Warm Fusion: Deriving Build-Catas from Recursive Definitions

John Launchbury & Tim Sheard
Oregon Graduate Institute of Science & Technology
P.O. Box 91000, Portland, OR 97291-1000 USA
{jl,sheard}@cse.ogi.edu

Abstract

Program fusion is the process whereby separate pieces of code are fused into a single piece, typically transforming a multi-pass algorithm into a single pass. Recent work has made it clear that the process is especially successful if the loops or recursions are expressed using catamorphisms (e.g.`foldr`) and constructor-abstraction (e.g. `build`). In this paper we show how to transform *recursive programs* into this form automatically, thus enabling the fusion transformation to be applied more easily than before.

1 Introduction

There are significant advantages to multi-pass algorithms, in which intermediate data-structures are created and traversed. In particular, each of the passes may be relatively simple, so are both easier to write and are potentially more reusable. By separating many distinct phases it becomes possible to focus on a single task, rather than attempting to do many things at the same time.

The classic toy example of this is to compute the sum of the squares of the numbers from 1 to n. We might write it as, *sum* (*map square* [*1* ..*n*]). There are two intermediate lists here. First the list of numbers [1..n], and second, the list of squares of those numbers. These lists serve the very useful purpose of acting as the "glue" which binds the components of the algorithm together. In this case the components are entirely standard, "off the shelf" parts.

Here's the rub, however. A direct implementation of the sum of squares would *literally* construct, traverse, and discard the intermediate lists, and so degrade execution time dreadfully compared with a function written to compute the sum of squares directly using, say, an accumulating parameter.

The problem is not simply one of lists. Compiler writers commonly face the challenge of deciding how many passes their compiler should perform. Exactly the same tradeoff is present. Increasing the number of passes means simpler, more modular and more maintainable code, whereas decreasing the number of passes leads to greater run-time efficiency. In this case the intermediate structure is an abstract-syntax tree.

Another telling example is that of depth-first traversal of graphs. Many standard algorithms may be expressed very succinctly and simply as explicit manipulations of a graph's depth-first spanning forest, but in execution the literal presence of the forest is inefficient [KL95].

The long-sought solution to this tension between modularity and reuse on the one hand, and efficiency on the other, is to increase the transformation-power of compilers so that the programmer may feel liberated to write programs in a component style, confident that the compiler can *fuse* the components together, removing the intermediate structures. In general, of course, we cannot expect that all intermediate structures can be removed: data structures do serve a true computational role. We can hope, however, that "unnecessary" intermediate structures will be removed whenever possible.

In this paper, we show how to preprocess recursive function definitions, turning them into a form particularly suited to enabling component fusion.

2 Background

Darlington and Burstall showed how fold-unfold transformations could be used (with human help) to produce single-pass function definitions from the composition of two or more other functions [DB76]. Turchin applied similar ideas within his supercompilation process [Tur86]. The *super*vising-*compiler* performed a symbolic execution of the program, building a residual program whenever computations could not be performed. Patterns in the nesting of recursive calls of functions were spotted and a single new recursive function produced. While the process showed a high degree of automation, the method of generalisation used to control termination was rather crude.

Wadler also developed many of these same ideas in his *listless transformer* [Wad84]. Multi-pass programs with intermediate lists were converted into single loops in an imperative language. Later, it became clear that the method could be expressed within the functional language itself, and he further refined the idea, calling the method *deforestation* [Wad90], a pun which has stuck and is now commonly used. In order to be precise about which intermediate lists were removed, Wadler defined a *treeless form* for function definitions in which functions were guaranteed to have no internal data structures. He then proved that compositions of treeless functions, could be deforested into a single treeless function. A termination proof followed later [FW89].

The original deforestation work was limited to first order. Attempts to extend it to higher order have met with limited success so far. In particular, it is often hard to find where to tie the recursive knot. Termination proofs also seem to be rather hard.

Recently, the fusion process has received impetus from a different direction entirely. Rejecting the focus on arbitrary recursive programs, the recent work has focussed on the fusion of *catamorphisms* (literally: down-formers), the list version variously known as `fold` in ML, `reduce` in early versions of Miranda, and `foldr` in Haskell. Catamorphisms express "regular recursion" over data structures. Generalising the datatype-specific work of Bird and Meerten's, Malcolm popularised the promotion theorems from category theory which describe how to fuse catamorphisms [Mal89].

Much of this theory was turned into practice by Sheard and Fegaras [SF93]. Working with a language without general recursion but containing catamorphisms (and their generalisation: homomorphisms), Sheard and Fegaras implemented a fusion algorithm based on the promotion theorems. However, a problem arises in practice. Without some user-supplied guidance, the fusion engine attempts to fuse everything, leading to combinatorial explosion.

About the same time, Gill, Launchbury and Peyton Jones explored a one-step fusion algorithm which relied on functions being written in a highly-stylized form [GLPJ93]. Functions had to consume their arguments using the catamorphism `foldr` and produce their list-results by first abstracting over `cons` and `nil`, and then using a new language construct called `build`. While it was unrealistic to expect programmers to program this way, the list-processing functions from the Haskell standard-prelude were reprogrammed in this style. From then on, any combination of these standard functions were automatically deforested including, for example, the sum of squares example from the introduction.

2.1 This Paper

This paper builds on these last two. We generalise Sheard and Fegaras' fusion algorithm to cope with explicit recursion, and use it to *automatically* derive the `build/cata` form from functions definitions written in the common recursive style. We can then use the one-step fusion law to achieve inter-function deforestation.

The two steps of the process bear an interesting physical analogy. First we "ionise" function definitions to expose the nucleus—this is the `build/cata` form—and then the separate components of the program "plasma" are able to fuse in a single step, simply by bringing them into contact with one another. Experiments suggest that this is much more realistic than attempting "cold fusion" on the original function definitions.

This two step approach addresses some of the shortcoming of previous methods.

- It is not limited to lists, but works for a large class of data structures.
- The scope of the fusion engine is limited to the body of the function. The fusion engine is not used for inter-function deforestation. This helps to control combinatorial explosion.
- At no time in the process do we need to search for arbitrary patterns of recursive calls. The fusion engine simply needs to spot a recursive call to the function it is currently processing.

The cost of deriving `build/cata` form is amortised across fusions. We pay once to generate the form but may fuse it many times with many other functions.

3 Language

To demonstrate our techniques, we use the language given in Figure 1. A program is a sequence of definitions. Function definitions may be directly recursive (though, as presented, our techniques only work for direct recursion — i.e. no mutual recursion), but recursion in type definitions is specified using an explicit recursion operator, `Rec`.

For example, lists and binary trees may be defined as follows.

$$\begin{array}{lcl} \mathit{List} & = & \Lambda\alpha\ .\ \mathtt{Rec}\ \beta\ . \\ & & \quad \mathbf{Nil}\ ()+\mathbf{Cons}\ (\alpha\times\beta) \\ \mathit{Tree} & = & \Lambda\alpha\ .\ \mathtt{Rec}\ \beta\ . \\ & & \quad \mathbf{Tip}\ ()+\mathbf{Node}\ (\beta\times\alpha\times\beta) \end{array}$$

Any well formed type constructor definition $T = \ldots \mathbf{C}_i\ t_i\ \ldots$ with n constructors can be decomposed into a sequence of $n+2$ equations in which the recursion is opened out, and the argument to each constructor is named. For example:

$$\begin{array}{lcl} \mathit{List} & = & \Lambda\alpha\ .\ \mathtt{Rec}\ \beta\ .\ E^{\mathrm{List}}\ \alpha\ \beta \\ E^{\mathrm{List}} & = & \Lambda\alpha\ .\ \Lambda\beta\ . \\ & & \quad \mathbf{Nil}\ (E_{\mathrm{Nil}}\ \alpha\ \beta)\ +\ \mathbf{Cons}\ (E_{\mathrm{Cons}}\ \alpha\ \beta) \\ E_{\mathrm{Nil}} & = & \Lambda\alpha\ .\ \Lambda\beta\ .\ () \\ E_{\mathrm{Cons}} & = & \Lambda\alpha\ .\ \Lambda\beta\ .\ \alpha\times\beta \end{array}$$

Type constructors correspond to functors (in the categorical sense), so they have a natural action on functions as well as on types. For example, the natural action of *List* on a function is to map it down a list. In particular, the $E_{\mathbf{C}}$ for each constructor are also functors whose definitions we will need later. The $E_{\mathbf{C}}$ turn out to have a stylised form:

$$E_{\mathbf{C}}\ ::=\ (\Lambda\alpha_i\ .\ )^*\ \Lambda\beta\ .\ t$$

where the β is the variable over which type recursion occurs. This is the only variable over which we need to parameterise the function part of $E_{\mathbf{C}}$ (the others are all instantiated to the identity function for our purposes). We define $E_{\mathbf{C}}\ (g)\ =\ \mathcal{E}_g[\![\ t\ ]\!]$ where

$$\begin{array}{lcl} \mathcal{E}_g[\![\ ()\ ]\!] & = & \lambda()\ .\ () \\ \mathcal{E}_g[\![\ int\ ]\!] & = & \lambda x\ .\ x \\ \mathcal{E}_g[\![\ \alpha_i\ ]\!] & = & \lambda x\ .\ x \\ \mathcal{E}_g[\![\ \beta\ ]\!] & = & g \\ \mathcal{E}_g[\![\ (t_1\times\ldots\times t_n)\ ]\!] & & \\ & = & \lambda(x_1,\ldots,x_n)\ .\ (\mathcal{E}_g[\![\ t_1\ ]\!]\ x_1,\ldots,\mathcal{E}_g[\![\ t_n\ ]\!]\ x_n) \\ \mathcal{E}_g[\![\ T\ t\ ]\!] & = & \lambda x\ .\ map^T\ (\mathcal{E}_g[\![\ t\ ]\!])\ x \end{array}$$

where map^T is the usual functional component of the functor T. For example, we get the following functional behaviours for E_{Nil} and E_{Cons}:

$$\begin{array}{rcl} E_{\mathrm{Nil}}\ (g) & = & \lambda()\ .\ () \\ E_{\mathrm{Cons}}\ (g) & = & \lambda(x,y)\ .\ (x,\ g\ y) \end{array}$$

