Bringing Functions into the Fold

Jasper Van der Jeugt Steven Keuchel Tom Schrijvers

Department of Applied Mathematics, Computer Science and Statistics
Ghent University, Krijgslaan 281, 9000 Gent, Belgium
{jasper.vanderjeugt,steven.keuchel,tom.schrijvers}@ugent.be

Abstract

Programs written in terms of higher-order recursion schemes like *foldr* and *build* can benefit from program optimizations like shortcut fusion. Unfortunately, programmers often avoid these schemes in favor of explicitly recursive functions.

This paper shows how programmers can continue to write programs in their preferred explicitly recursive style and still benefit from fusion. It presents syntactic algorithms to automatically find and expose instances of *foldr* and *build*. The algorithms have been implemented as GHC compiler passes and deal with all regular algebraic datatypes, not just lists. A case study demonstrates that these passes are effective at finding many instances in popular Haskell packages.

Keywords catamorphisms, fold/build fusion, analysis, transformation

1. Introduction

Higher-order functions are immensely popular in Haskell, whose Prelude alone offers a wide range of them (e.g., *map*, *filter*, *any*, ...). This is not surprising, because they are the key *abstraction* mechanism of functional programming languages. They enable capturing and reusing common frequent patterns in function definitions.

Recursion schemes are a particularly important class of patterns captured by higher-order functions. They capture forms of recursion found in explicitly recursive functions and encapsulate them in reusable abstractions. In Haskell, the *foldr* function is probably the best known example of a recursion scheme. It captures structural recursion over a list.

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ f\ z\ [\,] \quad\ = z \\ &foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs) \end{aligned}$$

By means of a recursion scheme like *foldr*, the programmer can avoid the use of explicit recursion in his functions, like

$$\begin{aligned} &sum :: [Int] \to Int \\ &sum\ [\,] \quad\ = 0 \\ &sum\ (x : xs) = x + sum\ xs \end{aligned}$$

in favor of implicit recursion in terms of the recursion scheme:

$$\begin{aligned} &sum :: [Int] \to Int \\ &sum\ xs = foldr\ (+)\ 0\ xs \end{aligned}$$

Gibbons [6] has likened the transition from the former style to the latter with the transition from imperative programming with `gotos` to structured programming. Indeed the use of recursion schemes has many of the same benefits for program construction and program understanding.

One benefit that has received a lot of attention in the Haskell community is program optimization. Various equational laws have been devised to *fuse* the composition of two recursion schemes into a single recursive function. Perhaps the best-known of these is *shortcut fusion*, also known as foldr/build fusion [7].

$$\forall c\ n\ g.foldr\ c\ n\ (build\ g) = g\ c\ n$$

This law fuses a list producer, expressed in terms of *build*, with a list consumer, expressed in terms of *foldr* in order to avoid the construction of the allocation of the intermediate list; the latter is called *deforestation* [22].

A significant weakness of foldr/build fusion and other fusion approaches is that programmers have to explicitly write their programs in terms of the appropriate recursion schemes. This is an easy requirement when programmers only reuse library functions written in the appropriate style. However, when it comes to writing their own functions, programmers usually resort to explicit recursion. This not just true of novices, but applies just as well to experienced Haskell programmers, including, as we will show, the authors of some of the most popular Haskell packages.

Hence, already in their '93 work on fold/build fusion, Gill et al. [7] have put forward an important challenge for the compiler: to automatically detect recursion schemes in explicitly recursive functions. This allows programmers to have their cake and eat it too: they can write functions in their preferred style and still benefit from fusion.

As far as we know, this work is the first to pick up that challenge and automatically detect recursion schemes in a Haskell compiler. Our contributions are in particular:

1. We show how to automatically identify and transform explicitly recursive functions that can be expressed as folds.
2. In order to support a particular well-known optimization, fold-build fusion, we also show how to automatically detect and transform functions that can be expressed as a call to *build*.
3. We provide a GHC compiler plugin[1] that performs these detections and transformations on GHC Core. Our plugin not only covers folds and builds over lists, but over all inductively defined directly-recursive datatypes.
4. A case study shows that our plugin is effective on a range of well-known Haskell packages. It identifies a substantial num-

[1] `http://github.ugent.be/javdrjeu/what-morphism`

ber of explicitly recursive functions that fit either the fold or build pattern, reveals that our approach works well with GHC's existing list fusion infrastructure.

2. Overview

This section briefly reviews folds, builds and fusion to prepare for the next sections, that provide systematic algorithms for finding folds and builds in explicitly recursive functions.

2.1 Folds

Catamorphisms are functions that *consume* an inductively defined datastructure by means of structural recursion. Here are two examples of catamorphisms over the most ubiquitous inductive datatype, lists.

```
upper :: String → String
upper []       = []
upper (x : xs) = toUpper x : upper xs

product :: [Int] → Int
product []       = 1
product (x : xs) = x * product xs
```

Instead of using explicit recursion again and again, the catamorphic pattern can be captured once and for all in a higher-order function, the fold function. In case of list, that fold function is $foldr$.

```
upper   = foldr (λx xs → toUpper x : xs) []
product = foldr (*) 1
```

Folds with Parameters Some catamorphisms take extra parameters to compute their result. Hinze et al. [9] distinguish two kinds of extra parameters: *constant* and *accumulating* parameters. List concatenation is an example of the former:

```
cat :: [a] → [a] → [a]
cat []       ys = ys
cat (x : xs) ys = x : cat xs ys
```

The ys parameter is a constant parameter because it does not change in the recursive calls. We can get rid of this constant parameter by hoisting it out of the loop.

```
cat :: [a] → [a] → [a]
cat l ys = loop l
  where
    loop :: [a] → [a]
    loop []       = ys
    loop (x : xs) = x : loop xs
```

Now, the local function can be rewritten in terms of a $foldr$ without having to bother with passing the constant parameter.

```
cat xs ys = loop l
  where
    loop l = foldr (:) ys l
```

Finally, we can inline the local function entirely.

```
cat xs ys = foldr (:) ys l
```

The intermediate steps can of course be skipped in practice and our algorithm in Section 3 does so.

Accumulating parameters are trickier to deal with as they may vary in recursive calls. An example is the accumulator-based sum function:

```
sumAcc :: [Int] → Int → Int
sumAcc []       acc = acc
sumAcc (x : xs) acc = sumAcc xs (x + acc)
```

where the acc parameter varies in the recursive call. Typically, such a function is defined in terms of the $foldl$ variant of $foldr$, which folds from the left rather than the right.

```
foldl :: (a → b → a) → a → [b] → a
foldl f z []       = z
foldl f z (x : xs) = foldl f (f z x) xs

sumAcc l acc = foldl (+) acc l
```

However, these functions too can be expressed in terms of $foldr$. The trick is to see such a function not as *having* an extra parameter but as *returning* a function that takes a parameter. For instance, $sumAcc$ should not be considered as a function that takes a list and an integer to an integer, but a function that takes a list to a function that takes an integer to an integer. This becomes more apparent when we make the precedence of the function arrow in the type signature explicit, as well as the binders for the extra parameter.

```
sumAcc :: [Int] → (Int → Int)
sumAcc []       = λacc → acc
sumAcc (x : xs) = λacc → sumAcc xs (x + acc)
```

Now we have an obvious catamorphism without extra parameter that can be turned trivially into a fold.

```
sumAcc l = foldr (λx xs acc → xs (x + acc))
                 (λacc → acc) l
```

As a last step we may want to η-expand the above definition.

```
sumAcc l acc = foldr (λx xs acc → xs (x + acc))
                     (λacc → acc) l acc
```

Finally, $foldl$ is an example of a function with both a constant parameter f and an accumulating parameter z. It is expressed as a $foldr$ thus:

```
foldl f z l = foldr (λx xs z → xs (f x z)) (λz → z) l z
```

Note that as a minor complication both extra parameters precede the list parameter.

Other Datatypes The above ideas for folds of lists easily generalize to other inductively defined algebraic datatypes. We illustrate that on leaf trees.

