Shortcut Fusion Rules for the Derivation of Circular and Higher-order Monadic Programs

Alberto Pardo
Universidad de la Republica, Uruguay
pardo@fing.edu.uy

João Paulo Fernandes *
Universidade do Minho & Universidade Atlântica, Portugal
jpaulo@di.uminho.pt

João Saraiva †
Universidade do Minho, Portugal
jas@di.uminho.pt

Abstract

Functional programs often combine separate parts using intermediate data structures for communicating results. These programs are modular, easier to understand and maintain, but suffer from inefficiencies due to the generation of those gluing data structures. To eliminate such redundant data structures, some program transformation techniques have been proposed. One such technique is shortcut fusion, and has been studied in the context of both pure and monadic functional programs.

Recently, we have extended standard shortcut fusion: in addition to intermediate structures, the program parts may now communicate context information, and it still is possible to eliminate those structures. This is achieved by transforming the original function composition into a circular program. This new technique, however, has been studied in the context of purely functional programs only. In this paper, we propose an extension to this new form of fusion, but in the context of monadic programming: we derive monadic circular programs from strict ones, maintaining the global effects. Later, the circularities in the derived programs are traded by higher-order definitions, using a well-known program transformation technique. We finally obtain very efficient deforested programs.

An important feature of our extensions is that they can be uniformly defined for a wide class of data types and monads, using generic calculation rules.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Compilers, Optimization; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructors - Program and Recursion Schemes

General Terms Languages, Theory, Algorithms

Keywords Circular Programming, Monadic Programming, Program Calculation, Shortcut Fusion, Deforestation

* Supported by Fundação para a Ciência e Tecnologia (FCT), grant No. SFRH/BD/19186/2004

† Supported by Fundação para a Ciência e Tecnologia (FCT), grant No. SFRH/BSAB/782/2008

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

1. Introduction

Functional programs often combine separate parts of the program using intermediate structures for communicating results. In general, we have programs such as $prog = cons \circ prod$, where $prod$ is called the producer function and $cons$ is called the consumer function. Programs so defined are modular and have many benefits, such as clarity and maintainability, but suffer from inefficiencies caused by the generation of the intermediate data structures that glue functions $cons$ and $prod$ together.

In response to this problematic, some program transformation techniques have been studied aiming at the elimination of intermediate data structures. One of these techniques is known as shortcut fusion, or shortcut deforestation [14]. This technique eliminates the generation of the intermediate structure, of type b, when $prod :: a \to b$, $cons :: b \to c$ and $prog = cons \circ prod$. Shortcut fusion has recently been studied and applied also in the context of monadic functional programs [11, 18].

In [9], we have proposed circular programs as an extension to *standard* shortcut fusion. Circular programs were first studied by Bird [4] as an elegant and efficient technique to eliminate multiple traversals of data structures. As the name suggests, circular programs are characterized by having what appears to be a circular definition: arguments in a function call depend on results of that same call. That is, they contain definitions of the form:

$$(..., x, ...) = f\ (..., x, ...)$$

Circular programs have also been studied as an optimization technique to deforest in accumulating parameters [24]. Furthermore, circular programs are a powerful, elegant and concise technique to express multiple traversal algorithms and attribute grammars [10].

In [9], we have shown how circular programs can be used to achieve intermediate structure deforestation in programs such as $prog = cons \circ prod$, but where $prod :: a \to (b, z)$ and $cons :: (b, z) \to c$. This means that the producer function may generate, besides the intermediate structure b, an additional value, of type z, that the consumer function may need to compute its result. Later, a calculation rule is applied to $prog$, which is transformed into an equivalent circular program that does not construct any intermediate structure and that traverses the input data (of type a) only once. The rule applied to $prog$ is generic in the sense that it can be applied to a wide range of programs and datatypes. However, it does not handle monadic functional programs, that is, programs that, for example, rely on a global state or perform I/O operations. Thus, the rule has a limited applicability scope since several programs, like compilers, pretty-printers or parsers do rely on global effects.

