A Design for Warm Fusion

László Németh * Simon Peyton Jones

Department of Computing Science
University of Glasgow

{laszlo,simonpj}@dcs.gla.ac.uk

Abstract

"Lists are often used as "glue" ... ", "Functional programs are often constructed by combining smaller programs, using an intermediate list ... ", "Intermediate lists — and, more generally, intermediate trees — are both the basis and the bane ... " and *"Fusion is the process ... "* are the beginning lines of a lot of published papers over the last three decades. The idea of removing intermediate data structures inspired plenty of research but rarely resulted in working, testable implementations.

After all these papers on *what* to do about intermediate data structures, we give a detailed account of *how*.

1 Introduction

The history of intermediate data structure removal is very old. Since Darlington and Burstall showed how fold-unfold transformations could be used (with human help) to derive a more efficient, single pass function from the composition of two or more functions [1], remarkable progress has been made. Their technique was extended to higher-order functions and later partially automated.

*Research supported in part by the Overseas Research Students Award Scheme No. ORS/96017017

Wadler developed similar ideas in his *listless transformer* [14], in which multi-pass algorithms were converted into single loops in a simple imperative language. Later, he recast his work in a first-order functional language. By defining a *tree-less* form for functions, functions with no intermediate structures, he was able to prove that the composition of such functions can be *deforested* to a single tree-less function. The proof of termination followed later [3]. Attempts to extend the deforestation work to the higher-order case have met with limited success and termination proofs became rather involved.

The major problem of these attempts was the presence of *general recursion*, which combined with higher-order functions made it hard to find where to tie the recursive knot.

By abandoning general recursion in favour of *primitive recursion*, or more generally different but 'regular' forms of recursion, interest has been renewed in the fusion process. Generalising the list-specific work of Bird and Meerten's, Malcolm explained how the promotion theorems from category theory achieve the same effect of removing intermediate structures by fusing *catamorphisms* [9]. But this was only theory.

The first attempt to turn this into practice was by Sheard and Fegaras [13] which was limited in the sense that their language didn't allow for general recursion only catamorphisms. The real breakthrough came when Launchbury and Sheard [8], demonstrated how to cope with general recursion and *automatically* turn functions written in explicit recursive form into catamorphisms.

About the same time, Gill, Launchbury and Peyton Jones introduced a new language construct, the `build`, and automated the application of a one-step fusion rule, but they made no attempt to transform functions written using explicit recursion to the form required by the fusion rule [5]. In order to allow the fusion rule to happen, they reprogrammed most list processing functions from the Haskell prelude and from then on, combinations of these standard functions were deforested. The major result of this work culminated in Gill's thesis [4], which proved that an implementation of deforestation can indeed be put into a real, fully-fledged compiler.

2 Contributions

This paper builds on and is a logical follow-up to these last three. It presents a complete design for a fusion engine in the context of a real compiler and addresses many shortcomings of previous work.

- It is the first implementation of the process of automatically transforming functions written with explicit recursion to catamorphic form.
- It puts all the earlier attempts into a proper, explicitly typed, polymorphic framework of F_2.
- Highlights numerous technical details which needed to be solved for successful fusion.
- Discusses various implementation trade-offs

The design also has several limitations, including

- No higher-order fusion
- No mutually recursive data types
- Fusion is restricted to polynomial and regular types

Extending our techniques to mutually recursive data types is relatively straightforward, the other two limitiations will likely to remain.

3 The language

The syntax of the language – which is the typed intermediate language of GHC – is shown in Fig. 1. It is essentially F_2; bound variables are annotated with their types. Its operational interpretation has been published many times [12, 7].

In the running example of $map^{\mathrm{Tree}\,\alpha}$ (see below) we'll be using a Haskell like notation with the exception that we make type variables explicit and use upper case letters to denote a variable which is related to its lower case counterpart.

Example

```
map :: forall a b.(a->b)->Tree a->Tree b
map  = Λab.λf.λt.
         case t of
           Leaf l     ->Leaf b (f l)
           Branch l r->Branch b
                             (map a b f l)
                             (map a b f r)
```

4 The big picture

The fusion transformation[1] proceeds in two separate phases: first, individual functions are preprocessed in an attempt to express their definition in terms of `build` and `cata`. This process is depicted below. Second, separate invocations of successfully transformed functions are fused with one another using the `cata-build` rule (8). This one step rewrite rule is implemented as an extension to the simplifier in GHC and as such doesn't pose any serious problem.