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

The $sumTree$ function is an example of a directly recursive catamorphism over leaf trees.

```
sumTree :: Tree Int → Int
sumTree (Leaf x)     = x
sumTree (Branch l r) = sumTree l + sumTree r
```

This catamorphic recursion scheme can be captured in a fold function too. The generic idea is that a fold function transforms the inductive datatype into a value of a user-specified type r by replacing every constructor with a user-specified function.

```
foldT :: (a → r)
      → (r → r → r)
      → Tree a
      → r
foldT l _ (Leaf x)     = l x
foldT l b (Branch x y) =
  b (foldT l b x) (foldT l b y)
```

The fold over leaf trees enables us to define $sumTree$ succinctly as

```
sumTree = foldT id (+)
```

Leaf trees also have a counterpart of left folds for lists, where the pattern is called *downward accumulation*. For instance, the following function from [9] labels every leaf with its depth:

```
depths :: Tree a → Int → Tree Int
depths (Leaf x)       d = Leaf d
depths (Branch l r) d = Branch (depths l (d + 1))
                               (depths r (d + 1))
```

This catamorphism is rewritten into a fold in a similar way as left folds.

```
depths t d =
  foldT (λd → Leaf d)
        (λl r d → Branch (l (d + 1)) (r (d + 1))) t d
```

2.2 Builds

Builds are the opposite of folds: they *produce* datastructures. For lists, this production pattern is captured in the function *build*.

```
build :: (∀b.(a → b → b) → b → b) → [a]
build g = g (:) []
```

where g is a function that builds an abstract list in terms of the provided 'cons' and 'nil' functions. The *build* function constructs a concrete list by applying g to the conventional $(:)$ and $[]$ list constructors.

Consider for instance the function *replicate*.

```
replicate :: Int → a → [a]
replicate n x
  | n ⩽ 0      = []
  | otherwise = x : replicate (n − 1) x
```

which can be expressed in terms of *build* as follows:

```
replicate n x = build (λcons nil →
  let g n x
    | n ⩽ 0      = nil
    | otherwise = cons x (g (n − 1) x)
  in g n x)
```

Other Datatypes The notion of a build function also generalizes. For instance, this is the build function for leaf trees:

```
buildT :: (∀b.(a → b) → (b → b → b) → b)
       → Tree a
buildT g = g Leaf Branch
```

With this build function we can express a producer of leaf trees

```
range :: Int → Int → Tree a
range l u
  | u > l      = let m = l + div (u − l) 2
                 in Branch (range l m) (range (m + 1) u)
  | otherwise = Leaf l
```

as a build:

```
range l u =
  buildT (λleaf branch →
    let g l u
      | u > l      = let m = l + div (u − l) 2
                     in branch (g l m) (g (m + 1) u)
      | otherwise = leaf l
    in g l u)
```

2.3 Fold/Build Fusion

The foldr/build fusion rule expresses that a consumer and producer can be fused:

```
foldr cons nil (build g)  ≡  g cons nil
```

Consider the following expressions:

```
sum (replicate n x)
```

which consumes a list with *sum* after generating it with *replicate*. With the help of the fusion rule and simple transformations, we can optimize it as follows.

```
  sum (replicate n x)
≡  ⟦ inline sum ⟧
  foldr (+) 0 (replicate n x)
≡  ⟦ inline replicate ⟧
  foldr (+) 0 (build $ λcons nil →
    let g n x
      | n ⩽ 0      = nil
      | otherwise = cons x (g (n − 1) x)
    in g n x)
≡  ⟦ foldr/build fusion ⟧
  (λcons nil →
    let g n x
      | n ⩽ 0      = nil
      | otherwise = cons x (g (n − 1) x)
    in g n x) (+) 0
≡  ⟦ β-reduction ⟧
  let g n x
    | n ⩽ 0      = 0
    | otherwise = x + g (n − 1) x
  in g n x
```

Although arguable less readable, we can see that the final version of $sum\ (replicate\ n\ x)$ does not create an intermediate list.

Pipelines Haskell best practices encourage building complicated functions by producing, repeatedly transforming and finally consuming an intermediate datastructure in a pipeline. An example of such a pipeline is the following sum-of-odd-squares function:

```
sumOfSquaredOdds :: Int → Int
sumOfSquaredOdds =
  sum ∘ map (^2) ∘ filter odd ∘ enumFromTo 1
```

This code is quite compact and easy to compose from existing library functions. However, when compiled naively, it is also rather inefficient because it performs four loops and allocates three intermediate lists (the results of respectively *enumFromTo* 1, *filter odd* and *map* $(\hat{}2)$) that are immediately consumed again.

With the help of fusion, whole pipelines like *sumOfSquaredOdds* can be fused into one recursive function:

```
sumOfSquaredOdds :: Int → Int
sumOfSquaredOdds n = go 1
  where
    go :: Int → Int
    go x
      | x > n      = 0
      | odd x      = x ^ 2 + go (x + 1)
      | otherwise = go (x + 1)
```

which performs only a single loop and allocates no intermediate lists.

The key to pipeline optimization is the fact that *transformation functions* like *map* and *filter* can be expressed as a *build* whose generator function is a *fold*.

$$
\begin{array}{l}
map :: (a \to b) \to [a] \to [b] \\
map\ f\ l = build\ (\lambda cons\ nil \to foldr\ (cons \circ f)\ nil\ l) \\
filter :: (a \to Bool) \to [a] \to [a] \\
filter\ p\ l = \\
\quad build\ (\lambda cons\ nil \to foldr\ (\lambda x \to \mathbf{if}\ p\ x\ \mathbf{then}\ cons\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ id)\ nil\ l)
\end{array}
$$

This means they can fuse with both a consumer on the left and a producer on the right.

Other Datatypes Any datatype that provides both a fold and build function, has a corresponding fusion law. For leaf trees, this law is:

$$foldT\ leaf\ branch\ (buildT\ g) \quad \equiv \quad g\ leaf\ branch$$

It is key to fusing two loops into one:

$$
\begin{array}{ll}
 & sumTree\ (range\ l\ u) \\
\equiv & [\![\ \text{inline } sumTree\ ]\!] \\
 & foldT\ id\ (+)\ (range\ l\ u) \\
\equiv & [\![\ \text{inline } range\ ]\!] \\
 & foldT\ id\ (+)\ (buildT\ \$\ \lambda leaf\ branch \to \\
 & \quad \mathbf{let}\ g\ l\ u \\
 & \qquad \mid u > l \qquad = \mathbf{let}\ m = l + div\ (u - l)\ 2 \\
 & \qquad\qquad\qquad\ \ \mathbf{in}\ branch\ (g\ l\ m)\ (g\ (m+1)\ u) \\
 & \qquad \mid otherwise = leaf\ l \\
 & \quad \mathbf{in}\ g\ l\ u) \\
\equiv & [\![\ \text{fold/build fusion}\ ]\!] \\
 & (\lambda leaf\ branch \to \\
 & \quad \mathbf{let}\ g\ l\ u \\
 & \qquad \mid u > l \qquad = \mathbf{let}\ m = l + div\ (u - l)\ 2 \\
 & \qquad\qquad\qquad\ \ \mathbf{in}\ branch\ (g\ l\ m)\ (g\ (m+1)\ u) \\
 & \qquad \mid otherwise = leaf\ l \\
 & \quad \mathbf{in}\ g\ l\ u)\ id\ (+) \\
\equiv & [\![\ \beta\text{-reduction}\ ]\!] \\
 & \mathbf{let}\ g\ l\ u \\
 & \quad \mid u > l \qquad = \mathbf{let}\ m = l + div\ (u - l)\ 2 \\
 & \qquad\qquad\ \ \mathbf{in}\ g\ l\ m + g\ (m+1)\ u \\
 & \quad \mid otherwise = l \\
 & \mathbf{in}\ g\ l\ u
\end{array}
$$

2.4 Automation

The main Haskell compiler, GHC, automates fold/build fusion by means of its rewrite rules infrastructure [11]. The fusion law is expressed as a rewrite rule in the GHC.Base module

```
{-# RULES
"fold/build"
   forall k z (g::forall b. (a->b->b) -> b -> b) .
   foldr k z (build g) = g k z
 #-}
```

which is applied whenever the optimizer encounters the pattern on the left. In addition, various library functions have been written[2] in terms of *foldr* and *build*. Whenever the programmer uses these functions and combines them with his own uses of *foldr* and *build*, fusion may kick in.