We will often use the notation $\bar{x}$ for the vector $(x_1,\ldots,x_n)$, especially when the size depends on the context. In addition because of the isomophism $(\Sigma\ t_i)\to t\cong\Pi\ (t_i\to t)$ and the definition $E(T)\ =\ \Sigma\ (E_{\mathbf{C}}(T))$ we often express functions out of a sum $E(\alpha)\ \to\ \alpha$ as the product of functions out of each summand $\Pi(E_{\mathbf{C}}(\alpha)\ \to\ \alpha)$. Thus the product of constructors $(\mathbf{Nil},\mathbf{Cons})$ has type $E^{\mathrm{List}}(\mathit{List}\ \alpha)\ \to\ \mathit{List}\ \alpha$, and the type of `cata` is: $(E^{\mathrm{List}}(\beta)\ \to\ \beta)\ \to\ \mathit{List}\ \alpha\ \to\ \beta$. Also, for any type constructor T the constructors of T can be specified by $\mathit{Constrs}(T)$. So, for example: $\mathit{Constrs}(\mathit{List})\ =\ (\mathbf{Nil},\mathbf{Cons}) : E^{\mathrm{List}}(\mathit{List}\ \alpha)\to\mathit{List}\ \alpha$.

$$
\begin{array}{llll}
e & ::= & v & \text{variables} \\
 & | & n & \text{constants} \\
 & | & (e_1, \ldots, e_n) & \text{tuples} \\
 & | & e_1\ e_n & \text{applications} \\
 & | & \texttt{case}\ e\ \texttt{of}\ \mathbf{C}_1 \bar{x}_1\ \texttt{->}\ e_1\ |\ \ldots\ |\ \mathbf{C}_n \bar{x}_n\ \texttt{->}\ e_n & \text{pattern matching case} \\
 & | & \lambda v\ .\ e & \text{lambda abstractions} \\
 & | & \lambda(v_1, \ldots, v_n)\ .\ e & \text{abstractions over tuples} \\
 & | & \texttt{cata}^T(e_1, \ldots, e_2) & \text{catamorphisms} \\
 & | & \texttt{build}^T\ e & \text{builds} \\
 & | & \mathbf{C} & \text{constructors} \\
 & & & \\
Decl & ::= & v = e & \text{(recursive) function definition} \\
 & | & T = (\Lambda \alpha_i\ .\ )^* Con & \text{type definition} \\
 & & & \\
Con & ::= & \texttt{Rec}\ \beta\ .\ \mathbf{C}_1\ t_1 + \ldots + \mathbf{C}_n\ t_n & \\
 & & & \\
t & ::= & ()\ |\ t_1 \times \ldots \times t_n\ |\ \alpha\ |\ int\ |\ T\ t & \text{constructor argument types}
\end{array}
$$

Figure 1: Abstract Syntax

The semantics of our language is that of a standard non-strict language with the addition of `build` and `cata`. The semantics of of these constructs obey the following equations.

$$
\begin{aligned}
&\texttt{cata}^T(f_1, \ldots, f_n)\ (C_i\ \overline{x}) \\
&\quad = \ f_i\ (E_i\,(\texttt{cata}^T(f_1, \ldots, f_n))\ \overline{x})
\end{aligned} \tag{1}
$$

$$
\texttt{build}^T\ f \ = \ f(Constrs\,T) \tag{2}
$$

$$
\texttt{cata}^T(f_1, \ldots, f_n)\ (\texttt{build}^T\ g) \ = \ g\ (f_1, \ldots, f_n) \tag{3}
$$

where $(f_1, \ldots, f_n) : \Pi(E_i(\alpha) \to \alpha)$ (viewed as the type $E(\alpha) \to \alpha$).

A catamorphism, $\texttt{cata}^T(f_1, \ldots, f_n)$, can be viewed as a function that replaces every constructor, $\mathbf{C}_i$, in a data structure with a corresponding function, f_i. For example by repeatedly applying equation 1:

$$
\begin{aligned}
& \texttt{cata}^{\text{List}}(n,c)\ (\mathbf{Cons}(7, \mathbf{Cons}(3, \mathbf{Nil}()))) \\
= \ & c\ (7, \texttt{cata}^{\text{List}}(n,c)\ (\mathbf{Cons}(3, \mathbf{Nil}()))) \\
= \ & c\ (7, c\ (3, \texttt{cata}^{\text{List}}(n,c)\ (\mathbf{Nil}()))) \\
= \ & c\ (7, c\ (3, n))
\end{aligned}
$$

A $\texttt{build}^T$ is applied to a function which abstracts over the constructors of the datatype T. Thus:

$$
\texttt{build}^{\text{List}}(\lambda(n,c)\ .\ c\ (7, c\ (3, n())))
$$

applies the function argument to the constructors of list, $(\mathbf{Nil}, \mathbf{Cons})$ (equation 2), and reconstructs the list:

$$
\mathbf{Cons}(7, \mathbf{Cons}(3, \mathbf{Nil}()))
$$

A more detailed description of build can be found in Section 6.

We assume renaming is done whenever there is any danger of name-capture, and for simplicity, we restrict ourselves to singly-recursive datatypes and functions (i.e. no mutual recursion), though we expect the algorithms to have a natural extension to the more general case. More significantly, we also restrict ourselves to ground data types, that is, data types which are not built using function space. Recent work by Hutton and Meijer may point the way to relaxing this restriction [MH95]. In a practical context, if these data types occur, then our algorithm simply makes no attempt to fuse them.

Finally, while the `build` form is presented as a part of the language, our intent is that it be used for internal purposes only and not exposed to a programmer. Our method for introducing `build` into a term guarantees the validity of a the `cata`-`build` equation above—in general the rule is not valid. If `build` was exposed to the programmer then a corresponding "cleanliness" check would be required: the type-rule of Gill, Launchbury and Peyton Jones serves this purpose, for example [GLPJ93].

4 Two-stage Fusion

As stated earlier, our fusion process proceeds in two separate phases. First individual function definitions are preprocessed in an attempt to re-express their definition in terms of a `build` and a catamorphism. Second, separate invocations of such functions are fused with one another using the one-step fusion rule (equation 3) in which `build`'s and `cata`'s cancel.

In reality, there is some interplay between the phases. As the `reverse` example will show later, it is often worth taking advantage of having already preprocessed functions used within the body of the function being preprocessed. But in the life-cycle of any given function, the two phases are distinct.

In principle, preparing for the first phase is simplicity itself. Suppose we had a function $foo = body : List\ int \to List\ int$, where *body* is a function-valued expression (usually of the form: $\lambda x\ .\ \ldots$) calling *foo* recursively. We may redefine *foo* as follows:

$$
\begin{aligned}
foo \ = \ & \lambda w\ .\ \texttt{build}^{\text{List}}(\lambda(n,c)\ .\ \texttt{cata}^{\text{List}}\ (n,c) \\
& \qquad (body\ (\texttt{cata}^{\text{List}}\ (\mathbf{Nil}, \mathbf{Cons})\ w)))
\end{aligned}
$$

There are two thing going on here. The first is that we apply a particular form of the identity function (namely $\texttt{cata}^{\text{List}}\ (\mathbf{Nil}, \mathbf{Cons})$, also called the *copy* function) to *body*'s argument. Our purpose is to attempt to fuse *body* with this catamorphism. If successful, we will have managed to reexpress the recursion in *body* as a catamorphism.

The second change to *foo* involves introducing $\texttt{build}^{\text{List}}$. The purpose of `build` is to enshroud a function which corresponds to a data structure (here a list) except that it is abstracted over the constructors of the data type. The abstraction is easily achieved using `cata`, since as described earlier cata replaces every constructor in a data structure with an associated function, Here the associated functions are the variables abstracting the constructors. Again, our goal is to fuse this use of `cata` into *body*, replacing any explicit "output constructors" with the parameters n and c.

4.1 Rewrite rules

The fusion algorithm can be expressed in terms of rewrite rules, except that in order to fuse catamorphisms, we have to allow the rule set to grow and shrink dynamically. The basic rewrite rules, called $\mathcal{R}_0$, are given in Figure 2. The syntax $b[v \mapsto e]$ denotes the (capture-free) substitution of e for free occurences of v in b.

The final rule performs one-step deforestation. As we noted in Section 3, we need some mechanism to guarantee its correctness. For our purposes, we guarantee correctness by limiting `build`-introduction to introduce a corresponding `cata` as in the example above (more details are in Section 6). Then, assuming that all rewrites preserve equality, any argument `g` to `build` must remain equal to a term of the form $\lambda\bar{c}\ .\ \mathtt{cata}\ \bar{c}\ e$, for some e. Thus

$$\begin{array}{ll} & \mathtt{cata}^T\ \bar{h}\ (\mathtt{build}\ g) \\ = & \mathtt{cata}^T\ \bar{h}\ (\mathtt{build}(\lambda\bar{c}\ .\ \mathtt{cata}^T\ \bar{c}\ e)) \\ = & \mathtt{cata}^T\ \bar{h}\ (\mathtt{cata}^T\ (\mathit{Constrs}(T))\ e) \\ = & \mathtt{cata}^T\ \bar{h}\ e \\ = & (\lambda\bar{c}\ .\ \mathtt{cata}^T\ \bar{c}\ e)\ \bar{h} \\ = & g\ \bar{h} \end{array}$$

so demonstrating the validity of the `cata`-`build` law.