Our motivation for the present work is to extend shortcut fusion to the kind of programs we studied in [9], but in the context of monadic programming. The goal is to achieve fusion of monadic

programs, maintaining the global effects. We study two cases: the case where the producer function is monadic and the consumer is given by a pure function, and the case where both functions are monadic. For both cases, fusion is achieved by transforming the original program into a circular one. We do not consider the case where the producer is given by a pure function (and the consumer is given by a monadic one) since it can already be fused using the shortcut fusion rule we presented in [9].

An alternative solution to achieve intermediate structure deforestation for programs such as *prog* is to transform them into higher-order programs using a well-known program transformation technique called lambda abstraction [20]. In our particular context, the idea of the transformation is to derive a new function $prog' :: a \to (z \to c, z)$, which returns a function and the same value of type z that would be generated by *prod*, such that $prog\ a = f\ z$ **where** $(f, z) = prog'\ a$. Obtaining such higher-order programs is interesting since its execution is not restricted to a lazy evaluation setting as it happens with the execution of the circular ones. Based on this idea, Voigtländer [25] introduced a shortcut fusion rule for the derivation of pure, higher-order programs from compositions like *prog* when lists are the intermediate structure. In this paper, we extend this result in two ways. First, we present a generic formulation of the shortcut fusion rule for the derivation of pure higher-order programs that can be applied to a wide range of datatypes as intermediate structure. Second, we extend the generic rule to the context of monadic programming, obtaining shortcut fusion rules for the derivation of monadic higher-order programs.

Experimental benchmarks we have conducted in [10] show that the performance of the higher-order programs derived from programs like *prog* is significantly better (up to 3 times) than the performance of their circular or original equivalents, both in terms of time and memory consumption, whereas the performance between circular programs and the ones from which they are derived is essentially the same.

Although the higher-order programs we finally derive in this paper are the ones with the best running performances, the calculation of circular programs is of great importance to our future work, and has strong relations with some of our previous studies [10]. Indeed, there is a close relationship between circular programs and Attribute Grammars (AGs): circular programs are the natural representation of AGs in a lazy setting. In the past [10], we have applied well-known AG techniques to circular programs, in order to derive efficient equivalent implementations. The correctness of such techniques, however, has not been formally proved yet, although they are widely used by the AG community. In order to be able to prove their correctness, we, therefore, would like to be able to express such techniques in calculational form. The calculation of circular programs, in the way presented in this paper and in a previous work [9], also brings us closer to that goal.

Throughout we will use Haskell notation, assuming a cpo semantics (in terms of pointed cpos), but without the presence of the *seq* function [17].

This paper is organized as follows. Sections 2 and 3 present two motivating examples that serve to illustrate the applicability of our techniques. The generic constructions that give rise to the specific program schemes and laws presented in those examples are developed in Sections 4 and 5. Finally, in Section 6 we draw some conclusions and describe directions for future work.

2. Bit String Transformation

To illustrate our technique we first consider an example based on a simple bit string conversion that has applications in cryptography [15]. Suppose we want to transform a sequence of bits into a new one, of the same length, by applying the exclusive or between each bit and the binary sum (sum modulo 2) of the sequence. We will consider that the input sequence is given as a string of bits, which will be parsed into a list and then transformed. It is in the parsing phase that computational effects will come into play, as we will use a monadic parser.

Suppose we are given the string `"101110110001"`. To transform this string of bits, we start by parsing it, computing as result a list of bits $[1,0,1,1,1,0,1,1,0,0,0,1]$, and its binary sum (1 in this case). Having the list and the binary sum, the original sequence is transformed into this one $[0,1,0,0,0,1,0,0,1,1,1,0]$ after applying the exclusive or of each bit with 1 (the binary sum).