Unfortunately, GHC's infrastructure shows two main weaknesses:

1. GHC does not explicitly cater for other datatypes than lists. While the programmer can replicate the existing list infrastructure for his own datatypes, it requires a considerable effort and rarely happens in practice.
2. Programmers or libraries need to explicitly define their functions in terms of *foldr* and *build*. If they don't, then fusion is not possible. This too turns out to be too much to ask as we see many uses of explicit recursion in practice (see Section 6).

This work addresses both limitations. It allows programmers to write their functions in explicitly recursive style and performs fold/build fusion for any directly inductive datatype.

3. Finding Folds

This section explains our approach to turning explicitly recursive functions into invocations of *fold*.

3.1 Syntax and Notation

To simplify the presentation, we do not explain our approach in terms of Haskell source syntax or even GHC's core syntax (based on System F). Instead, we use the untyped lambda-calculus extended with constructors and pattern matching, and (possibly recursive) bindings.

$$
\begin{array}{llll}
\text{binding} & b & ::= & x = e \\
\text{pattern} & p & ::= & K\ \overline{x} \\
\text{expression} & e & ::= & x \\
 & & \mid & e\ e \\
 & & \mid & \lambda x \to e \\
 & & \mid & K \\
 & & \mid & \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to e}
\end{array}
$$

The extension to GHC's full core syntax, including types, is relatively straightforward.

We will also need an advanced form of *(expression) context*:

$$
\begin{array}{lll}
E & ::= & x \\
 & \mid & E\ x \\
 & \mid & E\ \square \\
 & \mid & E\ \triangle
\end{array}
$$

A context E denotes a function applied to a number of arguments. The function itself and some of its arguments are given (as variables), while there are holes for the other arguments. In fact, there are two kinds of holes: boxes $\square$ and triangles $\triangle$. The former is used for a sequence of extra parameters, while the latter marks the main argument on which structural recursion is performed. The function $E[\overline{e}; e]$ turns a context E into an expression by filling in the holes with the given expressions.

$$
\begin{array}{lcl}
x[\epsilon; e] & = & x \\
(E\ x)[\overline{e}; e] & = & E[\overline{e}; e]\ x \\
(E\ \square)[\overline{e}, e_1; e] & = & E[\overline{e}; e]\ e_1 \\
(E\ \triangle)[\overline{e}; e] & = & E[\overline{e}; e]\ e
\end{array}
$$

Note that this function is partial; it is undefined if the number of expressions $\overline{e}$ does not match the number of box holes.

3.2 Finding Folds

Figure 1 shows our non-deterministic algorithm for rewriting function bindings in terms of folds. To keep the exposition simple, the algorithm is specialized to folds over lists; we discuss the generalization to other algebraic datatypes later on.

Single-Argument Functions The top-level judgement is of the form $b \rightsquigarrow b'$, which denotes the rewriting of a function binding b to b'. The judgement is defined by a single rule (F-Bind), but we

[2] or come with rewrite rules that rewrite them into those forms

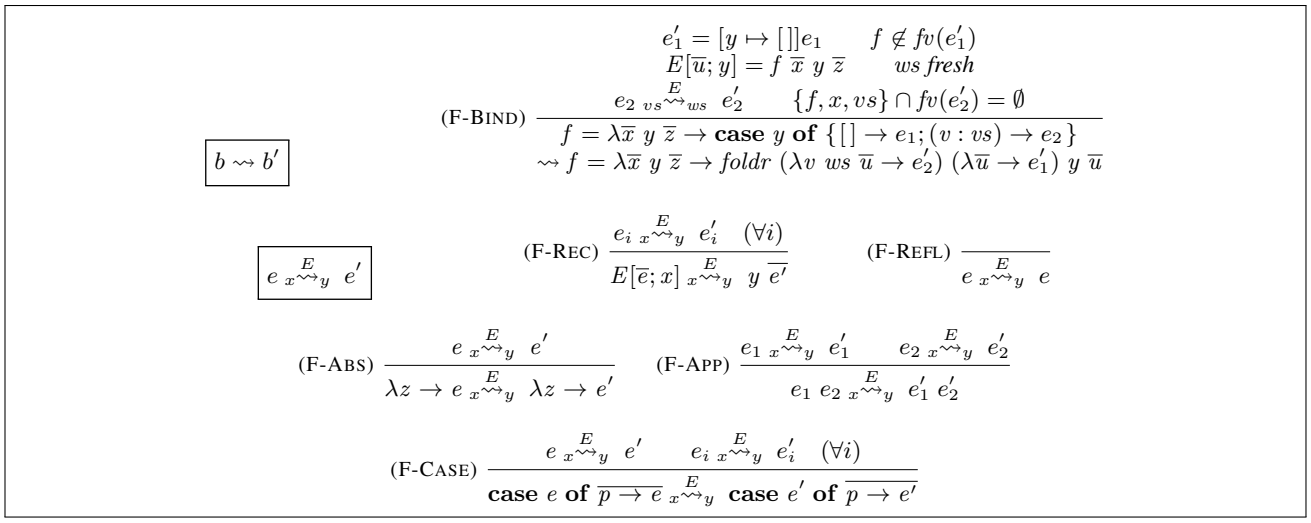

Figure 1. Fold discovery rules

first explain a specialized rule for single argument functions:

$$\text{(F-BIND')}\ \frac{e_1' = [y \mapsto [\,]]e_1 \quad f \notin fv(e_1) \quad ws\ fresh \quad e_2\ {}_{vs}\overset{f\ \triangle}{\rightsquigarrow}_{ws}\ e_2' \quad \{f, x, vs\} \cap fv(e_2') = \emptyset}{f = \lambda y \rightarrow \mathbf{case}\ y\ \mathbf{of}\ \{[\,] \rightarrow e_1; (v : vs) \rightarrow e_2\} \rightsquigarrow f = \lambda y \rightarrow foldr\ (\lambda v\ ws \rightarrow e_2')\ e_1'\ y}$$

This rule rewrites a binding like

$$\begin{array}{l} sum = \lambda y \rightarrow \mathbf{case}\ y\ \mathbf{of} \\ \quad [\,] \quad\ \ \rightarrow 0 \\ \quad (v : vs) \rightarrow (+)\ v\ (sum\ vs) \end{array}$$

into

$$sum = \lambda y \rightarrow foldr\ (\lambda v\ ws \rightarrow (+)\ v\ ws)\ 0\ y$$

mostly by simple pattern matching and replacement. The main work is to replace all recursive calls in e_2, which is handled by the auxiliary judgement of the form

$$e\ {}_{x}\overset{E}{\rightsquigarrow}_{y}\ e'$$

which is defined by five rewriting rules: rule (F-REC) takes care of the actual rewriting of recursive calls, while rule F-REFL provides the reflexive closure and the other three rules provide congruence closure. In the restricted setting of single argument functions rule (F-Rec) takes the simplified form

$$\text{(F-REC')}\ \frac{}{f\ x\ {}_{x}\overset{f\ \triangle}{\rightsquigarrow}_{y}\ y}$$

Hence, we have $sum\ vs\ {}_{vs}\overset{sum\ \triangle}{\rightsquigarrow}_{ws}\ ws$.

The side-conditions on the (F-BIND') rule make sure that the function being rewritten is a proper catamorphism.