It is often advantageous to extend the notation of a rewrite rule $\{lhs \rightarrow rhs\}$ to cover the case where both *lhs* and *rhs* are tuples. We introduce the notation $\Rightarrow$ for this case. We define $\{(x_1, \ldots, x_n)\ \Rightarrow\ (y_1, \ldots, y_n)\}$ to mean the set of rules extracted component-wise $\{x_1 \rightarrow y_1, \ldots, x_n \rightarrow y_n\}$.

We also extend the $\Rightarrow$ notation by allowing E functors over the tuples. Thus: $\{E_{\mathbf{C}}(g)(x_1, \ldots, x_n) \Rightarrow (y_1, \ldots, y_n)\}$ is equivalent to the set of rules $\{z_1\ \rightarrow\ y_1, \ldots, z_n\ \rightarrow\ y_n\}$ where z_i is some term depending on x_i, g, and $E_{\mathbf{C}}$, gained by expanding out the definition of $E_{\mathbf{C}}$ to give a vector of rules. For example

$$\begin{array}{ll} \multicolumn{2}{l}{\{E_{\mathrm{Cons}}(g)(x_1, x_2)\ \Rightarrow\ (y_1, y_2)\}} \\ = & \{(x_1, g\ x_2)\ \Rightarrow\ (y_1, y_2)\} \\ = & \{x_1 \rightarrow y_1,\ (g\ x_2) \rightarrow y_2\} \end{array}$$

4.2 Fusing Catamorphisms

The success of our technique depends critically on our ability to fuse catamorphisms. The algorithm for this is based on the *promotion theorem*, which describes when the composition of a (strict) function g with a `cata` can be expressed as another `cata` [Mal89, MFR91].

$$\frac{\forall \mathbf{C}:\ g\ (f_{\mathbf{C}}(y_1, \ldots, y_n))\ =\ h_{\mathbf{C}}(E_{\mathbf{C}}(g)(y_1, \ldots, y_n))}{g\ (\mathtt{cata}^T\ \bar{f}\ x)\ =\ \mathtt{cata}^T\ \bar{h}\ x}$$

Instantiating the theorem to lists gives:

$$\frac{\begin{array}{lcl} g\ (f_{\mathrm{Nil}}()) & = & h_{\mathrm{Nil}}() \\ g\ (f_{\mathrm{Cons}}(y_1, y_2)) & = & h_{\mathrm{Cons}}(y_1, g\ y_2) \end{array}}{g\ (\mathtt{cata}^{\mathrm{List}}\ (f_{\mathrm{Nil}}, f_{\mathrm{Cons}})\ x)\ =\ \mathtt{cata}^{\mathrm{List}}\ (h_{\mathrm{Nil}}, h_{\mathrm{Cons}})\ x}$$

The following example is helpful in understanding what the promotion theorem is doing. If the x in the theorem is taken to be the list $\mathbf{Cons}(7, \mathbf{Cons}(3, \mathbf{Nil}()))$, then we have:

$$\begin{array}{ll} & g\ (\mathtt{cata}^{\mathrm{List}}(f_{\mathrm{Nil}}, f_{\mathrm{Cons}})\ (\mathbf{Cons}(7, \mathbf{Cons}(3, \mathbf{Nil}())))) \\ = & g\ (f_{\mathrm{Cons}}(7, f_{\mathrm{Cons}}(3, f_{\mathrm{Nil}}()))) \end{array}$$

Applying the second hypothesis twice allows us to push g all the way to the end of the list, changing the f_{Cons}'s to h_{Cons}'s in the process.

$$\begin{array}{ll} = & g\ (f_{\mathrm{Cons}}(7, f_{\mathrm{Cons}}(3, f_{\mathrm{Nil}}()))) \\ = & h_{\mathrm{Cons}}(7, g\ (f_{\mathrm{Cons}}(3, f_{\mathrm{Nil}}()))) \\ = & h_{\mathrm{Cons}}(7, h_{\mathrm{Cons}}(3, g\ (f_{\mathrm{Nil}}()))) \end{array}$$

Now applying the first hypothesis removes g altogether:

$$\begin{array}{ll} = & h_{\mathrm{Cons}}(7, h_{\mathrm{Cons}}(3, g\ (f_{\mathrm{Nil}}()))) \\ = & h_{\mathrm{Cons}}(7, h_{\mathrm{Cons}}(3, h_{\mathrm{Nil}}())) \end{array}$$

But this is just

$$\mathtt{cata}^{\mathrm{List}}(h_{\mathrm{Nil}}, h_{\mathrm{Cons}})\ (\mathbf{Cons}(7, \mathbf{Cons}(3, \mathbf{Nil}())))$$

We intend to apply the promotion theorem as a left-to-right rewrite rule. In order to do this we must find a set of functions $h_{\mathbf{C}}$ which meet the conditions of the premise. In previous work, Sheard and Fegaras described a *fusion algorithm* which either computed the $h_{\mathbf{C}}$, or terminated with failure [SF93]. The challenge in computing the $h_{\mathbf{C}}$ is the presence of the $E_{\mathbf{C}}(g)$ term on the right-hand side of the premise (in the general case). In the list instance of the theorem this manifests itself as the g in the the call to h_{Cons} on the right-hand side of the second hypothesis. Without this term we would have an immediate definition of the $h_{\mathbf{C}}$.

The approach we take to generate the $h_{\mathbf{C}}$ functions is to introduce additional free variables, $\bar{z}$, and to extend the current set of rewrite rules with additional temporary rules which describe how to eliminate combinations of the old variables ($\bar{y}$) with g, in favor of the new free variables. If, at the end of rewriting, all the old y's have been eliminated then we have successfully discovered a definition for $h_{\mathbf{C}}$, otherwise failure is reported. See Section 4.3 and Figure 3 for a detailed example. Note that the additional rules are valid only for the body of the $h_{\mathbf{C}}$ function, and must be discarded once the rewriting of the body of $h_{\mathbf{C}}$ has terminated.

To make this formal we introduce the following sequent notation. We write $\mathcal{R}\ \vdash\ e\ \longrightarrow\ e'$ to mean that, using rule-set $\mathcal{R}$, the term e rewrites to e' (in one step). The notation $\mathcal{R}\ \vdash\ e \twoheadrightarrow e'$ is its reflexive, transitive closure. The complete rewrite system is given in Figure 3.

The first rule simply applies existing rewrite rules in any context. The second rule is the most interesting. It performs `cata`-fusion. If a term of the form $g\,(\mathtt{cata}^T(f_1, \ldots, f_n)\ x)$ is encountered then for each constructor in the data type T, an extended rewriting system is constructed and used to attempt to produce the body of the functions $h_{\mathbf{C}}$. Note that as far as the rewriting system is concerned, the vectors of new variables have to be treated as literals, and not as term-rewriting variables. We indicate this using bold font. Finally, if the result of any of these rewritings still contains any occurrence of the new free variables $\mathbf{y}$, then the premise of the rule fails, and `cata`-fusion is not performed.

4.3 Example

To see these rules in action, consider fusing the sum function with *map* when each are already expressed as catamorphisms (we will deal with explicit recursion in the next

$$
\mathcal{R}_0 = \{ \begin{array}{ll} (\lambda v \,.\, b)\; e & \rightarrow\; b[v \mapsto e], \quad \text{if } v \text{ linear in } b \\ (\lambda(v_1,\ldots,v_n) \,.\, b)\; (e_1,\ldots,e_n) & \rightarrow\; b[v_1 \mapsto e_1,\ldots,v_n \mapsto e_n], \quad \text{if } v_i \text{ linear in } e_i \\ \texttt{case}\; \mathbf{C}_i\; \bar{x}\; \texttt{of}\; \mathbf{C}_1 \bar{v}_1\; \texttt{->}\; e_1 \mid \ldots \mid \mathbf{C}_n \bar{v}_n\; \texttt{->}\; e_n & \rightarrow\; e_i[\bar{v}_i \mapsto \bar{x}], \quad \text{if } v_i \text{ linear in } e_i \\ \texttt{cata}^T(f_1,\ldots,f_n)\; (C_i\; \bar{x}) & \rightarrow\; f_i\; (E_i\; (\texttt{cata}^T(f_1,\ldots,f_n))\; \bar{x}), \\ \texttt{cata}^T(f_1,\ldots,f_n)\; (\texttt{build}^T(g)) & \rightarrow\; g\; (f_1,\ldots,f_n) \end{array} \}
$$

Figure 2: Basic Set of Rewrite Rules

$$
\mathcal{R} \cup \{l \rightarrow r\} \;\vdash\; P[l] \longrightarrow P[r]
$$

$$
\frac{\forall \mathbf{C} :\; \mathcal{R} \cup \{E_{\mathbf{C}}(g)\bar{\mathbf{y}} \Rightarrow \bar{\mathbf{z}}\} \;\vdash\; \lambda\bar{\mathbf{z}} \,.\, g\; (f_{\mathbf{C}}\; \bar{\mathbf{y}}) \twoheadrightarrow h_{\mathbf{C}}}{\mathcal{R} \;\vdash\; g\; (\texttt{cata}^T\; \bar{f}\; x) \longrightarrow \texttt{cata}^T\; \bar{h}\; x} \quad \mathbf{y}_i \notin FV(\bar{h})
$$

Figure 3: Rewrite Algorithm

section). Their definitions are:

$$
\begin{array}{lll} \mathit{sum} & = & \texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,\; (+)) \\ \mathit{map}\; f & = & \texttt{cata}^{\text{List}}\; (\mathbf{Nil},\; \lambda(x,w)\,.\,\mathbf{Cons}(f\; x, w)) \end{array}
$$