To construct the parser, we adopt a usual definition of parser monad (see [16] for more details):

$$\textbf{newtype}\ Parser\ a = P\ (String \to [(a, String)])$$

$$\begin{aligned}&\textbf{instance}\ Monad\ Parser\ \textbf{where}\\ &\quad return\ a = P\ (\lambda cs \to [(a, cs)])\\ &\quad p \gg\!= f\ \ = P\ (\lambda cs \to concat\ [parse\ (f\ a)\ cs' \mid\\ &\qquad\qquad\qquad\qquad (a, cs') \leftarrow parse\ p\ cs])\end{aligned}$$

$$\begin{aligned}&parse :: Parser\ a \to String \to [(a, String)]\\ &parse\ (P\ p) = p\end{aligned}$$

$$\begin{aligned}&(\langle|\rangle) :: Parser\ a \to Parser\ a \to Parser\ a\\ &(P\ p)\ \langle|\rangle\ (P\ q) = P\ (\lambda cs \to \textbf{case}\ p\ cs +\!\!+ q\ cs\ \textbf{of}\\ &\qquad\qquad\qquad\qquad\qquad\qquad [\,] \quad\ \to [\,]\\ &\qquad\qquad\qquad\qquad\qquad\qquad (x : xs) \to [x])\end{aligned}$$

$$\begin{aligned}&pzero :: Parser\ a\\ &pzero = P\ (\lambda cs \to [\,])\end{aligned}$$

$$\begin{aligned}&item :: Parser\ Char\\ &item = P\ (\lambda cs \to \textbf{case}\ cs\ \textbf{of}\\ &\qquad\qquad\qquad\qquad [\,] \quad\ \to [\,]\\ &\qquad\qquad\qquad\qquad (c : cs) \to [(c, cs)])\end{aligned}$$

Alternatives are represented by a deterministic choice operator ($\langle|\rangle$), which returns at most one result. The parser *pzero* is a parser that always fails. The *item* parser returns the first character in the input string.

We can use these simple parser combinators to define parsers for bits and bit strings. The binary sum is calculated as the exclusive or of the bits of the parsed sequence. We write $\oplus$ to denote exclusive or over the type *Bit*.

$$\textbf{data}\ Bit = Z \mid O$$

$$\begin{aligned}&bit :: Parser\ Bit\\ &bit = \textbf{do}\ c \leftarrow item\\ &\qquad\quad \textbf{case}\ c\ \textbf{of}\ \texttt{'0'} \to return\ Z\\ &\qquad\qquad\qquad\quad \texttt{'1'} \to return\ O\\ &\qquad\qquad\qquad\quad \_\ \ \to pzero\end{aligned}$$

$$\begin{aligned}&bitstring :: Parser\ ([Bit], Bit)\\ &bitstring = \textbf{do}\ b \qquad\ \leftarrow bit\\ &\qquad\qquad\quad (bs, s) \leftarrow bitstring\\ &\qquad\qquad\quad return\ (b : bs, b \oplus s)\\ &\qquad\qquad \langle|\rangle\ return\ ([\,], Z)\end{aligned}$$

Now, we implement the transformation function:

$$\begin{aligned}&transform :: ([Bit], Bit) \to [Bit]\\ &transform\ ([\,], \_) \quad\ = [\,]\\ &transform\ (b : bs, s) = (b \oplus s) : transform\ (bs, s)\end{aligned}$$

In summary, the transformation consists of:

```
shift :: Parser [Bit]
shift = do (bs, s) ← bitstring
           return (transform (bs, s))
```

We may notice that the above solution constructs an intermediate list of bits that we would like to eliminate with fusion. The fusion law to be used is a law in the style of shortcut fusion, similar to that conceived for the derivation of purely functional circular programs [9], but with the difference that now it deals with monadic functions. Below, we present the specific instance for lists (which is the type of the intermediate structure), and in Section 4 we show that both the law and the programs schemes respond to generic definitions that can be formulated for several datatypes.