1. If vs appears in e_2' as part of $f\ \ vs$, the latter has not been properly replaced by ws. If vs appears in any other capacity, then the function is not a catamorphism, but a *paramorphism*. An example of such a function, is the function $suffixes$.

   $$\begin{array}{l} suffixes = \lambda y \rightarrow \mathbf{case}\ y\ \mathbf{of} \\ \quad [\,] \quad\ \ \rightarrow [\,] \\ \quad (v : vs) \rightarrow vs : suffixes\ vs \end{array}$$

   This function can be written as

   $$suffixes = para\ (\lambda v\ vs\ ws \rightarrow vs : ws)\ [\,]$$

   where the higher-order pattern of paramorphisms is

   $$\begin{array}{l} para :: (a \rightarrow [a] \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ para\ f\ z\ [\,] \quad\ \ = z \\ para\ f\ z\ (x : xs) = f\ x\ xs\ (para\ f\ z\ xs) \end{array}$$

2. If f appears in any other form than as part of recursive calls of the form $f\ \ vs$, then again the function is not a proper catamorphism. An example of that case is the following non-terminating function:

   $$\begin{array}{l} f = \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\ \quad [\,] \quad\ \ \rightarrow 0 \\ \quad (v : vs) \rightarrow v + f\ vs + f\ [1, 2, 3] \end{array}$$

Note that we know in the branches e_1 and e_2 the variable x is an alias for respectively $[\,]$ or $(y : ys)$. Rule (F-BIND') exploits the former case to eliminate x in e_1'. The latter case reveals an improper catamorphism, where ys appears outside of a recursive call (issue 2 above); hence, rule (F-BIND') does not allow it.

Folds with Parameters Rule (F-BIND) generalizes rule (F-BIND') by supporting additional formal parameters $\overline{x}$ and $\overline{z}$ before and after the scrutinee argument y. The algorithm supports both constant and accumulating parameters. Rule (F-BIND) does not explicitly name the constant parameters, but captures them instead in the context E for recursive calls. For instance, for cat the context has the form $cat\ \triangle\ ys$ where ys is the constant parameter.

Rule (F-Bind) names the accumulating parameters $\overline{u}$ and leaves $\Box$ holes in the context E for them. For instance, the context of $sumAcc$ is $sumAcc\ \triangle\ \Box$. Because the $\overline{u}$ arguments vary throughout the iteration, their current and new values need to be bound, respectively supplied, at every step. The binding of the current values is taken care of by the binders $\lambda \overline{u} \rightarrow$ in the two first parameters of $foldr$. Also the initial values for $\overline{u}$ are supplied as extra parameters to $foldr$.

Rule (F-REC) captures the new values $\overline{e}$ for the accumulating parameters with the help of the context E; a recursive call takes the form $E[\overline{e}; vs]$. The rule passes these variant arguments explicitly to the recursive result ws.

Note that rule (F-REC) recursively rewrites the accumulating parameters $\overline{e}$ because they may harbor further recursive calls. For instance,

$$f = \lambda y\ acc \to \mathbf{case}\ y\ \mathbf{of}$$
$$\quad [\,] \qquad \to acc$$
$$\quad (v : vs) \to f\ vs\ (f\ vs\ (v + acc))$$

is rewritten to

$$f = \lambda y\ acc \to foldr\ (\lambda v\ ws\ acc \to ws\ (ws\ (v + acc)))$$
$$\quad (\lambda acc \to acc)\ y\ acc$$

3.3 Degenerate folds

Our algorithm also transforms certain non-recursive functions into folds. For instance, it rewrites

$$head :: [a] \to a$$
$$head = \lambda l \to \mathbf{case}\ l\ \mathbf{of}$$
$$\quad [\,] \qquad \to error\ \texttt{"empty list"}$$
$$\quad (x : xs) \to x$$

into

$$head :: [a] \to a$$
$$head = \lambda l \to foldr\ (\lambda z\ zs \to z)\ (error\ \texttt{"empty list"})\ l$$

These *degenerate* folds are of no interest to us, since non-recursive functions are much more easily understood in their conventional form than as a fold. Moreover, they can easily be optimized without fold/build fusion: by simple inlining and specialization. Fortunately we can easily avoid introducing degenerate folds by only rewriting recursive functions. In other words, the algorithm must use the rule (F-REC) at least once.

3.4 Other Datatypes

The algorithm generalizes straightforwardly to other datatypes than lists. The main issues are to cater for the datatype's constructors rather than those of lists, and to use the datatype's fold function rather than $foldr$.

A significant generalization from lists is that datatype constructors may have multiple recursive subterms. For instance, the $Branch$ constructor of our leaf trees has two recursive subterms, for the left and right subtrees. This means that the (F-Rec) rule has to allow for recursive calls over either. Also note that in the case of multiple recursive subterms, the recursive rewriting of accumulating parameters in rule F-REC is more likely to occur. Consider for instance the following function

$$flatten :: Tree\ a \to [a] \to [a]$$
$$flatten = \lambda t\ acc \to \mathbf{case}\ t\ \mathbf{of}$$
$$\quad Leaf\ x \qquad \to (x : acc)$$
$$\quad Branch\ l\ r \to flatten\ l\ (flatten\ r\ acc)$$

which is turned into

$$flatten = \lambda t\ acc \to foldT\ (\lambda x\ acc \to (x : acc))$$
$$\qquad\qquad\qquad\qquad\qquad (\lambda l\ r\ acc \to l\ (r\ acc))\ t\ acc$$

4. Finding Builds

Figure 2 lists our non-deterministic algorithm for finding list builds. The toplevel judgement is $b \rightarrowtail b'; b_g$. It rewrites a binding b into b' that uses $build$ and also returns an auxiliary binding b_g for the generator function used in the $build$. There is one rule defining this judgement, (B-BIND), that rewrites the body e of a binding into a $build$ and produces the generator function binding. Note that the rule allows for an arbitrary number of lambda abstractions $\lambda\overline{x} \to$ to precede the invocation of $build$. This allows auxiliary parameters to the generator function, e.g., to support an inductive definition. For instance, the map function can be written as a $build$ with two auxiliary parameters.

$$\boxed{b \rightarrowtail b'; b''} \qquad \text{(B-BIND)}\ \frac{c, n, g \text{ fresh} \quad e\ {}_f\overset{c,n}{\rightarrowtail}_g\ e'}{\begin{array}{r} f = \lambda\overline{x} \to e\ \rightarrowtail \\ f = \lambda\overline{x} \to build\ (g\ \overline{x}); \\ g = \lambda\overline{x} \to \lambda c \to \lambda n \to e' \end{array}}$$

$$\boxed{e\ {}_f\overset{c,n}{\rightarrowtail}_g\ e'} \qquad \text{(B-REC)}\ \frac{}{f\ \overline{e}\ {}_f\overset{c,n}{\rightarrowtail}_g\ g\ \overline{e}\ c\ n}$$

$$\text{(B-NIL)}\ \frac{}{[\,]\ {}_f\overset{c,n}{\rightarrowtail}_g\ n} \qquad \text{(B-CONS)}\ \frac{e_2\ {}_f\overset{c,n}{\rightarrowtail}_g\ e_2'}{(e_1 : e_2)\ {}_f\overset{c,n}{\rightarrowtail}_g\ c\ e_1\ e_2'}$$

$$\text{(B-BUILD)}\ \frac{}{build\ e\ {}_f\overset{c,n}{\rightarrowtail}_g\ e\ c\ n}$$

$$\text{(B-CASE)}\ \frac{e_i\ {}_f\overset{c,n}{\rightarrowtail}_g\ e_i' \quad (\forall i)}{\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to e}\ {}_f\overset{c,n}{\rightarrowtail}_g\ \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to e'}}$$

Figure 2. Build discovery rules

$$map = \lambda f \to \lambda l \to build\ (g\ f\ l)$$
$$g = \lambda f \to \lambda l \to \lambda c \to \lambda n \to$$
$$\quad \mathbf{case}\ l\ \mathbf{of}$$
$$\quad\quad [\,] \qquad \to n$$
$$\quad\quad (y : ys) \to c\ (f\ y)\ (g\ f\ ys\ c\ n)$$

Only the f parameter is constant. The l parameter changes in inductive calls as g is defined inductively over it.