Now, to enable rewiting these to a single catamorphism using the rule

$$
\mathcal{R}_0 \;\vdash\; \mathit{sum} \circ (\mathit{map}\; f) \longrightarrow \texttt{cata}^{\text{List}}\; (h_{\text{Nil}}, h_{\text{Cons}})
$$

we have to successfully derive h_{Nil} and h_{Cons} using extended rewrite systems. In the case of **Nil** we have

$$
\begin{array}{lll} \mathcal{R}_{\text{Nil}} & = & \mathcal{R}_0 \cup \{E_{\text{Nil}}(\mathit{sum})() \Rightarrow ()\} \\ & = & \mathcal{R}_0 \cup \{() \Rightarrow ()\} \\ & = & \mathcal{R}_0 \end{array}
$$

and for **Cons** we have,

$$
\begin{array}{ll} \mathcal{R}_{\text{Cons}} & \\ = & \mathcal{R}_0 \cup \{E_{\text{Cons}}(\mathit{sum})(\mathbf{y}_1,\mathbf{y}_2) \Rightarrow (\mathbf{z}_1,\mathbf{z}_2)\} \\ = & \mathcal{R}_0 \cup \{(\mathbf{y}_1, \mathit{sum}\; \mathbf{y}_2) \Rightarrow (\mathbf{z}_1,\mathbf{z}_2)\} \\ = & \mathcal{R}_0 \cup \{(\mathbf{y}_1, \texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,\; (+))\; \mathbf{y}_2) \Rightarrow (\mathbf{z}_1,\mathbf{z}_2)\} \\ = & \mathcal{R}_0 \cup \{\mathbf{y}_1 \rightarrow \mathbf{z}_1, \texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,\; (+))\; \mathbf{y}_2 \rightarrow \mathbf{z}_2\} \end{array}
$$

Rewriting the **Nil** case is immediate:

$$
\mathcal{R}_{\text{Nil}} \;\vdash\; \lambda()\,.\,\texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,(+))\; (\mathbf{Nil}\; ()) \longrightarrow \lambda()\,.\,0
$$

In the **Cons** case it proceeds as follows:

$$
\begin{array}{ll} \mathcal{R}_{\text{Cons}} & \\ \vdash & \lambda(\mathbf{z}_1,\mathbf{z}_2)\,.\,\texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,(+)) \\ & \quad ((\lambda(x,w)\,.\,\mathbf{Cons}(f\; x,w))\; (\mathbf{y}_1,\mathbf{y}_2)) \\ \longrightarrow & \lambda(\mathbf{z}_1,\mathbf{z}_2)\,.\,\texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,(+))\; (\mathbf{Cons}(f\; \mathbf{y}_1,\mathbf{y}_2)) \\ \longrightarrow & \lambda(\mathbf{z}_1,\mathbf{z}_2)\,.\,(+)\; (f\; \mathbf{y}_1, \texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,(+))\; \mathbf{y}_2) \\ \longrightarrow & \lambda(\mathbf{z}_1,\mathbf{z}_2)\,.\,(+)\; (f\; \mathbf{z}_1,\mathbf{z}_2) \end{array}
$$

In both cases, the results have eliminated the $\mathbf{y}$'s, so rewriting of the main term may proceed resulting in $\mathit{sum} \circ (\mathit{map}\; f)$ equal to

$$
\texttt{cata}^{\text{List}}\; (\lambda()\,.\,0,\; \lambda(z_1,z_2)\,.\,f\; z_1 + z_2)
$$

5 Expressing Recursive Functions as Catamorphisms.

The `cata`-fusion fusion algorithm provides the mechanism for computing the equivalent `cata` for a function. For a *unary* function, $g : T \rightarrow \alpha$, first express the identity function as a catamorphism over T, compose it with g on the left, and then apply the fusion algorithm. The identity function at type T is easily expressed by using the constructors as arguments to the catamorphism operator: $\texttt{cata}^T(\mathbf{C}_1,\ldots,\mathbf{C}_n)$.

For non-unary recursive function, f, there are some additional difficulties. To which argument should the catamorphism be applied? Which subexpression of f's body corresponds to tying the knot of the recursive cycle? We have developed heuristics to address these difficulties which seem to work well in a wide variety of cases.

In essense, we collect the *explicit* arguments to a function (those given by literal outer lambdas), and then look for the outermost `case` expression which we expect to be over one of the explicit arguments. If the function is written in some other form then we give up. This may seem restrictive, but it successfully catches all definitions written using pattern-matching arguments.

More formally, given a definition $f = \mathit{body}$, we express the structure of *body* in the form

$$
\mathit{body} = \lambda x_1 \ldots \lambda x_n \,.\, Q[\texttt{case}\; x_k\; \texttt{of}\; \mathit{pats}]
$$

where Q is a non-`case` context defined by,

$$
\begin{array}{rcl} Q & ::= & [\,] \mid \lambda x\,.\,Q \mid Q\; Q' \mid Q'\; Q \\ & & \mid (Q,Q') \mid (Q',Q) \mid \texttt{build}\; Q \\ Q' & ::= & n \mid v \mid \mathbf{C} \mid \lambda x\,.\,Q' \mid Q'\; Q' \\ & & \mid (Q',Q') \mid \texttt{build}\; Q' \end{array}
$$

and where $x_k, f \notin FV(Q[\,])$ and $x_k \notin FV(\mathit{pats})$. The condition that f does not occur in the free variables of the context ensures that any recursive call to f is 'guarded' by the case statement. The restriction on the x_k rejects a case of full primitive recursion.

If *body* does not have this structure we give up, otherwise we generate a new function definition $f^{\#}$ as follows:

$$
\begin{array}{lll} f & = & \lambda x_1..\lambda x_n\,.\,Q[f^{\#}\; x_k\; \bar{v}] \\ f^{\#} & = & \lambda x_k\,.\,\lambda\bar{v}\,.\,\texttt{case}\; x_k\; \texttt{of}\; \mathit{pats} \end{array}
$$

where $\bar{v} = FV(\mathit{pats}) - FV(\mathit{body}) - \{x_k\}$.

The function f is a "wrapper" and $f^{\#}$ a "worker" in the sense of Peyton Jones and Launchbury [PJL91]. Wrapper functions are freely unfolded, so we substitute the new body of f in the definition of $f^{\#}$.

We now have a recursive function with an outer `case` over the first argument. We attempt to fuse this definition of $f^{\#}$ with the copy function to obtain a catamorphic version of $f^{\#}$. If successful, we may substitute (the now non-recursive) definition of $f^{\#}$ back into the new defintion of f, so finally obtaining a definition of f as a catamorphism.

To see this in practice, consider the example of `map`.

$$\begin{array}{l} map = \lambda f \;.\; \lambda x \;. \\ \quad \texttt{case}\ x\ \texttt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ \mathbf{Nil}() \\ \quad\ \mid\ \ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathbf{Cons}(f\ z,\ map\ f\ zs) \end{array}$$

After breaking the definition into two components we have,

$$\begin{array}{l} map = \lambda f \;.\; \lambda x \;.\; map^{\#}\ x\ f \\ map^{\#} = \lambda x \;.\; \lambda f \;. \\ \quad \texttt{case}\ x\ \texttt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ \mathbf{Nil}() \\ \quad\ \mid\ \ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathbf{Cons}(f\ z,\ map\ f\ zs) \end{array}$$

Then, by unfolding the new definition for map we obtain the following recursive definition:

$$\begin{array}{l} map^{\#} = \lambda x \;.\; \lambda f \;. \\ \quad \texttt{case}\ x\ \texttt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ \mathbf{Nil}() \\ \quad\ \mid\ \ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathbf{Cons}(f\ z,\ map^{\#}\ zs\ f) \end{array}$$

This function's body is a case expression over it's first argument and is readily fused with $\texttt{cata}^{\mathrm{List}}\ (\mathbf{Nil}, \mathbf{Cons})$ to obtain a definition of the form:

$$\lambda x \;.\; \lambda f \;.\; \texttt{cata}^{\mathrm{List}}\ (h_{\mathrm{Nil}}, h_{\mathrm{Cons}})\ x$$

Note that, by construction, the function $map^{\#}$ is strict in its first argument (it is about to perform a case analysis) so the fusion theorem applies. To see the fusion in action, consider the rewite rules. Once again $\mathcal{R}_{\mathrm{Nil}} = \mathcal{R}_0$ and this time $\mathcal{R}_{\mathrm{Cons}} = \mathcal{R}_0 \cup \{\mathbf{y}_1 \to \mathbf{z}_1, map^{\#}\ \mathbf{y}_2 \to \mathbf{z}_2\}$. Then h_{Nil} and h_{Cons} are computed by:

$$\begin{array}{ll} \mathcal{R}_{\mathrm{Nil}} & \vdash\ \lambda() \;.\; map^{\#}(\mathbf{Nil}\ ()) \longrightarrow h_{\mathrm{Nil}} \\ \mathcal{R}_{\mathrm{Cons}} & \vdash\ \lambda(z_1, z_2) \;.\; map^{\#}(\mathbf{Cons}(\mathbf{y}_1, \mathbf{y}_2)) \longrightarrow h_{\mathrm{Cons}} \end{array}$$

under the condition that the $\mathbf{y}$'s are eliminated from the resulting terms.

Note that our implementation actually substitutes the body of $map^{\#}$ in the above. This is the only place we use information about the definition of $map^{\#}$ and this guarantees that it is unfolded exactly once.