Our main goal here is to discuss the first phase, which comprises of four steps.

$$\downarrow$$
Derive `cata` and `build` (Sect.5)
$$\downarrow$$
Buildify (Sect.6)
$$\downarrow$$
SAT (Sect.7.1)
$$\downarrow$$
Cataify (Sect.7)
$$\downarrow$$

[1]We will be very informal in this section, omitting type signatures, type variables whenever possible. The sections which detail the different steps give F_2 expressions in their full glory.

First, we derive the `cata` and `build` for every fusible type. This process is performed during type checking and is detailed in Sect. 5.

Buildify denotes the process of introducing `build`'s. It's purpose is twofold. (1) it makes explicit that f is a *good producer*, that is it can be fused with other functions. (2) it splits the function f into a *wrapper* and a *worker* [11] which allows the wrapper to be inlined.

$$\begin{array}{rcl} f & = & \Lambda\bar{\alpha}.\lambda\bar{v}.\mathit{body} \\ & \texttt{=>} & \{\text{- buildify -}\} \\ f & = & \Lambda\bar{\alpha}.\lambda\bar{v}.\texttt{build}\ (f'\ \bar{\alpha}\ \bar{v}) \\ f' & = & \Lambda\bar{\alpha}.\lambda\bar{v}.\Lambda\rho.\lambda\bar{c}.\texttt{cata}\ \mathit{body} \end{array}$$

While this transformation is sound, in some cases the result might be less efficient than the original definition. To avoid this loss of efficiency, we only perform it conditionally, depending on a syntactic check. The process of buildify therefore becomes

1. Perform transformation
2. Simplify
3. If the syntactic check is satisfied, return simplified definition, otherwise discard, and return the original

Cataify denotes the process of transforming the *unary* function f into a catamorphism. It makes explicit that the function is a *good consumer*.

$$\begin{array}{rcl} f & = & \lambda\iota.\mathit{body}\ \iota \\ & \texttt{=>} & \{\text{- cataify -}\} \\ f & = & \lambda\iota.\mathit{body}\ (\texttt{cata}\ \iota) \end{array}$$

Catamorphism are primitive recursive functions. Therefore, not every function can be transformed to catamorphic form. In order

Program	*Prog*	::=	$\mathit{TopDecl}_1$; … ; $\mathit{TopDecl}_n$	$n \geq 1$
Declarations	*TopDecl*	::=	*Binding* | *TypeDecl*	
Declaration	*TypeDecl*	::=	`data` $\mathit{Con}\ \bar{\alpha}$ = $\{C_i\ \bar{\sigma}_i\}_{i=1}^n$	
Types	σ, τ	::=	$\mathit{TyCon}\ [\sigma]$	
		|	$\sigma \to \sigma'$	
		|	$\forall\alpha.\sigma$	
		|	α	
Bindings	*Binding*	::=	*Bind* | `rec` $\mathit{Bind}_1 \ldots \mathit{Bind}_n$	
	Bind	::=	$\mathit{var} :: \sigma$ = *Expr*	
Expression	*Expr*	::=	*Expr Atom*	Application
		|	*Expr* τ	Type application
		|	$\lambda\ \mathit{var}_1 :: \sigma_1 \ldots \mathit{var}_n :: \sigma_n\ .\ \mathit{Expr}$	Lambda abstraction
		|	$\Lambda\ \mathit{ty}\ .\ \mathit{Expr}$	Type abstraction
		|	`case` *Expr* `of` *Alts*	Case expression
		|	`let` *Binding* `in` *Expr*	Local definition
		|	$\mathit{con}\ \mathit{var}_1 \ldots \mathit{var}_n$	Constructor $n \geq 0$
		|	$\mathit{prim}\ \mathit{var}_1 \ldots \mathit{var}_n$	Primitive $n \geq 0$
		|	*Atom*	
Atoms	*Atom*	::=	$\mathit{var} :: \sigma$	Variable
		|	*Literal*	Unboxed Object
Literal values	*Literal*	::=	*integer* | *float* | …	
Alternatives	*Alts*	::=	$\mathit{Calt}_1 ; \ldots ; \mathit{Calt}_n ;\ \mathit{Default}$	$n \geq 0$
		|	$\mathit{Lalt}_1 ; \ldots ; \mathit{Lalt}_n ;\ \mathit{Default}$	$n \geq 0$
Constr. alt	*Calt*	::=	$\mathit{Con}\ \mathit{var}_1 \ldots \mathit{var}_n$ -> *Expr*	$n \geq 0$
Literal alt	*Lalt*	::=	*Literal* -> *Expr*	
Default alt	*Default*	::=	`NoDefault` | *var* -> *Expr*	

Figure 1: Syntax of the Core language

to ensure soundness of this transformation, we need to take the same three step approach (transform, simplify, check) we did in the case of buildify, sacrificing completeness.