Like in standard shortcut fusion [14], our law assumes that the producer and the consumer (*bitstring* and *transform* in this case) are expressed in terms of certain program schemes. In standard shortcut fusion the consumer is required to be given by a structural recursive definition in terms of a recursion scheme called fold (usually called *foldr* in the case of lists [2]). In our law, we also require the consumer to be given by a structural recursive definition, but in terms of a variation of fold, called *pfold*, which admits as input an additional constant parameter to be used along the recursive calls:

```
pfold_L :: (z → b, a → b → z → b) → ([a], z) → b
pfold_L (hnil, hcons) = p_L
  where p_L ([], z)     = hnil z
        p_L (a : as, z) = hcons a (p_L (as, z)) z
```

Like in standard shortcut fusion, we require the producer to be able to show that the list constructors can be abstracted from the process that generates the intermediate list. The difference with the standard case is that we consider producers that generate the intermediate list as part of a pair which in turn is the result of monadic computation. This is expressed by a function called $mbuildp_L$:

```
mbuildp_L :: Monad m
  ⇒ (∀ b . (b, a → b → b) → m (b, z)) → m ([a], z)
mbuildp_L g = g ([], (:))
```

Having stated the forms required to the producer and the consumer it is now possible to formulate the law.

LAW 2.1 (PFOLD/MBUILDP FOR LISTS). *Let m be a recursive monad.*

```
  do (xs, z) ← mbuildp_L g
     return (pfold_L (hnil, hcons) (xs, z))
=
  mdo (v, z) ← let knil       = hnil z
                   kcons x y = hcons x y z
               in g (knil, kcons)
     return v
```

This law transforms a monadic composition, where the producer is an effectful function but may not necessarily the consumer be, into a single monadic function with a circular argument z. Indeed, z is a value computed by $g\ (knil, kcons)$ but in turn used by $knil$ and $kcons$. An interesting feature of this law is the fact that the introduction of the circularity needs the use of a recursive binding within a monadic computation, and therefore require the monad to be *recursive* [8]. A recursive **do** (**mdo**-notation) is supported by Haskell for those monads that are declared an instance of the $MonadFix$ class.

```
class Monad m ⇒ MonadFix m where
  mfix :: (a → m a) → m a
```

The parsing monad presented above can be declared an instance of this class:

```
instance MonadFix Parser where
  mfix f = P (λcs → mfixL (λ(x, y) → parse (f x) cs))
    where
      mfixL f = case fix (f ∘ head) of
                  []      → []
                  (x : _) → x : mfixL (tail ∘ f)
```

To see the law in action, we write *transform* and *bitstring* in terms of $pfold_L$ and $mbuildp_L$, respectively:

```
transform = pfold_L (hnil, hcons)
  where hnil _       = []
        hcons b r s = (b ⊕ s) : r

bitstring = mbuildp_L g
  where g (nil, cons) = do b       ← bit
                           (bs, s) ← g (nil, cons)
                           return (cons b bs, b ⊕ s)
                        ⟨|⟩ return (nil, Z)
```

Then, by applying Law 2.1 we obtain:

```
shift = mdo (bs, s) ← g ([], λb r → (b ⊕ s) : r)
            return bs
```

Inlining, we get the following circular monadic program:

```
shift = mdo (bs, s) ← let gk = do b       ← bit
                                  (bs', s') ← gk
                                  return ((b ⊕ s) : bs',
                                          b ⊕ s')
                              ⟨|⟩ return ([], Z)
                      in gk
            return bs
```

The above program avoids the construction of the intermediate list of bits, by introducing a circular definition. Indeed, we may notice that s (the modulo 2 sum of an input sequence of bits) is used (in $b \oplus s$) in the function call to gk. However, s is also a result of that same call, hence the circularity.

3. A Simple Programming Environment

Let us now consider that we wish to construct a program to deal with the scope rules of a simple programming language[1].

A program in our language, such as the one presented below, consists in a sequence of instructions, where each instruction may either be the declaration or the use of a variable.

$$[\,\mathbf{use}\ x; \mathbf{decl}\ x; \mathbf{decl}\ x; \mathbf{use}\ y;\,]$$

In order to be well formed, programs in our language must follow these scope rules:

1. all used variables must be declared. The declaration of a variable, however, may occur after its first use;

2. a variable must be declared at most once.