The toplevel judgement is defined in terms of the auxiliary judgement

$$e\ {}_f\overset{c,n}{\rightarrowtail}_g\ e'$$

that yields the body of the generator function. This judgement is defined by five different rules, the last of which, (B-CASE), is merely a congruence rule that performs the rewriting in the branches of a **case** expression. The first four rules distinguish four ways in which the function body can yield a list.

1. Rule (B-NIL) captures the simplest way for producing a list, namely with $[\,]$, which is rewritten to the new parameter n.
2. Rule (B-CONS) replaces the $(:)$ constructor with the new parameter c, and recursively rewrites the tail of the list.
3. Rule (B-REC) replaces recursive calls to the original function f by recursive calls to the generator function g.
4. Rule (B-BUILD) deals with the case where a list is produced by a call to $build$. In this situation the abstract c and n list constructors can be introduced dynamically as $foldr\ c\ n\ (build\ e)$. However, this expression can of course be statically fused to $e\ c\ n$.

In the map example, all rules but (B-BUILD) are used. Here is an example that does use rule (B-BUILD).

$$toFront :: Eq\ a \Rightarrow a \to [a] \to [a]$$
$$toFront\ x\ xs = x : filter\ (\not\equiv x)\ xs$$

After inlining $filter$, which itself is expressed as a build, this function becomes.

$$toFront = \lambda x \to \lambda xs \to x : build\ (g\ (\not\equiv x)\ xs)$$

which can be transformed into:

$$toFront = \lambda x \to \lambda xs \to build\ (g'\ x\ xs)$$
$$g' = \lambda x \to \lambda xs \to \lambda c \to \lambda n \to c\ x\ (g'\ (\not\equiv x)\ xs\ c\ n)$$

Degenerate Builds Just like non-recursive catamorphisms we can also call non-recursive builds degenerate. For instance,

$$f = 1 : 2 : 3 : [\,]$$

is rewritten to

$$f = build\ g$$
$$g = \lambda c \to \lambda n \to c\ 1\ (c\ 2\ (c\ 3\ n))$$

These functions can be easily identified by the absence of a recursive call. In other words, the rule (B-REC) is never used in the rewriting process.

Strictly speaking, such non-recursive functions do not require *fold*/*build* fusion to be optimized. They can also be optimized by inlining their definition and then unfolding the catamorphism sufficiently to consume the whole list.

$$\begin{array}{ll} & sum\ f \\ \equiv & \quad [\![\ \text{inline } f\ ]\!] \\ & sum\ (1 : 2 : 3 : [\,]) \\ \equiv & \quad [\![\ \text{unfold } sum\ ]\!] \\ & 1 + sum\ (2 : 3 : [\,]) \\ \equiv & \quad [\![\ \text{unfold } sum \text{ three more times}\ ]\!] \\ & 1 + (2 + (3 + 0)) \end{array}$$

However, in practice, GHC does not perform such aggressive inlining by default. Hence, *fold*/*build* fusion is still a good way of getting rid of the intermediate datastructure.

Other Datatypes The adaptation of the build algorithm to other datatypes is essentially similar to the adaptations we had to make for the fold algorithm: cater for a different set of constructors with other recursive positions and use the datatype's build function.

5. Implementation

Rather than to implement our algorithms as a source-level program transformation, we have implemented our algorithms as compiler passes in a plugin [20] for GHC. This has three important advantages:

- Firstly, the compiler passes work with GHC's core representation, which is much simpler than Haskell source syntax. For instance, the type that represents Haskell source expressions in the popular `haskell-src-exts` package features 46 different constructors, while the corresponding GHC core type has only 10.
- Secondly, GHC optimizer performs various beneficial transformation passes on a program's core representation before running our algorithms. Many of these passes help our algorithms by simplifying the source code. Consider the following example of two mutually recursive functions.

  $$f :: [Int] \to Int$$
  $$f\ [\,] = 1$$
  $$f\ (x : xs) = g\ x\ xs$$
  $$g :: Int \to [Int] \to Int$$
  $$g\ x\ xs = x * f\ xs + 1$$

  Our fold finding algorithm does not identify f as a catamorphism, because it does not see the recursive call, which is hidden in g. However, GHC's inlining pass is likely to inline g in the body of f and thus expose the recursive call.

  In other words, we can keep our algorithms relatively simple, because other GHC passes already do part of the work for us.
- Finally, GHC core is fully typed. While type information is not essential, our algorithms can make good use of it to improve their performance. Consider this simple function:

  $$add :: Int \to Int \to Int$$
  $$add\ x\ y = x + y$$

  The type signature reveals that none of the parameters is an inductively defined datatype, nor is the result type. As a consequence, without looking a the function body, our algorithms can conclude that the function cannot be expressed as a fold or a build.

5.1 Algorithm Implementation

We have implemented the two algorithms as separate passes in our plugin. The implementations deviate from the non-deterministic algorithms of Sections 3 and 4 on two accounts:

- They deal with the core language, which is slightly larger than the language used earlier. As already indicated, core carries type information, which our implementation must adjust when performing the transformation. Core also involves local binders in addition to toplevel binders. We analyze these too for occurrences of folds and builds.
- Our implementation is of course deterministic rather than non-deterministic, and, as described above, based on the available type information, it tries to fail fast.

Below are two more important points with respect to the quality of obtained fusions.

Local Generator Binders An important point for the *build* finding algorithm is that the new abstract generator function is actually introduced as a local binding, rather than a new toplevel binding. The reason is that we want GHC to specialize the generator function at the use site after fusion.

For instance, after the inlining and fusion of $sum\ (replicate\ n)$, we get $g\ (+)\ 0$ where g is the abstract generator function. If g is defined with a toplevel binder, GHC will not inline it because it is recursive. Hence, while the intermediate datastructure has been eliminated, we still pay the price of the generator's abstraction.

However, if g is actually defined locally in *replicate*, as shown in Section 2.2, GHC can easily specialize the recursive definition of the generator and eliminate the abstraction. The result is a tight first-order loop.

Relative Pass Scheduling As we have already argued in Section 2.3, it is crucial for the fusion of pipelines that transformation functions are expressed as a build of a fold. In order to obtain this required form, our build finding pass must be scheduled before the fold finding pass. Only in this order do we obtain, e.g., the definition

$$map\ f\ l = build\ (\lambda c\ n \to foldr\ (\lambda x\ xs \to c\ (f\ x)\ xs)\ n\ l)$$

If we first find the fold, we get stuck at the following intermediate form

$$map\ f\ l = foldr\ (\lambda x\ xs \to f\ x : xs)\ [\,]\ l$$

because our build finder is not equipped to deal with folds.

A more heavyweight solution would be to extend the build finding algorithm with a rule for handling folds.

$$\text{(B-FOLD)}\ \frac{e_1\ {}_f\overset{c,n}{\rightarrowtail}_g\ e_1' \qquad e_2\ {}_f\overset{c,n}{\rightarrowtail}_g\ e_2'}{\begin{array}{l} foldr\ (\lambda x\ xs \to e_1)\ e_2\ e_3 \\ {}_f\overset{c,n}{\rightarrowtail}_g\ foldr\ (\lambda x\ xs \to e_1')\ e_2'\ e_3 \end{array}}$$

Such a rule has the added benefit of handling handling folds introduced by the programmer. However, because it is not powerful

enough to deal with accumulating parameters, it remains best to schedule build finding first.

5.2 Fusion

The passes for finding folds and builds are inherently compatible with GHC's approach to list fusion. Whenever a list producer or consumer is rewritten into a *foldr* or *build*, it becomes a possible subject of the GHC rewrite rules that target these higher-order schemes.

Moreover, for fusion to happen it is not necessary for our passes to find a fold and a build together. Fusion can also happen when the programmer combines, e.g., his own explicitly recursive catamorphism with a library function that is already expressed as a build.

5.3 Datatype Support

In order to introduce calls to fold and build functions for a datatype, these functions have to be available for that datatype. At present, GHC only provides such functions for lists.