By applying the various rules we obtain:

$$\begin{array}{lcl} h_{\mathrm{Nil}} & = & \lambda() \;.\; \lambda f \;.\; \mathbf{Nil}() \\ h_{\mathrm{Cons}} & = & \lambda(\mathbf{z}_1, \mathbf{z}_2) \;.\; \lambda f \;.\; \mathbf{Cons}(f\ \mathbf{z}_1,\ \mathbf{z}_2\ f) \end{array}$$

Finally, substituting $map^{\#}$ back into the definition of map gives a new definition of map as a catamorphism.

$$map = \lambda f \;.\; \lambda x \;.\; \texttt{cata}^{\mathrm{List}}\ (h_{\mathrm{Nil}}, h_{\mathrm{Cons}})\ x\ f$$

5.1 Static Parameters, Tuples and Irrelevant Cases

The astute reader will notice that the previous example is actually a *higher-order* catamorphism: each term is a function waiting to be applied to f. While this generality is sometimes essential, it is not useful in this case. The value inherited by the recursive applications of `cata` are all f, they are identical to each other.

It is possible to make an improvement to the algorithm above whereby static parameters are not inherited between the recursive levels of a catamorphism. If we had done this in the case of map we would have obtained the following:

$$\begin{array}{l} map = \lambda f \;.\; \lambda x \;. \\ \quad \texttt{cata}^{\mathrm{List}}\ (\mathbf{Nil}, \lambda(z_1, z_2) \;.\; \mathbf{Cons}(f\ z_1,\ z_2))\ x \end{array}$$

which is as good as it gets. The higher-orderness of the original definition comes from passing f explicitly as an argument to $map^{\#}$. If we reduce the arguments to the worker function ($map^{\#}$ in this case) by omitting those arguments which are unchanged in the recursive calls, then in this and similar cases, the catamorphism becomes first order.

Note, however, that the worker is not now a function in it's own right, but should be viewed as a definition local to the wrapper as it contains extra free variables. However, once the catamorphic definition is substituted back into the definition of the worker, the free variables are captured once again, and all is well.

Another obvious extension is to allow each of the x_i to be tuples. In addition, we will see a couple of examples at the end of the paper where we also pass over a `case` when the argument to the case is other than a variable.

As with the static parameter optimisation, this additional generality is useful but not critical. As it complicates notation we left it out of the algorithm definition. Conversely, as none of these introduce any additional challenges we will feel free to assume from now on that static parameters, tuples and irrelevant cases are handled sensibly.

5.2 Linear Reverse

The linear version of the reverse function provides a good example of a changing recursive parameter, leading to an essential use of a higher-order catamorphism.

$$\begin{array}{l} lreverse = \lambda x \;.\; \lambda w \;. \\ \quad \texttt{case}\ x\ \texttt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ w \\ \quad\ \mid\ \ \mathbf{Cons}(z, zs)\ \texttt{->}\ lreverse\ zs\ (\mathbf{Cons}(z, w)) \end{array}$$

The definition is already in the form constructed by the context machinery (i.e. $lreverse^{\#}$ would be identical to $lreverse$), so we are ready to fuse $lreverse$ to $\texttt{cata}^{\mathrm{List}}\ (\mathbf{Nil}, \mathbf{Cons})$. As before, $\mathcal{R}_{\mathrm{Nil}} = \mathcal{R}_0$.

$$\mathcal{R}_{\mathrm{Nil}}\ \vdash\ \lambda() \;.\; \texttt{lreverse}(\mathbf{Nil}\ ()) \longrightarrow \lambda() \;.\; \lambda w \;.\; w$$

so $h_{\mathrm{Nil}} = \lambda() \;.\; \lambda w \;.\; w$.

In the **Cons** case we have

$$\begin{array}{ll} \mathcal{R}_{\mathrm{Cons}} & \\ \quad = & \mathcal{R}_0 \cup \{E_{\mathrm{Cons}}(\texttt{lreverse})(\mathbf{y}_1, \mathbf{y}_2) \Rightarrow (\mathbf{z}_1, \mathbf{z}_2)\} \\ \quad = & \mathcal{R}_0 \cup \{(\mathbf{y}_1, \texttt{lreverse}\ \mathbf{y}_2) \Rightarrow (\mathbf{z}_1, \mathbf{z}_2)\} \\ \quad = & \mathcal{R}_0 \cup \{\mathbf{y}_1 \to \mathbf{z}_1, \texttt{lreverse}\ \mathbf{y}_2 \to \mathbf{z}_2\} \end{array}$$

and the rewriting proceeds as follows

$$\begin{array}{lll} \mathcal{R}_{\mathrm{Cons}} & \vdash & \lambda(\mathbf{z}_1, \mathbf{z}_2) \;.\; \texttt{lreverse}(\mathbf{Cons}(\mathbf{y}_1, \mathbf{y}_2)) \\ & \longrightarrow & \lambda(\mathbf{z}_1, \mathbf{z}_2) \;.\; \lambda w \;.\; \texttt{lreverse}\ \mathbf{y}_2\ (\mathbf{Cons}(\mathbf{y}_1, w)) \\ & \longrightarrow & \lambda(\mathbf{z}_1, \mathbf{z}_2) \;.\; \lambda w \;.\; \mathbf{z}_2\ (\mathbf{Cons}(\mathbf{z}_1, w)) \end{array}$$

The $\mathbf{y}$'s have been eliminated from the resulting terms, so we may rewrite

$$\begin{array}{lll} \mathcal{R}_0 & \vdash & lreverse \circ \texttt{cata}^{\mathrm{List}}\ (\mathbf{Nil}, \mathbf{Cons}) \\ & \longrightarrow & \texttt{cata}^{\mathrm{List}}\ (\lambda() \;.\; \lambda w \;.\; w, \\ & & \qquad \lambda(\mathbf{z}_1, \mathbf{z}_2) \;.\; \lambda w \;.\; \mathbf{z}_2\ (\mathbf{Cons}(\mathbf{z}_1, w))) \end{array}$$

So, finally, we now define,

$$\begin{array}{l} \mathit{lreverse} = \mathtt{cata}^{\mathrm{List}}\ (\lambda()\ .\ \lambda w\ .\ w, \\ \qquad\qquad\qquad\qquad \lambda(\mathbf{z}_1, \mathbf{z}_2)\ .\ \lambda w\ .\ \mathbf{z}_2\ (\mathbf{Cons}(\mathbf{z}_1, w))) \end{array}$$

6 Expressing Terms as Builds

The purpose of `build` is to allow us to represent a term of some data type as a function parameterised over the output constructors. So, for example, rather than work with the literal list: $\mathbf{Cons}(1, \mathbf{Cons}(2, \mathbf{Cons}(3, \mathbf{Nil}())))$ we work with the function $\lambda(n, c)\ .\ c\ (1,\ c\ (2,\ c\ (3, n())))$. But now the expression is no longer a list! It's a function. So (following [GLPJ93]) we introduce a construct for each data type T called $\mathtt{build}^T$ defined by $\mathtt{build}^T\ g = g\ (\mathit{Constrs}(T))$. Thus, reducing the term

$$\mathtt{build}^{\mathrm{List}}\ (\lambda(n, c)\ .\ c\ (1,\ c\ (2,\ c\ (3, n()))))$$

simply reconstructs the list $\mathbf{Cons}(1, \mathbf{Cons}(2, \mathbf{Cons}(3, \mathbf{Nil}())))$.

Of course, the purpose of introducing `build`'s is not simply to reduce them away again! Rather the purpose is to enable the `build` to cancel with an enclosing catamorphism (as performed by the $\mathtt{cata} - \mathtt{build}$ law), and so remove the need to construct an intermediate structure.

We introduce appropriate `build`s to an expression by applying the syntax-to-syntax translation $\mathcal{B}$ defined as follows:

$$\begin{array}{lcll} \mathcal{B}\ x & = & x & [x \text{ is a variable}] \\ \mathcal{B}\ (\lambda x\ .\ e) & = & \lambda x\ .\ \mathcal{B}\ e & \\ \mathcal{B}\ (e_1, e_2) & = & (\mathcal{B}\ e_1,\ \mathcal{B}\ e_2) & \\ \textbf{or else} & & & \\ \mathcal{B}\ e & = & \mathtt{build}^T(\lambda \bar{p}\ .\ \mathtt{cata}^T\ \bar{p}\ e), & \textbf{if } e : T\ t \\ & = & e, & \textbf{otherwise} \end{array}$$

Consider the definition of a function such as *append*.

$$\begin{array}{l} \mathit{append} = \lambda x\ .\ \lambda y\ . \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ y \\ \quad \mid\ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathbf{Cons}(z, \mathit{append}\ zs\ y) \end{array}$$

Applying $\mathcal{B}$ to it's body gives:

$$\begin{array}{l} \mathit{append} = \lambda x\ .\ \lambda y\ .\ \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c)\ .\ \mathtt{cata}^{\mathrm{List}}\ (n, c) \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ y \\ \quad \mid\ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathbf{Cons}(z, \mathit{append}\ zs\ y)\ ) \end{array}$$

Given that our aim is to remove unnecessary intermediate data structures, then the `build` introduction strategy appears to be a bad idea, as we generate new intermediate data structures. However, these intermediates only exist temporarily—they will be fused with other components of the function definition.

Typically, the newly introduced `build` and `cata` come to rest immediately surrounding an existing `case` expression (as in the *append* example above). There are then two distinct ways to proceed:

1. either we may distribute the new `cata` across the `case` expression, and then convert the whole term into a single catamorphism using the techniques of Section 5; or

2. we may turn the whole term into a catamorphism immediately, and only then fuse the outer catamorphism to the new one (this latter fusion is typically a higher-order fusion, described in Section 7).

The second of these sometimes fails (none of the fusion steps go through), but when it succeeds it seems to give better results than the first. In constrast, the first seems the more robust: on the examples we have tried it succeeds whenever the second does. On many common examples, both methods are completely equivalent: they both succeed and produce exactly the same term. Such is the case with *append*. Applying the algorithm gives the result

$$\begin{array}{l} \mathit{append} = \lambda x\ .\ \lambda y\ . \\ \quad \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c)\ .\ \mathtt{cata}^{\mathrm{List}}\ (\lambda()\ .\ \mathtt{cata}^{\mathrm{List}}\ (n, c)\ y, c)\ x) \end{array}$$

which is equivalent to the version given in [GLPJ93].

