A Design for Warm Fusion

László Németh * Simon Peyton Jones
Department of Computing Science
University of Glasgow
{lasszlo,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], re-
markable progress has been made. Their technique was extended to higher-order functions and later partially automated.

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 treeless form for functions, functions with no intermediate structures, he was able to prove that the composition of such functions can be deformed to a single treeless 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 Meertens, 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 Seshadri and Pégase [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 Seshadri [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 the on standard 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 limitations 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 mapTree (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.
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 buildify. Its purpose is twofold. (1) It makes explicit that if 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.

\[
\begin{align*}
  f &= \Lambda \lambda \body. body \\
  &= \Lambda \lambda \body \cdot \text{buildify} \\
  f' &= \Lambda \lambda \body \cdot \text{buildify} (f' \odot 0)
\end{align*}
\]

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.

\[
\begin{align*}
  f &= \lambda \body \cdot \text{cataify} \\
  &= \Lambda \lambda \body \cdot \text{cataify} (f)
\end{align*}
\]

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

---

4 The big picture

The fusion transformation proceeds in two separate phases: first, individual functions are preprocessed in an attempt to express their definition in terms of buildify and cataify. This process is depicted below. Second, separate invocations of successfully transformed functions are fused with one another using the cataify-buildify 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.

Derive cata and buildify (Sect. 5)

Buildify (Sect. 6)

SAT (Sect. 7.1)

Cataify (Sect. 7)

---

![Figure 1: Syntax of the Core language](image-url)
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.

\[
f = \lambda d. \lambda e. x \Rightarrow \lambda x.e \quad f = \lambda d. \lambda e. \text{let } f' = \lambda x.e' \text{ in } f' e
\]

5 Catas and Builds

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

\[
\text{data } T(\alpha_1 \cdots \alpha_m) = (C_t(\alpha_1 \cdots \alpha_k) \alpha_{k+1} \cdots \alpha_m)_{r=1}^n. \quad (1)
\]

We will use the notation \( \alpha_1 \cdots \alpha_m \) (to denote type variables), \( \sigma_1 \cdots \sigma_k \) (to denote types) and \( \alpha, \beta \) interchangeably. According to this definition data constructors of \( T \) have type
\( C_t : \forall \alpha_1 \ldots \alpha_k \rightarrow T \alpha \). Type constructors \( T \) correspond to functions (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 \( E^{T \alpha} f = E^{T \alpha} f [\beta] \) where

\[
E^{T \alpha} f [\alpha] = \lambda x.e \quad E^{T \alpha} f [T] = f \quad E^{T \alpha} f [T] \beta = \lambda x. \text{map}(E^{T \alpha} f ) x, \text{if } T \alpha \neq T \beta \quad (2)
\]

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 \( T \alpha \) 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 \( \rho \) and replacing every occurrence of \( T \alpha \) in the constructors' type by \( \rho \). We will be using the notation \( A^{T \alpha} x = \beta \) to denote the process of systematically replacing occurrences of \( T \alpha \) by \( \rho \) in \( A \).

\[
A^{T \alpha} \rho [\alpha] = \alpha \quad A^{T \alpha} \rho [T] = \rho, \text{if } T \alpha = T \rho \quad A^{T \alpha} \rho [T'] = T (A^{T \alpha} \rho) [\beta], \text{if } T \alpha \neq T \rho \quad (3)
\]

Note that in the third line, \( A \) 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 \( \text{Nil} : \forall \alpha, [\alpha] \rightarrow [\alpha], \text{Cons} : \forall \alpha \rightarrow [\alpha] \\rightarrow [\alpha], A^{[\alpha]} \rho \) gives the type \( \rho \) and \( A^{[\alpha]} \beta \) given \( \alpha \rightarrow \rho \rightarrow \rho \).

To get the type and definition of build and cata for type \( T \alpha \), after applying \( A^{T \alpha} \) for all the constructors we quantify over \( \alpha \) and \( \rho \).

\[
\text{build}^{T \alpha} : \forall \alpha.(A^{T \alpha} \rho \alpha \rightarrow \rho) \rightarrow T \alpha \quad \text{cata}^{T \alpha} : \forall \alpha.(A^{T \alpha} \rho \alpha \rightarrow \rho) \rightarrow T \alpha \rightarrow \rho
\]

\[
\begin{align*}
\text{Example} \\
\text{data } \text{Tree } a = \text{Leaf } a \\
\quad \text{Branch } (\text{Tree } a) (\text{Tree } a)
\end{align*}
\]