In order to support additional datatypes, we allow the programmer to register fold and build functions for his own datatypes by means of annotations [19]. For instance, the following annotation does so for the type of leaf trees.

```
{-# ANN type Tree
     (RegisterFoldBuild "foldT" "buildT")
  #-}
```

Moreover, we also provide complimentary Template Haskell [17] routines to derive the implementations of the two functions.

```
$ (deriveFold ‘Tree "foldT")
$ (deriveBuild ‘Tree "buildT")
```

Similarly, to support fusion for other datatypes, we follow GHC's rewriting rules approach for lists. However, instead of having to write the fusion rewrite rule explicitly, we provide a handy Template Haskell routine. For instance,

```
$ (deriveFusion ‘Tree "foldT" "buildT")
```

generates

```
{-# "foldT/buildT-fusion"
   forall l b
    (g :: forall b. (a -> b) -> (b -> b -> b) -> b).
   foldT l n (buildT g) = g l b
  #-}
```

If desired, the responsibility for registering types and generating the higher-order schemes and rewrite rules can easily be moved to the compiler. At this time, and for the purpose of evaluation, it suits us to have a bit more control.

6. Evaluation

6.1 Identifying Folds

In order to test the quality of our fold finding algorithm, we have applied it to 13 popular Haskell packages during a compilation with GHC 7.6.3. We have not enabled any of the compiler's optimization flags and disabled the actual rewriting in our pass. In this way, we get an accurate estimate (a lower bound) of the number of explicitly recursive catamorphisms that experienced Haskell programmers write in practice.

Table 1 lists the results of the fold analysis. The first column lists the names of the packages and the second column (Total) reports the total number of discovered catamorphisms. The third and fourth column split up this total number in catamorphisms over lists (List) and catamorphisms over other algebraic datatypes (Other). The fifth column (Acc.) shows the number of catamorphisms with accumulating parameters (e.g., left folds). The sixth column (N. rec.) counts the number of accumulating parameter catamorphisms with nested recursive calls such as the $f\ \ vs\ \ (f\ \ vs\ \ (v + acc))$ example from Section 2.1.

Package	**Total**	List	Other	Acc.	N. rec.	**HLint**
Cabal-1.16.0.3	20	11	9	6	0	9
containers-0.5.2.1	100	11	89	41	11	1
cpphs-1.16	5	2	3	3	0	1
darcs-2.8.4	66	65	8	1	0	6
ghc-7.6.3	327	216	111	127	9	26
hakyll-4.2.2.0	5	1	4	3	0	0
haskell-src-exts-1.13.5	37	11	26	15	0	2
hlint-1.8.44	6	3	3	1	0	0
hscolour-1.20.3	4	4	0	0	0	2
HTTP-4000.2.8	6	6	0	2	0	3
pandoc-1.11.1	15	15	0	1	0	2
parsec-3.1.3	3	3	0	1	0	0
snap-core-0.9.3.1	4	3	1	1	0	0

Table 1. Folds found in well-known Haskell packages

Finally, the last column provides the analysis results of *hlint* [15] for comparison. This tool provides hints on how to refactor Haskell source code. The listed results reflect the number of suggestions on the use of *map*, *filter*, *foldl* and *foldr* rather than explicit recursion.

We see that across all packages our analysis finds more list catamorphisms than hlint. In fact, our tool discovers some list catamorphisms in hlint's own source code that hlint cannot find itself. Moreover, in addition to the results in the table, our tool also successfully identifies all the list folds in the hlint test suite. There are three other packages in which hlint does not find any catamorphisms. This is likely due to the fact that the authors of those packages have themselves used hlint to discover and eliminate explicit recursion.

We attribute the better results of our tool partly to the fact that more catamorphisms are exposed in the core representation by GHC's program transformations. However, to a large extent, the better results are due to the fact that our analysis is more powerful than that of hlint. Also, hlint does not look for catamorphisms over other datatypes than lists, while several packages do have a significant number of those.

Folds with a variant arguments (left folds) are found quite regularly in Haskell packages, but those with nested recursive calls are much rarer. We have only found them in the GHC and containers packages; they may be an indication of a rather advanced programming style.

6.2 Identifying builds

Table 2 shows the results of the build analysis for the 13 packages. The Total column lists the total number of builds per package, while the List and Other columns split up this number into list builds and builds of other datatypes. Column Rec show the number of recursive builds.

Our main observation is that the numbers are roughly proportional to those of the folds. Unlike for folds, we are not aware of any other tool that detects builds and would provide a basis for comparison.

Package	**Total**	List	Other	Rec.
Cabal-1.16.0.3	101	81	20	5
containers-0.5.2.1	25	2	23	12
cpphs-1.16	6	5	1	3
darcs-2.8.4	354	354	0	26
ghc-7.6.3	480	178	302	53
hakyll-4.2.2.0	22	18	4	2
haskell-src-exts-1.13.5	140	74	66	16
hlint-1.8.44	69	62	7	1
hscolour-1.20.3	33	33	0	2
HTTP-4000.2.8	11	11	0	5
pandoc-1.11.1	97	97	0	16
parsec-3.1.3	10	10	0	0
snap-core-0.9.3.1	4	4	0	0

Table 2. Builds found in well-known Haskell packages

6.3 Fusion

We have not measured any significant performance improvements in the above 13 packages. Likely, the critical paths in these packages have already been optimized by their authors.

Hence, instead, we offer the following artificial benchmarks, that demonstrate the potential impact of fusion on program runtime. We have two similar sets of benchmarks, one for lists and one for leaf trees, that consist of pipelines of increasing length. The $i$the benchmark consists of a producer ($upto$), followed by $i-1$ transformers ($map\ (+1)$ and a final consumer (sum). Each of the components of the pipeline is defined in an explicitly recursive style (see Appendix. A).

We have timed the pipelines with Criterion [16], using input $n = 100\,000$ on an Intel Core i3-2367M CPU @ 1.40GHz. Each of the benchmarks was compiled twice: once with the `-O2 -fenable-rewrite-rules` GHC flags, and once with those two flags and our compiler passes. We have inspected the produced core code and observed that in the former case the pipeline is not fused at all, and in the latter case it is fully fused. For instance, the fully fused code obtained for l_5 is:

$$
\begin{aligned}
&l_5 = \\
&\quad \lambda l\ u \to \mathbf{case}\ l > u\ \mathbf{of} \\
&\qquad \mathit{False} \to \mathbf{case}\ l_5\ (l+1)\ u\ \mathbf{of} \\
&\qquad\qquad n \to ((((l+1)+1)+1)+1)+n \\
&\qquad \mathit{True}\ \to 0
\end{aligned}
$$

Figure 3 shows the absolute runtimes and the speed-ups obtained by fusion for the list pipelines. The relative speed-ups are defined as $(t_u - t_f)/t_u$ where t_u is the runtime of the unfused pipelines and t_f the runtime of the fused pipelines.

We see in the unoptimized version that the shortest list pipeline l_1 has a base runtime of about 10ms; every additional transformation in the longer pipelines adds about 4ms. Our compiler passes cause big-speeds. Firstly, the base runtime is reduced by almost 80%. Moreover, the cost of the additional transformations is completely eliminated: all pipelines have the same absolute runtime. This means that the relative speed-up gradually converges to 100%.

Figure 4 Similar observations can be made for the leaf tree pipelines.