The difference between the two methods shows up in *reverse*, where the second method fails, and in its linear counterpart *lreverse*, where the second is superior.

6.1 Reverse example

We define *reverse* as usual.

$$\begin{array}{l} \mathit{reverse} = \lambda x\ . \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ \mathbf{Nil}() \\ \quad \mid\ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathit{append}\ (\mathit{reverse}\ zs)\ (\mathbf{Cons}(z, \mathbf{Nil}())) \end{array}$$

Pushing the `build` in place gives:

$$\begin{array}{l} \mathit{reverse} = \lambda x\ .\ \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c)\ .\ \mathtt{cata}^{\mathrm{List}}\ (n, c) \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ \mathbf{Nil}() \\ \quad \mid\ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathit{append}\ (\mathit{reverse}\ zs)\ (\mathbf{Cons}(z, \mathbf{Nil}()))) \end{array}$$

Now, following the first strategy from Section 6, we distribute the `cata` across the `case`. After some simple rewriting we get the term:

$$\begin{array}{l} \mathit{reverse} = \lambda x\ .\ \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c)\ . \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ n() \\ \quad \mid\ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathtt{cata}^{\mathrm{List}}\ (n, c) \\ \qquad\quad (\mathit{append}\ (\mathit{reverse}\ zs)\ (\mathbf{Cons}(z, \mathbf{Nil}())))) \end{array}$$

To turn this into a single catamorphism we proceed as in Section 5. We express *reverse* as a wrapper and a worker:

$$\begin{array}{l} \mathit{reverse} = \lambda x\ .\ \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c)\ .\ \mathit{reverse}^{\#}\ x\ (n, c)) \\ \mathit{reverse}^{\#} = \lambda x\ .\ \lambda(n, c)\ . \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ n() \\ \quad \mid\ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathtt{cata}^{\mathrm{List}}\ (n, c) \\ \qquad\quad (\mathit{append}\ (\mathit{reverse}\ zs)\ (\mathbf{Cons}(z, \mathbf{Nil}()))) \end{array}$$

and unfolding *reverse* in $\mathit{reverse}^{\#}$ gives

$$\begin{array}{l} \mathit{reverse}^{\#} = \lambda x\ .\ \lambda(n, c)\ . \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ n() \\ \quad \mid\ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathtt{cata}^{|Sublist}\ (n, c) \\ \qquad\quad (\mathit{append}\ (\mathtt{build}^{\mathrm{List}}\ (\mathit{reverse}^{\#}\ zs))\ (\mathbf{Cons}(z, \mathbf{Nil}()))) \end{array}$$

Recall that the `build`-`cata` form of *append* is,

$$\begin{array}{l} \mathit{append} = \lambda x\ .\ \lambda y\ . \\ \quad \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c)\ .\ \mathtt{cata}^{\mathrm{List}}\ (\lambda()\ .\ \mathtt{cata}^{\mathrm{List}}\ (n, c)\ y,\ c)\ x) \end{array}$$

$$\frac{\forall \mathbf{C}: \quad E_{\mathbf{C}}(\lambda s \,.\, g \circ s)\ \bar{s} = E_{\mathbf{C}}(\lambda r \,.\, r \circ k)\ \bar{r} \quad \Rightarrow \quad g \circ (f_{\mathbf{C}}\ \bar{s}) = (h_{\mathbf{C}}\ \bar{r}) \circ k}{g\ (\mathtt{cata}^T\ \bar{f}\ x\ w)\ =\ \mathtt{cata}^T\ \bar{h}\ x\ (k\ w)}\ g \text{ strict}$$

Figure 4: The second order fusion theorem

$$\frac{\forall \mathbf{C}: \quad \mathcal{R} \cup \{E_{\mathbf{C}}(\lambda s \,.\, g \circ s)\bar{\mathrm{s}} \Rightarrow E_{\mathbf{C}}(\lambda r \,.\, r \circ g)\ \bar{\mathbf{r}}\} \cup \{g\ \mathbf{y} \to \mathbf{z}\}\ \vdash\ \lambda \bar{\mathbf{r}} \,.\, \lambda \mathbf{z} \,.\, g\ (f_{\mathbf{C}}\ \bar{\mathrm{s}}\ \mathbf{y}) \longrightarrow\!\!\!\!\!\rightarrow h_{\mathbf{C}}}{\mathcal{R}\ \vdash\ g\ (\mathtt{cata}^T\ \bar{f}\ x\ w) \longrightarrow \mathtt{cata}^T\ \bar{h}\ x\ (g\ w)}\ \mathrm{s}_i, \mathbf{y} \notin FV(\bar{h})$$

Figure 5: Rewrite for 2^{nd} order Cata-Fusion

Substituting this, and performing two `cata`-`build` reductions gives

$$\begin{array}{l} \mathit{reverse}^{\#} = \lambda x \,.\, \lambda(n, c) \,. \\ \quad \mathtt{case}\ x\ \mathtt{of} \\ \qquad \mathbf{Nil}()\ \texttt{->}\ n() \\ \quad \mid\ \ \mathbf{Cons}(z, zs)\ \texttt{->}\ \mathit{reverse}^{\#}\ zs\ (c\ (z, n\ ()),\ c) \end{array}$$

Now we are in a position to turn this recursive definition into a catamorphism. Assuming we optimise for static parameters (the c), we get the following higher-order catamorphism.

$$\begin{array}{l} \mathit{reverse}^{\#} = \lambda x \,.\, \lambda(n, c) \,. \\ \quad \mathtt{cata}^{\mathrm{List}}\ (\lambda() \,.\, \lambda w \,.\, w, \lambda(z_1, z_2) \,.\, \lambda w \,.\, z_2\ (c\ (z_1, w)))\ x \end{array}$$

This can be substituted back into the definition of *reverse*, giving:

$$\begin{array}{l} \mathit{reverse} = \lambda x \,.\, \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c) \,. \\ \quad \mathtt{cata}^{\mathrm{List}}\ (\lambda() \,.\, \lambda w \,.\, w, \lambda(z_1, z_2) \,.\, \lambda w \,.\, z_2\ (c\ (z_1, w)))\ x) \end{array}$$

Interestingly, the fusion we performed with *append* has turned the original quadratic definition of *reverse* into a linear version (if the `build` and the `cata` of the new definition of reverse are expanded, then we obtain exactly the usual linear version). The reason for this is that the worker $\mathit{reverse}^{\#}$ was abstracted over the tail of its list (the n parameter)—an abstraction induced by the introduction of `build`.

If we were to carry the same program through but *starting* with the linear version of reverse, then the result is not quite so good. We obtain the result

$$\begin{array}{l} \mathit{lreverse} = \lambda x \,.\, \lambda w \,.\, \mathtt{build}^{\mathrm{List}}\ (\lambda(n, c) \,. \\ \quad \mathtt{cata}^{\mathrm{List}}\ (\lambda() \,.\, \lambda u \,.\, \mathtt{cata}^{\mathrm{List}}\ (n, c)\ u, \\ \qquad\qquad \lambda(z_1, z_2) \,.\, \lambda u \,.\, z_2\ (\mathbf{Cons}(z_1, u)))\ x\ w) \end{array}$$

where a true intermediate list is inherited and constructed, and only at the end it is abstracted over n and c.

In order to avoid this, we need to adopt method 2 above, in which the `cata` is not distributed across the `case`, but is fused with the result of turning the inner term into a `cata`. This second fusion is typically second-order, which we address now.

7 Second-Order Cata-Fusion

A second order catamorphism, $\mathtt{cata}^T\ \bar{f}\ x$, is a catamorphism which traverses the structure x and constructs a function. Assuming we use the static parameter optimization, higher-order catamorphisms only arise when the recursive invocations of the catamorphism really need an inherited attribute from above. For example, iterative reverse works by passing the reversed front of the list as an inherited attribute. At each level this is augmented, and at the end the accumulated list is returned.

We need a fusion algorithm to handle such higher-order catamorphisms. Once again, the algorithm is derived from a fundamental theorem found in Figure 4. The theorem may be proved by an easy fixed-point induction.

Consider an instantiation of this theorem in the case of lists.

$$\frac{\begin{array}{c} g \circ f_{\mathrm{Nil}}() = h_{\mathrm{Nil}}() \circ k \\ \left(\begin{array}{l} (s_1, g \circ s_2) = (r_1, r_2 \circ k) \quad \Rightarrow \\ \quad g\ (f_{\mathrm{Cons}}(s_1, s_2)\ a) = h_{\mathrm{Cons}}(r_1, r_2)\ (k\ a) \end{array} \right) \end{array}}{\begin{array}{l} g\ (\mathtt{cata}^{\mathrm{List}}\ (f_{\mathrm{Nil}}, f_{\mathrm{Cons}})\ x\ w)\ = \\ \qquad \mathtt{cata}^{\mathrm{List}}\ (h_{\mathrm{Nil}}, h_{\mathrm{Cons}})\ x\ (k\ w) \end{array}}$$

Once again, we intend to interpret this law as a left-to-right rewrite rule. This is given in Figure 5, with the difference that the *same* function g is used both on the left and right (i.e. k is instantiated to g). This is less general than the theorem allows, but seems to be sufficient for the cases we have seen. The more general case poses the problem of generating an appropriate k during rewriting.

7.1 Example

As an example, consider fusing the function $\mathtt{cata}^{\mathrm{List}}\ (n, c)$ (where n and c are variables) to the iterative reverse function when the latter is expressed as a second order catamorphism. This is exactly the sort of situation which must succeed for method 2 of Section 6 to work.