The word unary in the preceding paragraph explains why we perform the static argument transformation (SAT) before cataify. Since in our implementation cataify is limited to *first-order* fusion, that is fusion for functions with only one argument, SAT creates a local binding for f where static arguments – those which not change in recursive calls – are not passed around. This increases the opportunities for fusion.

$$\begin{array}{lcl} f & = & \Lambda\bar{\alpha}.\lambda\bar{v}.e \\ & \texttt{=>} & \{\text{- SAT -}\} \\ f & = & \Lambda\bar{\alpha}.\lambda\bar{v}.\texttt{let}\ f' = \lambda x.e'\ \texttt{in}\ f'\ x \end{array}$$

5 Catas and Builds

Our starting point is the type declarations occurring in the text of programs.

$$\texttt{data}\ T\langle\alpha_1\ldots\alpha_m\rangle = \{C_i\langle\sigma_{i1}\ldots\sigma_{ik}\rangle\}_{i=1}^{n} \quad (1)$$

We will use the notation $\langle\alpha_1\ldots\alpha_m\rangle$ (to denote type variables), $\langle\sigma_{i1}\ldots\sigma_{ik}\rangle$ (to denote types) and $\bar{\alpha}$, $\bar{\sigma}_i$ interchangeably. According to this definiton data constructors of $T\bar{\alpha}$ have type $C_i :: \forall\bar{\alpha}.\bar{\sigma}_i \to T\bar{\alpha}$. Type constructors T correspond to functors (in the categorical sense) – in particular they are built up from functors– so they have a natural action on functions as well as on types. We define the action on functions by induction $E^{T\bar{\alpha}}f = \mathcal{E}^{T\bar{\alpha}}f[\![\bar{\sigma}_i]\!]$ where

$$\begin{array}{ll} \mathcal{E}^{T\bar{\alpha}}f[\![\alpha]\!] = \lambda x.x & (2) \\ \mathcal{E}^{T\bar{\alpha}}f[\![T'\bar{\sigma}]\!] = f,\ \text{if}\ T\bar{\alpha} = T'\bar{\sigma} & \\ \mathcal{E}^{T\bar{\alpha}}f[\![T'\bar{\sigma}]\!] = \lambda x.map^{T'\bar{\sigma}}(E^{T\bar{\alpha}}f)\ x, \text{if}\ T\bar{\alpha} \neq T'\bar{\sigma} & \end{array}$$

For every such declaration (1), which declares *regular*[2] and *polynomial*[3] type constructors, we derive the type and definition of two functions, the `build` and the `cata`. `build` expresses the idea how data structures of the given type $T\bar{\alpha}$ are constructed, while `cata` expresses regular consumption. Both construction and consumption have to be polymorphic enough, that is, it must proceed by using only the functions passed to `build` and cata. This is guaranteed by the "free theorem" [15]. We achieve this by introducing a new type variable ρ and replace every occurrence of $T\bar{\alpha}$ in the constructors' type by ρ. We will be using the notation $\mathcal{A}^{T\bar{\alpha}}\ \tau\ \sigma$ to denote the process of systematically replacing occurrences of $T\bar{\alpha}$ by τ in σ.

$$\begin{array}{lcll} \mathcal{A}^{T\bar{\alpha}}\ \rho\ [\![\alpha]\!] & = & \alpha & (3) \\ \mathcal{A}^{T\bar{\alpha}}\ \rho\ [\![T'\bar{\sigma}]\!] & = & \rho,\ \text{if}\ T\bar{\alpha} = T'\bar{\sigma} & \\ \mathcal{A}^{T\bar{\alpha}}\ \rho\ [\![T'\bar{\sigma}]\!] & = & T\ (\mathcal{A}^{T\bar{\alpha}}\ \rho)\ [\![\bar{\sigma}]\!],\ \text{if}\ T\bar{\alpha} \neq T'\bar{\sigma} & \end{array}$$

Note that in the third line, $\mathcal{A}$ is applied to a list of types, which is done by applying it to each element of the list.

For example, for the type of the constructors of the data type of lists $Nil :: \forall\alpha.[\alpha]$, and $Cons :: \forall\alpha.\alpha \to [\alpha] \to [\alpha]$, $\mathcal{A}^{[\alpha]}\ \rho$ gives the type ρ and $\mathcal{A}^{[\alpha]}$ gives $\alpha \to \rho \to \rho$.