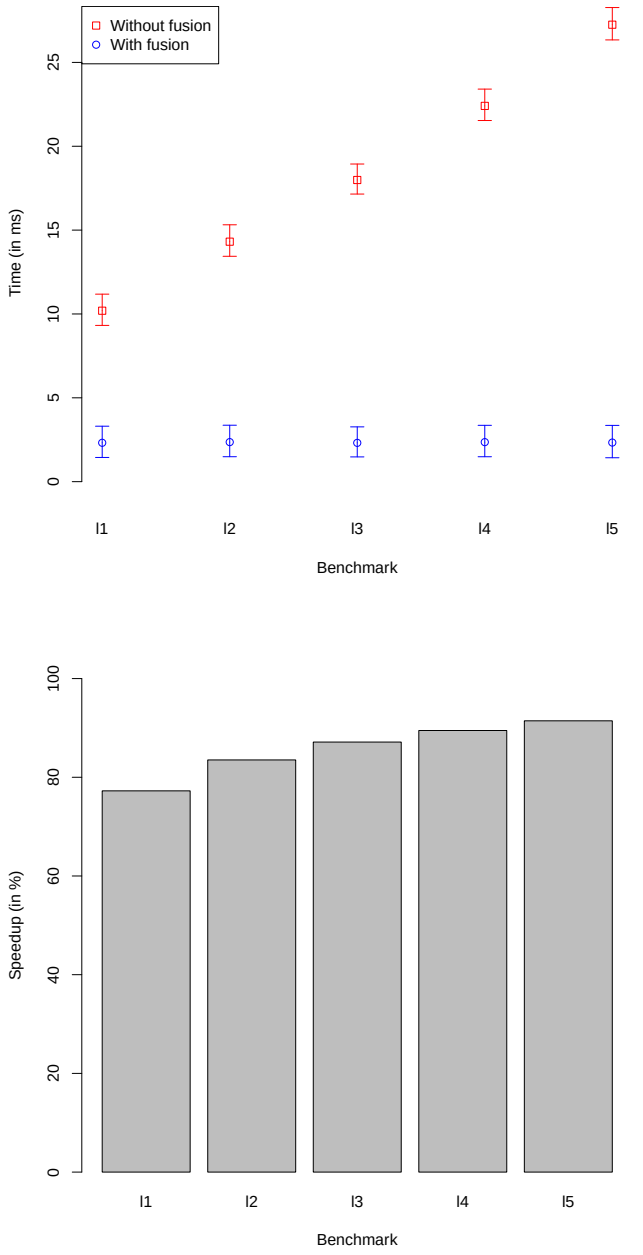

Figure 3. Absolute runtimes (top) and relative speed-ups (bottom) for list pipelines.

7. Related Work

7.1 Folds and Fusion

There is a long line of work on writing recursive functions in terms of structured recursion schemes, as well as proving fusion properties of these and exploiting them for deriving efficient programs. Typically the derivation process is performed manually.

Bird and Meertens [1, 13] have come up with several equational laws for recursion schemes to serve them in their work on program calculation. With their Squiggol calculus, Meijer et al. [14] promote

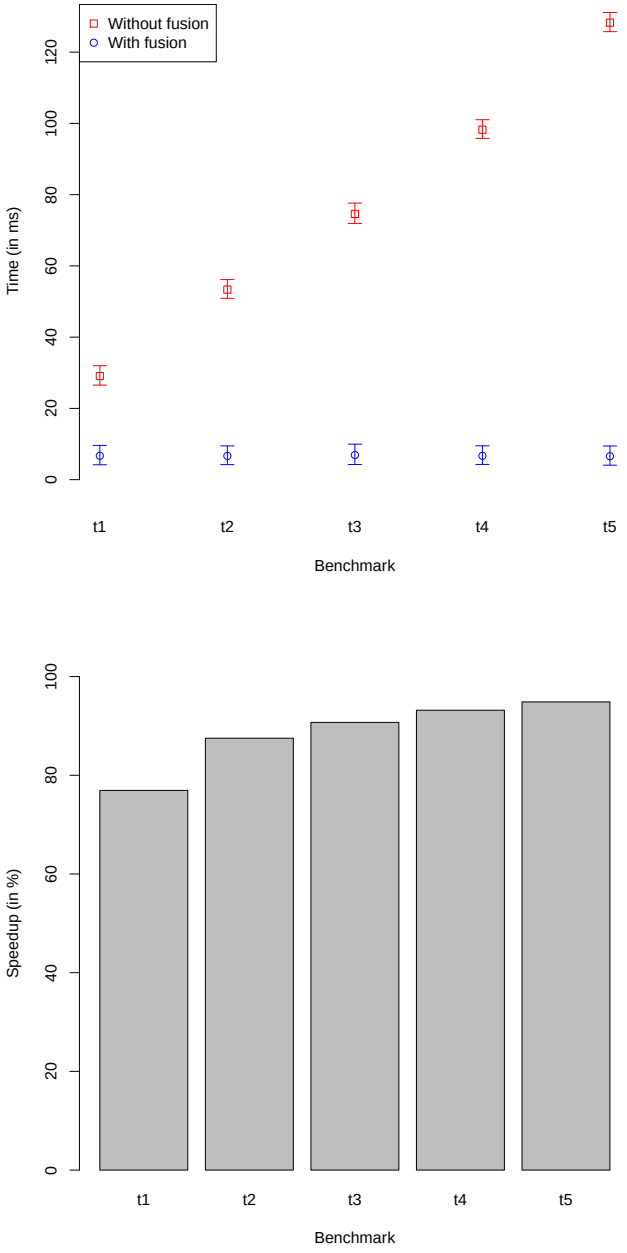

Figure 4. Absolute runtimes (top) and relative speed-ups (bottom) for leaf tree pipelines.

the use of structured recursion by means of recursion operators. These operators are defined in a datatype generic way and are equipped with a number of algebraic laws that enable equivalence-preserving program transformations.

Gibbons [6] promotes explicit programming in terms of folds and unfolds, which he calls *origami* programming. Unfolds are the dual of folds, and capture a special case of builds.

Gill et al. [7] present the foldr/build fusion rule and discuss its benefits. They mention that it would be desirable, yet highly challenging, for the compiler to notice whether functions can be expressed in terms of $foldr$ and $build$. That would allow programmers to write programs in whatever style they like.

Various authors have investigated variants of short-cut fusion where datastructures are produced and consumed in the context of some computational effect. Delbianco et al. [4] consider the case where the effect modeled by an applicative functor, and both Ghani & Johan [5] and Manzino & Pardo [12] tackle monadic effects. It would be interesting to extend our approach to finding uses of their effectful recursion schemes.

7.2 Automation

Higher-order matching is a general technique for matching expressions in functional programs against expression templates. In the context of Haskell, Sittampalam and de Moor [18] have applied higher-order matching in their rewriting system, called MAG. They have used MAG for fusion in the following way. the programmer specifies the initial program, a specification of the target program and suitable rewrite rules. The latter includes a rule for $foldr$ fusion:

$$f\ (foldr\ c\ n\ l) = foldr\ c'\ n'\ l$$
$$\textbf{if}\ f\ n = n'$$
$$\forall x\ y.f\ (c\ x\ y) = c'\ x\ (f\ y)$$

Then the MAG system will attempt to derive the target implementation by applying the rewrite rules. Finally, the programmer needs to check whether MAG has only applied the fusion rule to strict functions f, a side condition of the fusion rule that cannot be specified in MAG.

GHC rewrite rules [11] are a lightweight way to (semi-)automate fusion. The programmer provides rewrite rules to rewrite program patterns into their fused forms and the compiler applies these whenever it finds an opportunity. The one part that is not automated is that programmers still have to write their code in terms of the higher-order recursion schemes. GHC has set up various of its base libraries in this way to benefit from fold/build fusion among others.

Building on GHC rewrite rules, Coutts et al. [3] have proposed stream fusion as a convenient alternative to foldr/build fusion. Stream fusion is able to fuse zips and left folds, but, on the downside, it is less obvious for the programmer to write his functions in the required style. Hinze et al. [8] provide clues for how to generalize stream fusion: by expressing functions in terms of an unfold, followed by a natural transformation and a fold.

The *hlint* [15] tool is designed to recognize various code patterns and offer suggestions for improving them. In particular, it recognizes various forms of explicit recursion and suggests the use of appropriate higher-order functions like map, $filter$ and $foldr$ that capture these recursion patterns. As we already showed in Section 6.2, we are able to detect more instances of folds for Haskell lists than hlint. Moreover, hlint makes no effort to detect folds for other algebraic datatypes.

Supercompilation [21] is a much more generic and brute-force technique for program specialization that is capable of fusing producer-consumer pipelines. Unfortunately, the current state of the art of supercompilation for Haskell [2] is still too unreliable to be used in practice.

