]).
- $1 - MOD_1 = MAC$, $MOD = MAC_f$ where MOD_1 is the class of unary modular tree transducers, MAC_f is the class of macro tree transducers with external functions ([FHVV93]).
- $MOD^n \subset 1 - MOD$

Objectives

- The strict composition hierarchy does not allow deforestation.
- Under certain circumstances, we can reduce the number of modules ([KGK01]).
- We need composition results for macro tree transducers.

Attributed tree transducers
Definition: *Right-hand sides*

Definition
Let A_s and A_i be disjoint unary ranked alphabets, Δ be a ranked alphabet disjoint from A_s and A_i and $k \geq 0$ an integer. The set $RHS(A_s, A_i, \Delta, k)$ of right-hand sides over A_s, A_i, Δ and k is the smallest subset RHS of $T_{A_s \bigcup A_i \bigcup \Delta}(\{\pi, \pi 1, \ldots, \pi k\})$ satisfying:

1. $\forall a \in A_s, 1 \leq i \leq k, a(\pi i) \in RHS$
2. $\forall b \in A_i, b(\pi) \in RHS$
3. $\forall l \geq 0, \delta \in \Delta^{(l)}, \xi_1, \ldots, \xi_l \in RHS, \delta(\xi_1, \ldots, \xi_l) \in RHS$

Note
π is here considered as a string variable, representing the path to the current node in the control tree. πi is the path to the ith child of the node of path π.

Attributed tree transducers
Definition

Definition
An *attributed tree transducer* is a tuple $A = (Att, \Sigma, \Delta, a_0, R, E)$ where $Att = Att_{inh} \bigcup Att_{syn}$ is the disjoint union of two ranked alphabets, Σ and Δ are ranked akphabets, $a_0 \in A_{syn}$, $E : Att_{inh} \rightarrow T_\Delta$ and $R = \bigcup_{\sigma \in \Sigma} R_\sigma$ where each R_σ satisfies:

1. $\forall a \in Att_{syn}. \exists! a(\pi) \rightarrow \xi \in R_\sigma$ where $\xi \in RHS(Att_{syn}, Att_{inh}, \Delta, \mathrm{rank}_\Sigma(\sigma))$
2. $\forall b \in Att_{inh}, 1 \leq i \leq \mathrm{rank}_\Sigma(\sigma). \exists! b(\pi i) \rightarrow \xi \in R_\sigma$ where $\xi \in RHS(Att_{syn}, Att_{inh}, \Delta, \mathrm{rank}_\Sigma(\sigma))$

Intuition
An attributed tree transducer "is" an attributed system.

Attributed tree transducers
Example

Shifting trees

- $Att_{syn} = \{a_0, a_1\}, Att_{inh} = \{b\}$
- $\Sigma = \Delta = \left\{\#^{(1)}, \sigma^{(2)}, \alpha_1^{(0)}, \alpha_2^{(0)}\right\}$
- $E = (b \mapsto \alpha_1)$
- $$R_{\alpha_1} = \left\{ \begin{array}{l} a_0(\pi) \rightarrow b(\pi) \\ a_1(\pi) \rightarrow \alpha_1 \end{array} \right\}, \; R_\# = \left\{ \begin{array}{l} a_0(\pi) \rightarrow \#(a_0(\pi 1)) \\ a_1(\pi) \rightarrow \alpha_1 \\ b(\pi 1) \rightarrow a_1(\pi 1) \end{array} \right\}$$
  $$R_{\alpha_2} = \left\{ \begin{array}{l} a_0(\pi) \rightarrow b(\pi) \\ a_1(\pi) \rightarrow \alpha_2 \end{array} \right\}, \; R_\sigma = \left\{ \begin{array}{l} a_0(\pi) \rightarrow a_0(\pi 2) \\ a_1(\pi) \rightarrow a_1(\pi 1) \\ b(\pi 1) \rightarrow \alpha_1 \\ b(\pi 2) \rightarrow \sigma(b(\pi), a_1(\pi 2)) \end{array} \right\}$$

Attributed tree transducers
Some results

- Let A be an attributed tree transducer. There exists a macro tree transducer M that computes the same program ([FV98]). Thus, $ATT \subseteq MAC$ (and we even have a strict inclusion).
- $ATT \circ ATT \not\subseteq ATT$
- $ATT \circ TOP = ATT$, but $TOP \circ ATT \not\subseteq ATT$
- But we have some weaker composition results, that can be related with composition results for macro tree transducers (and are even used to indirectly compose mtts).

Macro-attributed tree transducers
Ideas and results

- Generalize attributed tree transducers using the concept of *synthesized* and *inherited functions* ([KV94]).
- $MAT = ATT \circ ATT$
- We have a strict composition hierarchy: $MAT^n \subsetneq MAT^{n+1}$
- $MAC \subsetneq MAT$ (thus $ATT \subsetneq MAT$).