To get the type and definition of `build` and `cata` for type $T\bar{\alpha}$, after applying $\mathcal{A}^{T\bar{\alpha}}$ for all

[2] A data type is called regular, iff recursive calls in the body are all of the form of the head of the definition. Otherwise the data type is non-regular.

[3] Sums, products, application of type constructors which are built up from polynomial types and primitive types are all polynomial.

the constructors we quantify over $\bar{\alpha}$ and ρ.

$$\begin{array}{rl} \texttt{build}^{T\bar{\alpha}} & :: \forall\bar{\alpha}.(\forall\rho.\overline{(\mathcal{A}^{T\bar{\alpha}}\ \rho\ \bar{\sigma} \to \rho)} \to \rho) \\ & \to T\ \bar{\alpha} \\ \texttt{build}^{T\bar{\alpha}}\ \bar{\sigma}\ f & = f\ (T\bar{\sigma})\ (\mathcal{A}^{T\bar{\alpha}}\ T\bar{\sigma}\ Constrs(T\bar{\alpha})) \\ \texttt{cata}^{T\bar{\alpha}} & :: \forall\bar{\alpha}\rho.\overline{(\mathcal{A}^{T\bar{\alpha}}\ \rho\ \bar{\sigma} \to \rho)} \to T\ \bar{\alpha} \to \rho \end{array}$$

Example

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

For this example of *map* we get that `build` and `cata` have types

$$\begin{array}{lcl} \texttt{build}^{\mathit{Tree}\ \alpha} & :: & \forall\alpha.(\forall\rho.(\alpha \to \rho) \\ & & \to (\rho \to \rho \to \rho) \to \rho) \\ & \to & \mathit{Tree}\ \alpha \\ \texttt{cata}^{\mathit{Tree}\ \alpha} & :: & \forall\alpha\rho.(\alpha \to \rho) \to (\rho \to \rho \to \rho) \\ & \to & \mathit{Tree}\ \alpha \to \rho \end{array}$$

6 Expressing functions as build 's ('buildify')

We will be following the general procedure we explained in Sect. 4, that is we perform the transformation given by Eq. (4) for every function which has a fusible result type, simplify and check whether the results are what we expect.

6.1 Introducing build

The transformation described here requires a bit of explanation. The `cata` we are introducing in Eq. (4) is rather special. Its sole purpose is to ensure abstraction over the constructors, so at the end of 'buildify' we expect it to disappear via the new rules in Fig. 2. In particular, it usually requires the application of rules (7) and (9) with some inlining to bring the cata close to the `build` so rule (8) applies. To make this check simple we mark the `cata` — keeping in spirit with the fusion analogy, it becomes 'radioactive'.

$$\begin{array}{lcll} f & :: & \forall\bar{\beta}.\bar{\nu} \to T\ \bar{\sigma} & \\ f & = & \Lambda\bar{\beta}.\lambda\bar{v}.e & \\ \texttt{=>} & & & (4) \\ f & :: & \forall\bar{\beta}.\bar{\nu} \to T\ \bar{\sigma} & \\ f & = & \Lambda\bar{\beta}.\lambda\bar{v}.\texttt{build}^{T\bar{\alpha}}\ \bar{\sigma}\ (f'\bar{\beta}\ \bar{v}) & \\ f' & :: & \forall\bar{\beta}.\bar{\nu} \to (\forall\rho.\overline{(\mathcal{A}^{T\bar{\sigma}}\ \rho\ \bar{\sigma} \to \rho)} \to \rho) & \\ f' & = & \Lambda\bar{\beta}.\lambda\bar{v}.\Lambda\rho.\lambda\bar{c}.\texttt{cata}^{T\bar{\alpha}}\ \bar{\tau}\ \rho\ \bar{c}\ e & \end{array}$$

The way `build`'s are introduced also incorporates the idea of workers and wrappers [11]. The wrapper f, including the `build`, is small and can be freely inlined – even into its own worker, which must happen to make the buildify process successful.

Example

```
map :: forall a b.(a->b)->Tree a->Tree b
map  = Λab.λf.λt.build b (map' a b f t)
map':: forall a b.(a->b)->Tree a->
         forall x.(b->x)->(x->x->x)->x
map' = Λab.λf.λt.Λx.λL.λB.
        cata b x L B (
          case t of
           Leaf l    ->Leaf b (f l)
           Branch l r->Branch b
                         (map a b f l)
                         (map a b f r))
```

6.2 Simplification

Simplification is simple: we call a slightly extended version of the simplifier. The new rules are given in Fig. 2.