From Section 5 we had

$$\begin{array}{l} \mathit{lreverse} = \mathtt{cata}^{\mathrm{List}}\ (\lambda() \,.\, \lambda w \,.\, w, \\ \qquad\qquad\qquad \lambda(z_1, z_2) \,.\, \lambda w \,.\, z_2\ (\mathbf{Cons}(z_1, w))) \end{array}$$

The fusion we want to perform is

$$\mathtt{cata}^{\mathrm{List}}\ (n, c)\ (\mathit{lreverse}\ x\ (\mathbf{Nil}\ ()))$$

We calculate the additional rules in each of the two cases. For **Nil** the additional rules are

$$\begin{array}{rl} & \{E_{\mathrm{Nil}}(\lambda s \,.\, \mathtt{cata}^{\mathrm{List}}\ (\mathbf{n}, \mathbf{c}) \circ s)\ () \\ & \qquad \Rightarrow E_{\mathrm{Nil}}(\lambda r \,.\, r \circ \mathtt{cata}^{\mathrm{List}}\ (\mathbf{n}, \mathbf{c}))()\} \\ & \quad \cup \{\mathtt{cata}^{\mathrm{List}}\ (\mathbf{n}, \mathbf{c})\ \mathbf{y} \to \mathbf{z}\} \\ = & \{() \to ()\} \cup \{\mathtt{cata}^{\mathrm{List}}\ (\mathbf{n}, \mathbf{c})\ \mathbf{y} \to \mathbf{z}\} \\ = & \{\mathtt{cata}^{\mathrm{List}}\ (\mathbf{n}, \mathbf{c})\ \mathbf{y} \to \mathbf{z}\} \end{array}$$

and for **Cons** we have (writing $\texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})$ as g for brevity),

$$\begin{aligned}
&\{E_{\text{Cons}}(\lambda s\ .\ g \circ s)\,(\mathrm{s}_1,\mathrm{s}_2) \Rightarrow E_{\text{Cons}}(\lambda r\ .\ r \circ g)(\mathbf{r}_1,\mathbf{r}_2)\}\\
&\quad \cup \{g\ \mathbf{y} \rightarrow \mathbf{z}\}\\
= \ &\{(\mathrm{s}_1,(\lambda s\ .\ g \circ s)\ \mathrm{s}_2) \Rightarrow (\mathbf{r}_1,(\lambda r\ .\ r \circ g)\ \mathbf{r}_2)\}\\
&\quad \cup \{g\ \mathbf{y} \rightarrow \mathbf{z}\}\\
= \ &\{\mathrm{s}_1 \rightarrow \mathbf{r1},\ g \circ \mathrm{s}_2 \rightarrow \mathbf{r}_2 \circ g,\ g\ \mathbf{y} \rightarrow \mathbf{z}\}\\
= \ &\{\mathrm{s}_1 \rightarrow \mathbf{r1},\ g\ (\mathrm{s}_2\ x) \rightarrow \mathbf{r}_2\ (g\ x),\ g\ \mathbf{y} \rightarrow \mathbf{z}\}
\end{aligned}$$

So the additional rules for **Cons** are:

$$\begin{aligned}
&\{\mathrm{s}_1 \rightarrow \mathbf{r1},\\
&\ \texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ (\mathrm{s}_2\ x) \rightarrow \mathbf{r}_2\ (\texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ x),\\
&\ \texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ \mathbf{y} \rightarrow \mathbf{z}\}
\end{aligned}$$

Performing the rewriting for h_{Cons} and h_{Nil} we obtain the fused version of $\texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ (\mathit{lreverse}\ x\ (\textbf{Nil}()))$ as follows:

$$\begin{aligned}
\mathcal{R}_{\text{Nil}} \quad &\vdash \quad \lambda()\ .\ \lambda \mathrm{z}\ .\ \texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ ((\lambda()\ .\ \lambda w\ .\ w)\ ()\ \mathbf{y})\\
&\longrightarrow \quad \lambda()\ .\ \lambda \mathrm{z}\ .\ \texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ \mathbf{y}\\
&\longrightarrow \quad \lambda()\ .\ \lambda \mathrm{z}\ .\ \mathrm{z}
\end{aligned}$$

and in the **Cons** case:

$$\begin{aligned}
&\mathcal{R}_{\text{Cons}}\\
&\quad \vdash \quad \lambda(\mathbf{r}_1,\mathbf{r}_2)\ .\ \lambda \mathrm{z}\ .\ \texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\\
&\qquad\qquad ((\lambda(z_1,z_2)\ .\ \lambda u\ .\ z_2\ (\textbf{Cons}(z_1,u)))\ (\mathrm{s}_1,\mathrm{s}_2)\ \mathbf{y})\\
&\longrightarrow \quad \lambda(\mathbf{r}_1,\mathbf{r}_2)\ .\ \lambda \mathrm{z}\ .\ \texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ (\mathrm{s}_2\ (\textbf{Cons}(\mathrm{s}_1,\mathbf{y})))\\
&\longrightarrow \quad \lambda(\mathbf{r}_1,\mathbf{r}_2)\ .\ \lambda \mathrm{z}\ .\ \mathbf{r}_2\ (\texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ (\textbf{Cons}(\mathrm{s}_1,\mathbf{y})))\\
&\longrightarrow \quad \lambda(\mathbf{r}_1,\mathbf{r}_2)\ .\ \lambda \mathrm{z}\ .\ \mathbf{r}_2\ (\mathbf{c}\ (\mathrm{s}_1,\texttt{cata}^{\text{List}}\ (\mathbf{n},\mathbf{c})\ \mathbf{y})))\\
&\longrightarrow \quad \lambda(\mathbf{r}_1,\mathbf{r}_2)\ .\ \lambda \mathrm{z}\ .\ \mathbf{r}_2\ (\mathbf{c}\ (\mathrm{s}_1,\mathrm{z})))\\
&\longrightarrow \quad \lambda(\mathbf{r}_1,\mathbf{r}_2)\ .\ \lambda \mathrm{z}\ .\ \mathbf{r}_2\ (\mathbf{c}\ (\mathbf{r}_1,\mathrm{z})))
\end{aligned}$$

so eliminating all s's and $\mathbf{y}$. Putting this all in the context of the build, we obtain,

$$\begin{aligned}
&\mathit{lreverse} = \lambda x\ .\ \lambda w\ .\ \texttt{build}^{\text{List}}\ (\lambda(n,c)\ .\\
&\quad \texttt{cata}^{\text{List}}\ (\lambda()\ .\ \lambda z\ .\ z,\ \lambda(r_1,r_2)\ .\ \lambda z\ .\ r_2\ (c\ (r_1,z)))\ x\ w)
\end{aligned}$$

which is equivalent to the result obtained from quadratic reverse!

8 Status

The algorithm we have described handles a wide variety of cases—expressions which are consumers or producers of structured types, or both. Expressions which are producers of T objects are transformed into `build`'s over T. Consumers of T objects are transformed into `cata`'s over T. Functions which are producers of T objects and consumers of S objects are transformed into `build`'s over S surrounding a `cata` over T.

When we have a function which is both a producer and a consumer we first introduce a `build` into the recursion, and then try to turn this recursive function into a `cata`. Of the two methods of Section 6 initially try the second, since when it succeeds it seems to produce superior results. If the fusion algorithm fails using this strategy, we then attempt the first method (pushing the `cata (n,c)` across the `case`), and then attempt to obtain a catamorphism. If neither works we simply give up and leave the function as it was orignally defined.

8.1 More Examples

The `upto` function produces lists from integers.

$$\begin{aligned}
&\mathit{upto} = \lambda \mathit{low}\ .\ \lambda \mathit{high}\ .\\
&\qquad \texttt{case}\ \mathit{low} > \mathit{high}\ \texttt{of}\\
&\qquad\quad \textbf{False}()\ \texttt{->}\ \textbf{Cons}(\mathit{low}, \mathit{upto}\ (\mathit{low}+1)\ \mathit{high})\\
&\qquad \mid\ \ \textbf{True}()\ \texttt{->}\ \textbf{Nil}()
\end{aligned}$$

Since (in our setting) integers are not freely constructed, `upto` is only a producer and not a consumer, so we merely obtain a `build` form:

$$\begin{aligned}
&\mathit{upto} = \lambda \mathit{low}\ .\ \lambda \mathit{high}\ .\ \texttt{build}\ (\mathit{upto}^{\#}\ \mathit{low}\ \mathit{high})\\
&\mathit{upto}^{\#} = \lambda \mathit{low}\ .\ \lambda \mathit{high}\ .\ \lambda(n,c)\ .\\
&\qquad \texttt{case}\ \mathit{low} > \mathit{high}\ \texttt{of}\\
&\qquad\quad \textbf{False}()\ \texttt{->}\ c\ (\mathit{low}, \mathit{upto}^{\#}\ (\mathit{low}+1)\ \mathit{high}\ (n,c))\\
&\qquad \mid\ \ \textbf{True}()\ \texttt{->}\ n\ ()
\end{aligned}$$

A function which both produces and consumes lists is the `zip` function. It is interesting because it recurses over two arguments simultaneously:

$$\begin{aligned}
&\mathit{zip} = \lambda x\ .\ \lambda y\ .\\
&\quad \texttt{case}\ x\ \texttt{of}\\
&\qquad \textbf{Nil}()\ \texttt{->}\ \textbf{Nil}()\\
&\quad \mid\ \textbf{Cons}(a,b)\ \texttt{->}\ \ \texttt{case}\ y\ \texttt{of}\\
&\qquad\qquad\qquad\qquad \textbf{Nil}()\ \texttt{->}\ \textbf{Nil}()\\
&\qquad\qquad\qquad\quad \mid\ \textbf{Cons}(c,d)\ \texttt{->}\ \textbf{Cons}((a,c),\ \mathit{zip}\ b\ d)
\end{aligned}$$

The resulting wrapper/worker pair, wraps a `build` overs *lists* around a `cata` over *lists*.

$$\begin{aligned}
&\mathit{zip} = \lambda x\ .\ \lambda y\ .\ \texttt{build}^{\text{List}}\ (\mathit{zip}^{\#}\ x\ y)\\
&\mathit{zip}^{\#} = \lambda x\ .\ \lambda y\ .\ \lambda(n,c)\ .\\
&\quad \texttt{cata}^{\text{List}}\ (\lambda()\ .\ \lambda u\ .\ n\ (),\\
&\qquad\qquad \lambda(w,g)\ .\ \lambda u\ .\\
&\qquad\qquad\quad \texttt{case}\ u\ \texttt{of}\\
&\qquad\qquad\qquad \textbf{Nil}()\ \texttt{->}\ n()\\
&\qquad\qquad\quad \mid\ \textbf{Cons}(z,zs)\ \texttt{->}\ c\ ((w,z),\ g\ zs))\ x\ y
\end{aligned}$$

Only *zip*'s first argument is traversed using `cata`. The second is taken apart by explicit case-analysis. Thus, as in [GLPJ93, SF93], *zip* only fuses on it's first argument.