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

\[
\begin{align*}
\text{build}^{\text{Tree } a} & : \forall \alpha.(\forall \alpha.(\alpha \rightarrow \rho) \\
& \quad \rightarrow (\rho \rightarrow \rho \rightarrow \rho) \\
& \quad \rightarrow \text{Tree } \alpha \\
\text{cata}^{\text{Tree } a} & : \forall \alpha.(\alpha \rightarrow \rho) \rightarrow (\rho \rightarrow \rho) \\
& \quad \rightarrow \text{Tree } \alpha \\
\end{align*}
\]

6 Expressing functions as "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 buildify

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’.

\[
\begin{align*}
\text{Example} \\
\text{data } \text{Tree } a = \text{Leaf } a \\
\quad \text{Branch } (\text{Tree } a) (\text{Tree } a)
\end{align*}
\]

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

\[
\begin{align*}
\text{build}^{\text{Tree } a} & : \forall \alpha.(\forall \alpha.(\alpha \rightarrow \rho) \\
& \quad \rightarrow (\rho \rightarrow \rho \rightarrow \rho) \\
& \quad \rightarrow \text{Tree } \alpha \\
\text{cata}^{\text{Tree } a} & : \forall \alpha.(\alpha \rightarrow \rho) \rightarrow (\rho \rightarrow \rho) \\
& \quad \rightarrow \text{Tree } \alpha \\
\end{align*}
\]

6.2 Simplification

Simplification is simple: we call a slightly extended version of the simplifier. The new rules are given in Fig. 2.
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 :: [a] → [a] → [a]. 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 \( \text{Ve} \cdot (\beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow (\alpha \rightarrow [\alpha] \rightarrow [\alpha]) \) which hides the bad properties of append [6]. 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 Fegusar style catsa 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 bindings 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.

7 Expressing recursive functions as catamorphisms ('catafy')

Functions in \( F_\lambda \) 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 catomorphic 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.\(^4\)

\[
\forall C^f_r : I(C^r(y_1 \ldots y_k)) = h_C(f(x_1 \ldots x_k)) \\
\frac{f(cata^C_{\alpha_1 \ldots \alpha_n} x) = cata^C_{\alpha_1 \ldots \alpha_n} x}{(5)}
\]

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 \( T \alpha \), which is readily expressed by using the abstraction constructs as arguments to the catamorphism operator \( cata^{T \alpha} \circ (T \alpha) \) (\( A^{T \alpha} (T \alpha) \) Constr\( (T \alpha) \)), where this rather obscure expression denotes replacing \( T \alpha \) by \( T \alpha \) and instantiating \( \alpha \)'s with the corresponding \( \alpha \)'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.

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

\[
f \circ \{\phi\} = \{\psi\} \iff fe \downarrow \Leftrightarrow \psi \circ f = \psi \\
f \circ \{\phi\} = \{\psi\} \iff fe \downarrow \Leftrightarrow f \circ \{\phi\} = f \circ \psi
\]

A slight variation of the fusion law is to replace the condition \( fe \downarrow \Leftrightarrow \psi \circ f = \psi \) with \( fe \downarrow \Leftrightarrow f \circ \{\phi\} = f \circ \psi \) is strict.

Notice, that these newly introduced local bindings have free variables \( y_1 \ldots y_k \) and unused variables \( z_{a1} \ldots z_{ak} \). The relation between \( y \) and \( \varepsilon \) 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 \( y \), for some \( j \), has type \( T \beta_j \), that is the same as the arguments type to \( f \), the corresponding \( z_j \) has the abstracted type \( \rho \). Otherwise, they have the same type.
Second, we eliminate explicit recursion by replacing combinations of the old variables with recursive calls of \( f \), in favour of newly introduced variables \( z_i \). This rewriting process is explained in Sect. 7.2.

\[ \text{Example} \]
\[ \text{bLeaf} :: b \rightarrow x \]
\[ \text{bLeaf} = \lambda x.11.1 \ (f \ y11) \]
\[ \text{hBranch} :: x \rightarrow x \rightarrow x \]
\[ \text{hBranch} = \lambda x11.1 \lambda x22.2 \ (\text{map}' \ x21) \ (\text{map}' \ y22) \]

### 7.1 Static parameters

The astute reader will notice the emphasis on unicity 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 similar 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. Trivialities 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

\[ \text{cata}^\alpha \ p \ v \ C_1 (x_1, \ldots, x_k) \rightarrow c(f \ C_1 (y_1, \ldots, y_k)) \]
\[ \text{cata}^\alpha \ p \ v \ (\text{build}^\alpha \ p \ f) \rightarrow f \ p \ v \]
\[ \text{cata}^\alpha \ p \ v \ \text{case } x :: \ T \rightarrow (C \ y \rightarrow C) \rightarrow \text{case } x :: T \rightarrow \text{of } (C \ y \rightarrow \text{cata}^\alpha \ p \ v \ c) \]
\[ \text{cata}^\alpha \ p \ v \ \text{error} \rightarrow \text{error} \]