Example

```
map:: forall a b.(a->b)->Tree a->Tree b
map = Λab.λf.λt.build b (map' a b f t)
map':: forall a b.(a->b)->Tree a->
       forall x.(b->x)->(x->x->x)->x
map'= Λab.λf.λt.Λx.λL.λB.
      case t of
        Leaf l    ->L (f l)
        Branch l r->B (map' a b f l x L B)
                      (map' a b f r x L B)
```

6.3 Possible reasons for failure

Our definition of failure for this transformation is that of the 'radioactive' `cata` remains in the simplified bindings. As explained earlier, leaving this `cata` may result in less efficient code, which we aren't prepared to accept since we cannot know in advance how frequently this will occur. It would be interesting to see how big the performance penalty really is in general programs.

As an example for the `cata` to remain, consider the function *append* :: $\forall\alpha.[\alpha] \to [\alpha] \to [\alpha]$. Even though, it is a perfectly good producer, build introduction fails because the 'radioactive' `cata` remains on *append*'s second argument. This is rather unfortunate since *append* tends to occur frequently in programs.

So what solutions exist? The hackish solution to this problem is to make *append* special and leave the remaining cata. This however incurs a performance penalty if the `cata` doesn't fuse with a build, as we unnecessarily traverse the second list.

A more involved solution is to introduce a function *augment* with type $\forall\alpha.(\forall\beta.\beta \to (\alpha \to \beta \to \beta) \to \beta) \to [\alpha] \to [\alpha]$ which hides the bad properties of *append* [4]. The reason we don't follow this route is that it's not immediately obvious what augment's definition would be for data types other than list.

The Right Solution is to have Fegaras style catas which induct over multiple arguments [2].

6.4 Check if simplification is successful

By explicitly marking the 'radioactive' `cata` we made this check simple. By traversing the simplified binding(s) we check whether the marked `cata` is gone. If this is the case, we discard the previous definition of f and return the new binding(s).

7 Expressing recursive functions as catamorphisms ('cataify')

Functions in F_2 can be written using explicit recursion. While this generality is sometimes useful the `cata-build` rule (8) – which eliminates intermediate data structures – applies to functions in catamorphic form. Our goal is to express as many functions as possible in catamorphic form so they can be fused.

The method we adopt here is based on the *promotion theorem* of Malcolm [9], which describes when the composition of a strict function f with a catamorphism can be expressed as another catamorphism.[4]

$$\frac{\forall C_i^T : f(C_i^T\langle y_{i1}\dots y_{ik}\rangle) = h_{C_i}(E_{C_i}^T f\langle y_{i1}\dots y_{ik}\rangle)}{f(\mathtt{cata}^{T\bar{\alpha}}\ c_1\dots c_n\ x) = \mathtt{cata}^{T\bar{\alpha}}\ h_{C_1}\dots h_{C_n}\ x} \quad (5)$$

Other authors call this the *fusion theorem* [8]. In composing f with a catamorphism, the choice for the later is not arbitrary. Since we do not want to change the meaning of f, we have to use the *identity catamorphism* at type $T\bar{\sigma}$, which is readily expressed by using the abstracted constructors as arguments to the catamorphism operator $\mathtt{cata}^{T\bar{\alpha}}\ \bar{\sigma}\ (T\bar{\sigma})\ (\mathcal{A}^{T\bar{\alpha}}\ (T\bar{\sigma})\ Constrs(T\bar{\alpha}))$, where this rather obscure expression denotes replacing $T\bar{\alpha}$ by $T\bar{\sigma}$ and instantiating α's with the corresponding σ's (substitution).

We are going to use a two step approach, which greatly simplifies earlier work and allows us to prove important properties of the second step while leaving the (yet unproven) properties of the first step unaffected.

First, for *unary* functions we perform the transformation given by (6) and simplify the new bindings.

Example

```
hLeaf   :: b -> x
hLeaf    = λz11.map'' (Leaf y11)
hBranch :: x -> x -> x
hBranch  = λz21.λz22.
           map'' (Branch y21 y22)
```

In effect, we separate the action of f into n cases, where n is the number of constructors of f's arguments type, and denote these cases h_{C_i} reflecting which constructor they belong to. In other words, we express the function f out of a sum as the product of functions out of each summand.