8. Discussion

We have presented a syntactic approach to transforming explicitly recursive functions into invocations of higher-order recursion schemes (folds and builds in particular). Our experimental evaluation shows that this technique is effective at finding many such occurrences in popular Haskell packages written by experienced programmers.

Our work currently targets the most common case: directly recursive functions over directly recursive regular datatypes. In fu-

ture work it would be interesting to extend this class to encompass a wider range, although we expect diminishing returns.

Mutually Recursive Functions Our approach does not deal with mutually recursive functions like:

```
concat :: [[a]] → [a]
concat []       = []
concat (x : xs) = concat' x xs
  where
    concat' :: [a] → [[a]] → [a]
    concat' [] xs = concat xs
    concat' (y : ys) xs = y : concat' ys xs
```

which is ideally rewritten to:

$$concat\ l = build\ (\lambda c\ n \to foldr\ (\lambda x\ xs \to foldr\ c\ xs\ x)\ n\ l)$$

but the mutual recursion obscures the *build*, and instead we get:

$$concat\ l = foldr\ (\lambda x\ xs \to foldr\ (:)\ xs\ x)\ []\ l$$

In order to solve this problem, the *build* finding algorithm should be extended to handle mutually recursive groups.

Mutually Recursive Datatypes Mutually recursive functions arise naturally for mutually recursive datatypes, which are common representations for abstract syntax trees and document structures. A simple example are rose trees:

```
data Rose = Node Int Forest
data Forest = Nil | Cons Rose Forest

sizeR :: Rose → Int
sizeR (Node _ f) = 1 + sizeF f

sizeF :: Forest → Int
sizeF Nil        = 0
sizeF (Cons r f) = sizeR r + sizeF f
```

can be written as

```
data Rose = Node Int Forest
data Forest = Nil | Cons Rose Forest

sizeR = foldR (λx f → 1 + f) 0 (λr f → r + f)
sizeF = foldF (λx f → 1 + f) 0 (λr f → r + f)
```

where the two mutually recursive datatypes have the following signatures for their fold functions:

```
foldR :: (Int → f → r) → f → (r → f → f) → Rose   → r
foldF :: (Int → f → r) → f → (r → f → f) → Forest → f
```

GADTs GADTs pose an additional challenge because they represented a family of types whose members are selected by means of one or more type indices. For instance, the GADT *Expr a* represents a family of expression types by means of the type index a.

```
data Expr a where
  Lit :: Int → Expr Int
  Add :: Exp Int → Exp Int → Exp Int
  Eq :: Exp Int → Exp Int → Exp Bool
```

This type indexing is carried over to the type signatures of its fold and build definitions [10], where $Expr :: * \to *$ is abstracted to $r :: * \to *$.

```
foldE :: (Int → r Int)
      → (r Int → r Int → r Int)
      → (r Bool → r Bool → r Bool)
      → Exp a → r a

buildExp :: (∀r . (Int → r Int)
             → (r Int → r Int → r Int)
             → (r Bool → r Bool → r Bool)
             → r a) → Exp a
```

This means that when we rewrite a catamorphism like

```
eval :: Expr a → a
eval (Lit n)   = n
eval (Add x y) = eval x + eval y
eval (Eq x y)  = eval x ≡ eval y
```

into an invocation of $foldE$, we must come up with an appropriate indexed type to instantiate r. In particular, when the catamorphism does not feature such an indexed type, we may have to introduce a new one:

```
newtype R a = R {runR :: a}
eval e =
  runR $ foldE (λn → R n)
               (λx y → R (runR x + runID y))
               (λx y → R (runR x ≡ runR y))
               e
```

References

[1] R. Bird. Constructive functional programming. In M. Broy, editor, *Marktoberdorf International Summer school on Constructive Methods in Computer Science*, NATO Advanced Science Institute Series. Springer Verlag, 1989.

[2] M. C. Bolingbroke. Call-by-need supercompilation. Technical Report UCAM-CL-TR-835, University of Cambridge, Computer Laboratory, May 2013. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-835.pdf`.

[3] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP07*, 2007.

[4] G. A. Delbianco, M. Jaskelioff, and A. Pardo. Applicative shortcut fusion. In R. Peña and R. L. Page, editors, *Trends in Functional Programming*, pages 179–194. Springer, 2011.

[5] N. Ghani and P. Johann. Short cut fusion of recursive programs with computational effects. In P. Achten, P. Koopman, and M. Morazan, editors, *Trends in Functional Programming*, volume 9 of *Trends in Functional Programming*, pages 113–128. Intellect, 2009.

[6] J. Gibbons. Origami programming. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave, 2003. ISBN 1-4039-0772-2.

[7] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[8] R. Hinze, T. Harper, and D. W. James. Theory and practice of fusion. In J. Hage and M. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 19–37. Springer Berlin / Heidelberg, 2011.

[9] R. Hinze, N. Wu, and J. Gibbons. Unifying recursion schemes. In *International Conference on Functional Programming*, March 2013. URL `http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/urs.pdf`. Submitted for publication.

[10] P. Johann and N. Ghani. Foundations for structured programming with gadts. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 297–308, New York, NY, USA, 2008. ACM.

[11] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[12] C. Manzino and A. Pardo. Shortcut fusion of monadic programs. *Journal of Universal Computer Science*, 14(21):3431–3446, 2008.

[13] L. Meertens. Algorithmics – towards programming as a mathematical activity. In *CWI symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[14] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. pages 124–144. Springer-Verlag, 1991.

[15] N. Mitchell. *HLint Manual*, 2006. URL `http://community.haskell.org/~ndm/darcs/hlint/hlint.htm`.

[16] B. O'Sullivan. Criterion, a new benchmarking library for haskell, 2009. URL `http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/`.

[17] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002. ISSN 0362-1340. doi: 10.1145/636517.636528. URL `http://doi.acm.org/10.1145/636517.636528`.

[18] G. Sittampalam and O. de Moor. Mechanising fusion. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 79–129. Palgrave, 2003. ISBN 1-4039-0772-2.

[19] The GHC Team. The ghc wiki: Annotations, 2011. URL `http://hackage.haskell.org/trac/ghc/wiki/Annotations`.

[20] The GHC Team. Extending and using ghc as a library: Compiler plugins, 2011. URL `http://hackage.haskell.org/trac/ghc/wiki/NewPlugins`.

[21] V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[22] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990. ISSN 0304-3975.

A. Pipeline Benchmarks

```
  -- List Pipelines
suml :: [Int] → Int
suml []       = 0
suml (x : xs) = x + suml xs

mapl :: (a → b) → [a] → [b]
mapl f = go
  where
    go []       = []
    go (x : xs) = f x : go xs

uptol :: Int → Int → [Int]
uptol lo up = go lo
  where
    go i
      | i > up      = []
      | otherwise   = i : uptol (i + 1) up

l1, l2, l3, l4, l5 :: Int → Int
l1 n = suml (1 `uptol` n)
l2 n = suml (mapl (+1) (1 `uptol` n))
l3 n = suml (mapl (+1) (mapl (+1) (1 `uptol` n)))
l4 n = ...
l5 n = ...

  -- Tree Pipelines
sumt :: Tree Int → Int
sumt (Leaf x)     = x
sumt (Branch l r) = sumt l + sumt r

mapt :: (a → b) → Tree a → Tree b
mapt f = go
  where
    go (Leaf x)     = Leaf (f x)
    go (Branch l r) = Branch (go l) (go r)

uptot :: Int → Int → Tree Int
uptot lo hi
    | lo ⩾ hi    = Leaf lo
    | otherwise =
      let mid = div (lo + hi) 2
      in Branch (uptot lo mid) (uptot (mid + 1) hi)

t1, t2, t3, t4, t5 :: Int → Int
t1 n = sumt (1 `uptot` n)
t2 n = sumt (mapt (+1) (1 `uptot` n))
t3 n = sumt (mapt (+1) (mapt (+1) (1 `uptot` n)))
t4 n = ...
t5 n = ...
```