We aim to develop a semantic function that analyzes a sequence of instructions and computes a list containing the variable identifiers of the instructions which do not obey to the rules of the language. When such an instruction is found, we also want to output

[1] Due to space limitations, we consider a simplified version of the Algol 68 rules only. The complete definition may be found in [7], and is used as a generic library for the name analysis task of the Eli system [27].

an error message explaining the programming error encountered. In order to make the problem more interesting, and also to make it easier to detect which identifiers are incorrect, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is $[x, y]$: variable x has been declared twice, and the use of variable y has no binding occurrence at all. As a side-effect of computing this list, the following error messages must be displayed:

Duplicate: decl x
Missing: decl y

Because we allow an *use-before-declare* discipline, a conventional implementation of the required analysis naturally leads to a program which traverses the abstract syntax tree twice: once for accumulating the declarations of identifiers and constructing the environment, and once for checking the uses of identifiers, according to the computed environment.

The uniqueness of names is detected in the first traversal: for each newly encountered declaration it is checked whether that identifier has already been declared. In this case an error is computed. The second traversal receives the global environment of the program and detects the variables that are used without being declared.

In a straightforward implementation of the semantic function, this strategy has two important consequences: the first is that a *"gluing"* data structure has to be constructed to explicitly pass the detected errors from the first to the second traversal in order to compute the final list of errors in the desired order; the second is that, in order to be able to compute the missing declarations of a program, it is necessary to pass from the first to the second traversal the names of the variables that are used in the program.

The abstract syntax of the language may be described by the following data type definitions:

$$\mathbf{type}\ Prog = [It] \qquad \mathbf{data}\ It = Decl\ String \mid Use\ \ String$$

In order to pass the necessary information from the first to the second traversal of a program, we define:

$$\mathbf{type}\ Prog_2 = [It_2] \quad \mathbf{data}\ It_2 = Dupl_2\ String \mid Use_2\ \ String$$

Errors resulting from duplicate declarations are passed using constructor $Dupl_2$.

According to the strategy defined earlier, we compute the semantic errors that occur in a program as follows:

$$\begin{aligned}&semantics :: Prog \to IO\ [String]\\ &semantics\ p = \mathbf{do}\ (p', env) \leftarrow duplicate\ [\,]\ p\\ &\qquad\qquad\qquad missing\ (p', env)\end{aligned}$$

The function *duplicate* detects duplicate variable declarations by collecting all the declarations occurring in a program. It is a monadic function since it needs to output error messages resulting from the errors it detects. The definition of such function is as follows[2]:

$$\begin{aligned}&duplicate :: [String] \to Prog \to IO\ (Prog_2, [String])\\ &duplicate\ ds\ [\,] = return\ ([\,], ds)\\ &duplicate\ ds\ (Decl\ v : p)\\ &\quad = \mathbf{do}\ (p', ds') \leftarrow duplicate\ (v : ds)\ p\\ &\qquad\ \mathbf{if}\ (v \in ds)\\ &\qquad\quad \mathbf{then\ do}\ put\ (\texttt{"Duplicate: decl"} +\!\!+ v)\\ &\qquad\qquad\qquad return\ (Dupl_2\ v : p', ds')\\ &\qquad\quad \mathbf{else}\ \ return\ (p', ds')\\ &duplicate\ ds\ (Use\ v : p)\\ &\quad = \mathbf{do}\ (p', ds') \leftarrow duplicate\ ds\ p\\ &\qquad\ return\ (Use_2\ v : p', ds')\end{aligned}$$