$$\begin{array}{rcl}
f & :: & T\bar{\sigma} \to \tau \\
f & = & \lambda\iota.e \\
=> & & \\
f & :: & T\bar{\sigma} \to \tau \\
f & = & \lambda\iota.\mathtt{cata}^{T\bar{\alpha}}\ \bar{\sigma}\ \rho\ h_{C_1}\dots h_{C_n}\ \iota \\
\text{where} & & \\
\forall i & \in & 1,\dots,n, \forall j \in 1,\dots,k \\
h_{C_i} & :: & \sigma'_{i1} \to \dots \to \sigma'_{ik} \to \tau \\
h_{C_i} & = & \lambda z_{i1}\dots z_{ik}.f\ (C_i^T\ \overline{\langle y_i :: \sigma_i\rangle}) \\
\sigma'_{ij} & = & \mathcal{A}_i^{T\bar{\sigma}}\ \rho\ \sigma_{ij}
\end{array} \quad (6)$$

Notice, that these newly introduced local bindings have *free* variables $y_{i1}\dots y_{ik}$ and *unused* variables $z_{i1}\dots z_{ik}$. The relation between $\bar{y}$ and $\bar{z}$ is that the former represents data *before* the recursive call to f, while the latter represents data *after* the recursive call. This is manifested in their types. Whenever y_{ij}, for some j, has type $T\bar{\sigma}$, that is the same as the arguments type to f, the corresponding z_{ij} has the abstracted type ρ. Otherwise, they have the same type.

---

[4] A succinct form of this theorem is given, called the *fusion law* in Meijer [10] as:

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \Leftarrow f \circ \bot = (\!|\psi|\!) \circ \bot \wedge f \circ \varphi = \psi_L$$

A slight variation of the fusion law is to replace the condition $f \circ \bot = (\!|\psi|\!) \circ \bot$ by $f \circ \bot = \bot$, i.e. f is strict.

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \Leftarrow f \circ \bot = \bot \wedge f \circ \varphi = \psi_L$$

Second, we eliminate explicit recursion by replacing combinations of the old variables with recursive calls of f, in favour of newly introduced variables z_{ij}. This rewriting process is explained in Sect. 7.2.

Example

```
hLeaf   :: b -> x
hLeaf    = λz11.L (f y11)
hBranch :: x -> x -> x
hBranch  = λz21.λz22.B (map'' y21)
                       (map'' y22)
```

7.1 Static parameters

The astute reader will notice the emphasis on *unary* in Sect. 7 and raise the question whether this is too restrictive. The answer is yes and no. While it is true, that most functions in usual programs have more than one argument, in most cases the additional arguments are *static*, i.e. they don't change in recursive calls. A simple transformation, the static argument transformation [12], will derive a new local function, where the static arguments are not passed around. Frequently, we end up with a recursive local function with one argument where our techniques become applicable. We use the static argument transformation to increase the opportunity for turning functions into catamorphisms.

Another way around this problem, is to extend the algorithm to deal with functions with more than one argument. This requires generalising the fusion theorem and the rewriting process. Troubles don't end here, since the *second-order fusion theorem* and the corresponding rewrite rules devised by Launchbury and Sheard [8] can transform functions with more than one argument to catamorphic form, fusion will happen *only on one argument.* So, for example *zip* will not get fused on both of its arguments. If we want *zip* and similar functions to be fusible on all of their arguments we have to look at the work of Fegaras et al [2]. To implement this sort of fusion engine we would have to give up one of our fundamental assumptions. As we explained earlier, currently we derive the *type and definition* of the catamorphism and `build` from the type declaration once for all, so all functions with the given type share these. A Fegaras style fusion engine would require deriving the `cata` and `build` from the definition of every function, making the process more complicated. We feel, that the additional machinery required to implement higher-order fusion or fusion for multiple inductive arguments is not worth until it is proven that fusion is a valuable compiler optimisation.

Example

```
map:: forall a b.(a->b)->Tree a->Tree b
map = Λab.λf.λt.build b (map' a b f t)
map':: forall a b.(a->b)->Tree a->
       forall x.(b->x)->(x->x->x)->x
map'= Λab.λf.λt.Λx.λL.λB.
      let
        map'' :: Tree a -> x
        map''  = λt.
                 case t of
                   Leaf l    ->L (f l)
                   Branch l r->B (map'' l)
                                 (map'' r)
      in map'' t
```