The `take` function is interesting since it's Q context is non-trivial, because of testing the integer argument in the original recursive definition. Tying the recursive knot in take is complicated by the fact that this context needs to be duplicated inside the `cata`. For example:

$$\begin{aligned}
&\mathit{take} = \lambda m\ .\ \lambda x\ .\\
&\quad \texttt{case}\ m == 0\ \texttt{of}\\
&\qquad \textbf{True}()\ \texttt{->}\ \textbf{Nil}()\\
&\quad \mid\ \textbf{False}()\ \texttt{->}\ \ \texttt{case}\ x\ \texttt{of}\\
&\qquad\qquad\qquad\qquad \textbf{Nil}()\ \texttt{->}\ \textbf{Nil}()\\
&\qquad\qquad\qquad \mid\ \textbf{Cons}(a,b)\ \texttt{->}\ \textbf{Cons}(a, \mathit{take}\ (m-1)\ b)
\end{aligned}$$

results in the following:

$$\begin{aligned}
&\mathit{take} = \lambda m\ .\ \lambda x\ .\ \texttt{build}^{\text{List}}\ (\lambda(n,c)\ .\\
&\quad \texttt{case}\ m == 0\ \texttt{of}\\
&\qquad \textbf{True}()\ \texttt{->}\ \textbf{Nil}()\\
&\quad \mid\ \textbf{False}()\ \texttt{->}\\
&\qquad\quad \texttt{cata}^{\text{List}}\ (\lambda()\ .\ \lambda m\ .\ n(),\\
&\qquad\qquad\qquad \lambda(z,g)\ .\ \lambda m\ .\\
&\qquad\qquad\qquad\quad c\ (z,\ \ \texttt{case}\ m-1 == 0\ \texttt{of}\\
&\qquad\qquad\qquad\qquad\qquad \textbf{True}()\ \texttt{->}\ n()\\
&\qquad\qquad\qquad\qquad \mid\ \textbf{False}()\ \texttt{->}\ g\ (m-1)))\ x\ m
\end{aligned}$$

The flatten function which squashes trees into lists, illustrates the case where the producer and consumer are of different types.

$$\begin{aligned}
&\mathit{flatten} = \lambda t\ .\ \ \texttt{case}\ t\ \texttt{of}\\
&\qquad\qquad\qquad \textbf{Tip}()\ \texttt{->}\ \textbf{Nil}()\\
&\qquad\qquad\quad \mid\ \textbf{Node}(a,b,c)\ \texttt{->}\\
&\qquad\qquad\qquad\quad \mathit{append}\ (\mathit{flatten}\ a)\ (\textbf{Cons}(b, \mathit{flatten}\ c))
\end{aligned}$$

It is interesting to note that, like the *reverse* function, the additional abstraction introduced by `build` produces a linear flatten function:

$$
\begin{aligned}
&\mathit{flatten} = \lambda t \;.\; \texttt{build}\;(\mathit{flatten}^{\#}\; t)\\
&\mathit{flatten}^{\#} = \lambda t \;.\; \lambda(n,c)\;.\\
&\qquad \texttt{cata}^{\mathrm{Tree}}\;(\lambda()\;.\;\lambda n'\;.\;n'(),\\
&\qquad\qquad \lambda(f,x,g)\;.\;\lambda n'\;.\;\lambda()\;.\;f\;(c\;(x, g\;n')))\;t\;n
\end{aligned}
$$

Our final example is taken from a slightly richer type, intended to indicate how our algorithm is likely to perform when applied to abstract syntax structures. Consider the types

$$
\begin{aligned}
&\mathit{Exp} = \texttt{Rec}\beta\;.\;\mathbf{Num}\;\mathit{Int}\;+\;\mathbf{Id}\;\mathit{String}\;+\;\mathbf{Plus}\;(\beta,\beta)\\
&\mathit{Code} = \mathbf{LoadI}\;\mathit{Int}\;+\;\mathbf{LoadV}\;\mathit{String}\;+\;\mathbf{Add}()
\end{aligned}
$$

The first represents expressions, the second a postfix code form. A typical translation function from the one to the other could be defined as follows.

$$
\begin{aligned}
&\mathit{postfix} = \lambda x\;.\\
&\quad \texttt{case}\;x\;\texttt{of}\\
&\qquad \mathbf{Num}\;n\;\texttt{->}\;\mathbf{Cons}(\mathbf{LoadI}\;n, \mathbf{Nil}())\\
&\quad\mid\;\; \mathbf{Id}\;s\;\texttt{->}\;\mathbf{Cons}(\mathbf{LoadV}\;s, \mathbf{Nil}())\\
&\quad\mid\;\; \mathbf{Plus}(x,y)\;\texttt{->}\;\mathit{append}\;(\mathit{postfix}\;x)\\
&\qquad\qquad (\mathit{append}\;(\mathit{postfix}\;y)\;(\mathbf{Cons}(\mathbf{Add}(), \mathbf{Nil}())))
\end{aligned}
$$

Applying the transformation of the paper gives:

$$
\begin{aligned}
&\mathit{postfix} = \lambda x\;.\;\texttt{build}^{\mathrm{List}}(\mathit{postfix}^{\#}\;x)\\
&\mathit{postfix}^{\#} = \lambda x\;.\;\lambda(n,c)\;.\\
&\quad \texttt{cata}^{\mathrm{Exp}}\;(\lambda m\;.\;\lambda n'\;.\;c\;(\mathbf{LoadI}\;m, n'()),\\
&\qquad\qquad \lambda s\;.\;\lambda n'\;.\;c\;(\mathbf{LoadV}\;s, n'()),\\
&\qquad\qquad \lambda(f,g)\;.\\
&\qquad\qquad\quad \lambda n'\;.\;f\;(\lambda()\;.\;g\;(\lambda()\;.\;c\;(\mathbf{Add}(), n'())))\;x\;n
\end{aligned}
$$

8.2 What next?

We have implemented the algorithm described here, but currently only in the form of a toolbox of operations which are applied to function definitions. A next step would be to combine them in a single operation which takes a Haskell module, say, and rewrites many of the definitions in `build`-`cata` form. Doing this would provide more realistic experience with our heuristics than at present.

One notable shortcoming of our presentation is the informal use of types, even though these are critical to deciding which form of `build` or `cata` to use. There would be significant benefit in exploring our algorithm more formally within an extended 2nd-Order polymorphic lambda calculus.

9 Acknowledgments

The authors wish to acknowledge Leo Fegaras for a careful reading of an earlier draft. The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-93-C-0069.

References

[DB76] J.Darlington and R.Burstall, *A System which Automatically Improves Programs.* Acta Informatica, 6(1), pp 41-60, 1976.

[FW89] A.Ferguson and P.Wadler, *When will deforestation stop?.* Proc. Glasgow workshop on Functional Programming, Rothesay, Scotland, Dept. of CS, Glasgow, 1989.

[GLPJ93] A.Gill, J.Launchbury, and S.Peyton Jones, *A Short-cut to Deforestation.* Proc. ACM FPCA 93, Copenhagen, 1993.

[KL95] D.King and J.Launchbury, *Structuring DFS Algorithms in Haskell.* Proc. ACM POPL 95, San Francisco, 1995.

[Mal89] G.Malcolm. *Homomorphisms and Promotability.* In *Mathematics of Program Construction*, pp 335–347. Springer-Verlag, June 1989.

[MFR91] E.Meijer, M.Fokkinga, and R.Paterson, *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire.* Proc. FPCA 91, LNCS 523, S-V 1991.

[MH95] E.Meijer and G.Hutton, *Bananas in Space: Extending Squiggol to Function-Space Types.* Proc. FPCA 95,

[PJL91] S.Peyton Jones and J.Launchbury, *Unboxed Values as First Class Citizens in a Non-strict Functional Language*, Proc. FPCA 91, LNCS 523, S-V 1991.

[SF93] T.Sheard and L.Fegaras, *A Fold for All Seasons.* Proc. ACM FPCA 93, Copenhagen, 1993.

[SF93] T. Sheard, L. Fegaras and T. Zhou, *Improving Programs Which Induct Over Multiple Inductive Structures.* ACM SIGPLAN workshop on Partial Evaluation and Semantic's Based Program Manipulation, PEPM'94. Orlando Florida. June 1994.

[Tur86] V.Turchin, *The Concept of a Supercompiler.* ACM TOPLAS, **8**, 3, pp 292-325, 1986.

[Wad84] P.Wadler, *Listlessness is better than laziness: lazy evaluation and garbage collection at compile time.* Proc. ACM L&FP, Austin, 1984.

[Wad90] P.Wadler, *Deforestation: Transforming Programs to Eliminate Trees.* TCS 73, pp 231-284, North Holland 1990.