Figure 2: New cata related rules

the post-recursion variables \( \beta \). 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**

- \( \text{lhs} \rightarrow \text{rhs} \) is a rewrite rule, which allows us to replace (in one step) \( \text{lhs} \) with \( \text{rhs} \)

\[ \text{map} : \forall a. \ b \rightarrow (\text{Tree} a \rightarrow \text{Tree b}) \]
\[ \text{map} = \lambda a. \ \lambda \text{Tree b} (\text{map}' \ a \ b \ f \ t) \]
\[ \text{map}' : \forall a. \ (\text{Tree a} \rightarrow \text{Tree b}) \rightarrow \text{Tree b} \]
\[ \text{let} \]
\[ \text{map}' = \lambda a. \ \lambda \text{Tree b} (\text{map}' \ a \ b \ f \ t) \]
\[ \text{let} \]
\[ \text{map}' = \lambda a. \ \lambda \text{Tree b} (\text{map}' \ a \ b \ f \ t) \]
\[ \text{let} \]
\[ \text{map}' = \lambda a. \ \lambda \text{Tree b} (\text{map}' \ a \ b \ f \ t) \]
\[ \text{let} \]
\[ \text{map}' = \lambda a. \ \lambda \text{Tree b} (\text{map}' \ a \ b \ f \ t) \]

### 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 \( \beta \) with \( f \), in favour of

\[ \text{cata}^\alpha \ p \ v \ C_1 (x_1, \ldots, x_k) \rightarrow c(f \ C_1 (y_1, \ldots, y_k)) \]
\[ \text{cata}^\alpha \ p \ v \ (\text{build}^\alpha \ p \ f) \rightarrow f \ p \ v \]
\[ \text{cata}^\alpha \ p \ v \ \text{case } x :: \ T \rightarrow (C \ y \rightarrow C) \rightarrow \text{case } x :: T \rightarrow \text{of } (C \ y \rightarrow \text{cata}^\alpha \ p \ v \ c) \]
\[ \text{cata}^\alpha \ p \ v \ \text{error} \rightarrow \text{error} \]

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

The proofs are trivial.

**Example**

\[ y11 \rightarrow z11, \]
\[ \text{map}' \ y21 \rightarrow z21, \text{map}' \ y22 \rightarrow z22 \]
\[ \text{hBranch} :: x \rightarrow x \rightarrow x \]
\[ \text{hBranch} = \lambda x11.1 \lambda x22.2 \ (\text{map}' \ x21) \ (\text{map}' \ y22) \]

**Definition 1** For each constructor \( C_i \) of type \( T \) we define the dynamic rewrite rules to be the set:

\[ \{ E_{C_i} f (y_1, \ldots, y_k) \Rightarrow (x_1, \ldots, x_k) \} \]

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 is 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.

### 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  

\[ f :: [a] \rightarrow \text{Int} \]

\[ f = \lambda \text{case of} \]  

\[ \text{Cons } x \ z \rightarrow f \ x + \text{length } z \]

The dynamic rewrite system will generate (amongst others) the rule \( f \ x \rightarrow z \), where \( x \) is a new variable, which will not replace \text{length } z, \text{so the pre-recursion variable } z \text{ remains.}

- the function is not primitive recursive

In both cases, failure will show up as pre-recursion \( y_g \) 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 \( y_g \) remain, we discard everything we've done in this section and continue using the original definition of \( f \).

Example

\[ \text{map} :: \forall a b. (a \rightarrow b) \rightarrow \text{Tree } a \rightarrow \text{Tree } b \]

\[ \text{map} = \text{Aux Auction build } b (\text{map' } \ a \ b \ f \ t) \]

\[ \text{map'} :: \forall a b. (a \rightarrow b) \rightarrow \text{Tree } a \rightarrow \text{Tree } b \]

\[ \text{map'} = \text{Aux Auction Auction Auction} \]

\[ \text{Aux } b \ x \ (\text{Aux11 } b (f \ x) (\text{Aux22 } b (\text{Aux11 } x (\text{Aux22 } x)) (\text{Aux22 } x)) \]

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 cata close so the \text{cata}_\text{build rule} 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 \( h_c \), functions will certainly be too big to be inline, so no fusion will happen.

To make the \text{cata} inlinesable we perform \text{lambda} lifting [6], which lifts the local \( h_c \)'s leaving the function containing \text{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 tripling. 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 complete, 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 \( F_3 \) 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 \text{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 in a optimizing compiler or not.

References