7.2 The dynamic rewrite system

As the final step, we are going to use a simple rewrite system to eliminate combinations of the pre-recursion variables ($\bar{y}$) with f, in favour of

$$
\begin{array}{rcll}
\mathtt{cata}^{T\bar{\alpha}}\,\bar{\tau}\,\rho\,\bar{c}\,(C_i\langle y_{i1}\ldots y_{ik}\rangle) & \to & c_i(E^{T}_{C_i}(\mathtt{cata}^{T\bar{\alpha}}\,\bar{\tau}\,\rho\,\bar{c})\langle y_{i1}\ldots y_{ik}\rangle) & (7)\\
\mathtt{cata}^{T\bar{\alpha}}\,\bar{\tau}\,\rho\,\bar{c}\,(\mathtt{build}^{T\bar{\alpha}}\,\bar{\tau}\,f) & \to & f\,\rho\,\bar{c} & (8)\\
\mathtt{cata}^{T\bar{\alpha}}\,\bar{\tau}\,\rho\,\bar{c}\,(\mathtt{case}\ x :: T\bar{\tau}\ \mathtt{of}\ \{\overline{C\,\bar{y} \to e}\}) & \to & \mathtt{case}\ x :: T\bar{\tau}\ \mathtt{of}\ \{\overline{C\,\bar{y} \to \mathtt{cata}^{T\bar{\alpha}}\,\bar{\tau}\,\rho\,\bar{c}\,e}\} & (9)\\
\mathtt{cata}^{T\bar{\alpha}}\,\bar{\tau}\,\rho\,\bar{c}\,\mathit{error} & \to & \mathit{error} & (10)
\end{array}
$$

Figure 2: New `cata` related rules

the post-recursion variables ($\bar{z}$). One interesting property of this rewriting process is that the rewrite rules are not fixed: we have to generate a set unique rewrite rules for each constructor. First we introduce some notation.

Notation 1
- *$lhs \to rhs$ is a rewrite rule, which allows us to replace (in one step) lhs with rhs*
- *$lhss \Rightarrow rhss$ is the same as $\to$ except that $lhss$ and $rhss$ are* n-tuples. *We define $\{(x_1,\ldots,x_n) \Rightarrow (y_1,\ldots,y_n)\}$ to mean the set of rules extracted component-wise $\{x_1 \to y_1,\ldots,x_n \to y_n\}$*
- *we also extend the $\Rightarrow$ notation by allowing E functors over tuples. Thus: $\{E_C f(x_1,\ldots,x_n) \Rightarrow (y_1,\ldots,y_n)\}$ is equivalent to the set of rules $\{E_C f x_1 \to y_1,\ldots,E_C f x_n \to y_n\}$*

Definition 1 *For each constructor C_i of type $T\bar{\alpha}$ we define the dynamic rewrite rules to be the set:*

$$\{E_{C_i}\,f\,(y_{i1},\ldots,y_{ik}) \Rightarrow (z_1,\ldots,z_k)\} \quad (11)$$

Looking at the rewrite system one might naturally ask the question, why don't these rewrite rules have type variables? The simple answer is that static argument transformation has been applied to f and it found all the type arguments to be static. The only case when a type variable is not static in the recursive call to f itself is the case of *polymorphic recursion.* Currently, we can see no easy way to incorporate these into our fusion engine.

Theorem 1 *The rewrite system generated by (11) is confluent and terminating.*

The proofs are trivial.

Example

```
y11 -> z11,
 map'' y21 -> z21, map'' y22 -> z22
hLeaf   :: b -> x
hLeaf    = λz11.L (f z11)
hBranch :: x -> x -> x
hBranch  = λz21.λz22.B z21 z22
```

7.3 Reasons for failure

There are various reasons for this transformation to fail:

- the inductive argument is consumed by another function, therefore the dynamic rewrite systems fails to replace all pre-recursion variables with post-recursion ones

$$
\begin{array}{lcl}
f & :: & [\alpha] \to \mathrm{Int} \\
f & = & \lambda\iota.\mathtt{case}\ \iota\ \mathtt{of} \\
 & & \quad \mathit{Cons}\ x\ xs \to f\ xs + \mathit{length}\ xs
\end{array}
$$

The dynamic rewrite system will generate (amongst others) the rule $f\ xs \to zs$, where zs is a new variable, which will not replace $\mathit{length}\ xs$, so the pre-recursion variable xs remains.

- the function is not primitive recursive

In both cases, failure will show up as pre-recursion y_{ij} variables remaining in the simplified and rewritten bindings.

7.4 Check if simplification is successful

Simplification is successful if none of the pre-recursion variables remain after the rewriting. We check for this by traversing the simplified bindings. If any of y_{ij} remain, we discard everything we've done in this section and continue using the original definition of f.

Example

```
map :: forall a b.(a->b)->Tree a->Tree b
map  = Λab.λf.λt.build b (map' a b f t)
map':: forall a b.(a->b)->Tree a->
          forall x.(b->x)->(x->x->x)->x
map' = Λab.λf.λt.Λx.λL.λB.
        cata b x (λz11.L (f z11))
                 (λz21.λz22.B z21 z22)
                 t
```

7.5 Workers and Wrappers again

The transformations we have described so far do present the full story of warm fusion but aren't enough to actually reap the benefits. The success depends on bringing builds and catas close so the `cata-buildrule` applies. This bringing them close means inlining. Unfortunately, inlining is a rather delicate aspect of compilers since it can lead to code explosion, which in our case means, that the newly generated cata with the local h_{C_i} functions will certainly be too big to be inlined, so no fusion will happen.