Goals (and current progress)

- Bridge the gap between modular and attributed worlds.
- Bridge the gap between tree transduction world and Wadler's deforestation. Two main leads:
  - represent programs in *treeless form* as a specific class of tree transducers.
  - use *circular proofs* to make a four-way bridge between category theory, proof theory, Wadler's deforestation and tree transduction.
- Understand the existing bridges (algebraic and monadic transducers).


Robin Cockett.

Deforestation, program transformation, and cut-elimination.

Electronic Notes in Theoretical Computer Science, 47:1–40, 2001.


Joost Engelfriet and Heiko Vogler.

Modular tree transducers.

Theoretical Computer Science, 78(2):267–303, 1991.


Zoltán Fülöp, Frank Herrmann, Sándor Vágvölgyi, and Heiko Vogler.

Tree transducers with external functions.

Theoretical Computer Science, 108(2):185–236, 1993.


Zoltán Fülöp and Heiko Vogler.

Syntax-Directed Semantics: Formal Models Based on Tree Transducers.

Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.


Andrew Gill.

Cheap Deforestation for Non-strict Functional Languages.

PhD thesis, University of Glasgow, 1996.


Andrew Gill, John Launchbury, and Simon Peyton Jones.

A short cut to deforestation.

Functional Programming languages and Computer Architecture, pages 223–232, 1993.


Claus Jürgensen.

Categorical semantics and composition of tree transducers.

PhD thesis, Technischen Universität Dresden - Fakultät Informatik, 2003.


Claus Jürgensen and Heiko Vogler.

Syntactic composition of top-down tree transducers is short cut fusion.

Mathematical Structures in Computer Science, 14(2):215–282, 2004.


Armin Kühnemann, Robert Glück, and Kazuhiko Kakehi.

Relating accumulative and non-accumulative functional programs.

In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 154–168, London, UK, 2001. Springer-Verlag.


Armin Kühnemann and Heiko Vogler.

Synthesized and inherited functions. a new computational model for syntax-directed semantic.

Acta Informatica, 31(5):431–477, 1994.


Erik Meijer and Akihiko Takano.

Shortcut deforestation in calculational form.

In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 306–313, New York, NY, USA, 1995. ACM Press.


Philip Wadler.

Deforestation: Transforming programs to eliminate trees.

Theoretical Computer Science, 73:231–248, 1990.

Bibliography

[CFZ82a] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute grammars and recursive program schemes i. *Theoretical Computer Science*, 17(2):163–191, 1982. 42

[CFZ82b] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute grammars and recursive program schemes ii. *Theoretical Computer Science*, 17(3):235–257, 1982. 42

[Coc01] Robin Cockett. Deforestation, program transformation, and cut-elimination. *Electronic Notes in Theoretical Computer Science*, 47:1–40, 2001. 11, 12, 14

[EFV01] Joost Engelfriet, Zoltán Fülöp, and Heiko Vogler. Bottom-up and top-down tree series transformations. *Journal of Automata, Languages and Combinatorics*, 7(1):11–70, 2001. 41

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71–146, 1985. 42

[EV91] Joost Engelfriet and Heiko Vogler. Modular tree transducers. *Theoretical Computer Science*, 78(2):267–303, 1991. 7, 42, 48

[Fül81] Zoltán Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981. 44

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998. 9, 44, 48

[Gil96] Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1996. 5, 6, 17

[GLP93] Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. *Functional Programming languages and Computer Architecture*, pages 223–232, 1993. 5

[Joh01] Patricia Johann. Short cut fusion: Proved and improved. In *SAIG '01*, pages 47–71. Springer-Verlag, 2001. 6

[Jür03] Claus Jürgensen. *Categorical semantics and composition of tree transducers.* PhD thesis, Technischen Universität Dresden - Fakultät Informatik, 2003. 2, 21, 22, 41, 42

[JV04] Claus Jürgensen and Heiko Vogler. Syntactic composition of top-down tree transducers is short cut fusion. *Mathematical Structures in Computer Science*, 14(2):215–282, 2004. 22

[KGK01] Armin Kühnemann, Robert Glück, and Kazuhiko Kakehi. Relating accumulative and non-accumulative functional programs. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 154–168, London, UK, 2001. Springer-Verlag. 24

[Küh98] Armin Kühnemann. Benefits of tree transducers for optimizing functional programs. In Springer-Verlag, editor, *LNCS*, volume 1530, pages 146–157, 1998. 7

[KV94] Armin Kühnemann and Heiko Vogler. Synthesized and inherited functions. a new computational model for syntax-directed semantic. *Acta Informatica*, 31(5):431–477, 1994. 45

[MT95] Erik Meijer and Akihiko Takano. Shortcut deforestation in calculational form. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 306–313, New York, NY, USA, 1995. ACM Press. 21, 22

[NV01] Thomas Noll and Heiko Vogler. The universality of higher-order attributed tree transducers. *Theory of Computing Systems*, 34(1):45–75, 2001. 46, 48

[Rou68] William Rounds. *Tress, transducers and transformations.* PhD thesis, Stanford university, 1968. 41

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. 3, 24