To make the `cata` inlineable we perform lambda lifting [6], which lifts the local h_{C_i}'s leaving the function containing `cata` small.

8 The real work...

In preceding sections we have presented the full design for a fusion engine. It takes functions written with explicit recursion and – whenever possible – transforms them into catamorphisms by abstracting over the constructors. This abstraction however has its price. During the first runs of the modified compiler we noticed that in the resulting programs the total memory allocation increased considerably, sometimes tripled. It is a rather unexpected behaviour from an optimisation which claims to decrease allocation by eliminating intermediate data structures! A closer look at the resulting code revealed what was happening. The transformation splits a single function into several smaller ones, one including the build, another the `cata` plus one function for each constructor the given data type has. This increases the number of closures and the additional variables increases the size of these closures. Unfortunately, for data types with more constructors, this gets even worse.

9 Conclusion and future work

We presented the design of a fusion engine for the non-strict functional language Haskell, which simplifies earlier attempts and puts them into practice. The design allows a neat separation of the two rewritings in the process of transforming strict functions to catamorphisms. The implementation based on this design is completed, what we have left is to ensure that the code related to fusion works smoothly with the rest of GHC and we do get the benefits of fusion. This part turns out to be a lot harder and more tiresome than we initially expected.

This design, while it has its own limitations – most notably the lack of higher-order, or multiple argument fusion – vastly extends the applicability of previous work, from the data type of lists over a fixed set of functions to polynomial, regular data types over primitive recursive functions, written with explicit recursion.

It gives us deeper insight into what is needed for fusion to work and raises several questions:

- The key to successful fusion is *precise control* over inlining: we have to be able to:
  - control whether inlining can, cannot or must happen on a per function, per simplifier pass basis,
  - depending on the result of other transformations, change inlining properties.

  How this control is best achieved?
- When exactly should this transformation happen to be most beneficial for a large scale of programs? Currently in GHC, the set of available optimisations together with the calls to the simplifier are hard-wired into a Perl script, and the order of transformations is fixed [12]. We have already seen that fusion requires us to abandon this model of compilation as different actions must be taken depending on the success or failure of each pass.
- Transforming functions to catamorphisms creates new functions with more arguments than the original. In the case of a set of mutually recursive data types the number of arguments to each successfully transformed function is one for each constructor for each data type. For example, within the compiler, the Core language (F_2) is represented as a set of mutually recursive types with 18 constructors altogether. Every single function which acts on any of these types would take 18 more arguments, leading to larger closure sizes and lot more allocation. It is highly unlikely that current compilers would be able to produce efficient code without special care.

Our next goal is to polish the implementation, and measure benefits of this optimisation on a large set of *real* programs, perhaps including the compiler itself! Evaluating these measurements will enable us to finally address the long open question whether fusion is worthy to be included to a optimising compiler or not.

References

[1] R. M. Burstall and J. Darlington. A System which Automatically Improves Programs. *Acta Informatica*, 6:41–60, 1976.

[2] L. Fegaras, T. Sheard, and T. Zhou. Improving programs which recurse over

multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–32, Orlando, Florida, 25 June 1994.

[3] A. B. Ferguson and P. Wadler. When will deforestation stop? In *Functional Programming, Glasgow 1988. Workshops in Computing*, pages 39–56, Aug. 1988.

[4] A. J. Gill. *Cheap Deforestation for Non-Strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1996.

[5] A. J. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *FPCA*, 1993.

[6] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture, Nancy*, LNCS 201, Sept. 1985.

[7] S. L. P. Jones. Compiling Haskell by program transformation: A report from the trenches. In H. R. Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44, Linköping, Sweden, 22–24 Apr. 1996. Springer.

[8] J. Launchbury and T. Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *Functional Programming & Computer Architecture*, pages 314–323, San Diego, US, 1995.

[9] G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[10] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[11] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, LNCS 523*, pages 636–666. Springer Verlag, June 1991.

[12] A. Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1995.

[13] T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming and Computer Architecture*, pages 233–242. Association for Computing Machinery, 1993.

[14] P. Wadler. Listlessness is Better than Laziness. In *ACM Symposium on Lisp and Functional Programming, Austin, Texas*, Aug. 1984.

[15] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. Association for Computing Machinery, 1989.