Besides detecting the invalid declarations, function *duplicate* also computes a data structure, of type $Prog_2$, that is later traversed in order to detect variables that are used without being declared. This detection is performed by function *missing*, which is monadic as it also outputs error messages:

$$\begin{aligned}&missing :: (Prog_2, [String]) \to IO\ [String]\\ &missing\ ([\,], \_) = return\ [\,]\\ &missing\ (Dupl_2\ v : p, env)\\ &\quad = \mathbf{do}\ r \leftarrow missing\ (p, env)\\ &\qquad\ return\ (v : r)\\ &missing\ (Use_2\ v : p, env)\\ &\quad = \mathbf{do}\ r \leftarrow missing\ (p, env)\\ &\qquad\ \mathbf{if}\ (v \in env)\\ &\qquad\quad \mathbf{then}\ return\ r\\ &\qquad\quad \mathbf{else\ do}\ put\ (\texttt{"Missing: decl"} +\!\!+ v)\\ &\qquad\qquad\qquad return\ (v : r)\end{aligned}$$

We would like to eliminate the intermediate structure of type $Prog_2$ generated by *duplicate*. If we attempted to directly apply Law 2.1 for that aim, then we would see that in this case the result of the law is a function that returns a monadic computation which in turn yields a monadic computation (and not a value) as result, that is, something of type $m\ (m\ a)$, for some a. This is because the consumer is also monadic. To obtain a value and not a computation as final result, it is simply necessary to run the computation. This gives the following shortcut fusion law, which requires the same schemes for consumer and producer as Law 2.1 but is able to fuse effectful functions.

LAW 3.1 (EFFECTFULL PFOLD/MBUILDP FOR LISTS). *Let m be a recursive monad.*

$$\begin{aligned}&\mathbf{do}\ (xs, z) \leftarrow mbuildp_L\ g\ c\\ &\quad pfold_L\ (hnil, hcons)\ (xs, z)\\ =\\ &\mathbf{mdo}\ (m, z) \leftarrow \mathbf{let}\ knil = hnil\ z\\ &\qquad\qquad\qquad\qquad\ kcons\ x\ y = hcons\ x\ y\ z\\ &\qquad\qquad\qquad\ \mathbf{in}\ \ g\ (knil, kcons)\ c\\ &\qquad m\end{aligned}$$

Observe that, in this case, $hnil :: z \to m\ \ b$ and $hcons :: a \to m\ \ b \to z \to m\ \ b$, for some monad m, and therefore $pfold_L\ (hnil, hcons) :: ([a], z) \to m\ b$. Also, notice that,

$$\begin{aligned}&mbuildp_L :: Monad\ m\\ &\quad \Rightarrow (\forall\ b\ .\ (b, a \to b \to b) \to c \to m\ (b, z))\\ &\qquad \to c \to m\ ([a], z)\end{aligned}$$

that is, $mbuildp_L\ g$ is a function of type $c \to m\ ([a], z)$. It is in this way that it will be considered in Section 4 when we will define the generic formulation of the laws. However, in Section 2 it was defined as a value of type $m\ ([a], z)$ because that form is more appropriate for writing monadic parsers.

For the present example we do not need to provide the instance of the *MonadFix* class for the *IO* monad as it is automatically provided by the Glasgow Haskell Compiler (GHC), which is the reference compiler we are using.

[2] We abbreviate $putStrLn$ as put.

Now, if we write *missing* and *duplicate* in terms of $pfold_L$ and $mbuildp_L$, respectively,

```
missing = pfold_L (hnil, hcons)
  where
    hnil _ = return []
    hcons (Dupl_2 v) mr env
      = do r ← mr
           return (v : r)
    hcons (Use_2 v) mr env
      = do r ← mr
           if (v ∈ env)
             then return r
             else do put ("Missing:decl" ++ v)
                     return (v : r)

duplicate ds p = mbuildp_L (g ds) p
  where
    g ds (nil, cons) [] = return (nil, ds)
    g ds (nil, cons) (Decl v : p)
      = do (p', ds') ← g (v : ds) (nil, cons) p
           if (v ∈ ds)
             then do put ("Duplicate:decl" ++ v)
                     return (cons (Dupl_2 v) p', ds')
             else return (p', ds')
    g ds (nil, cons) (Use v : p)
      = do (p', ds') ← g ds (nil, cons) p
           return (cons (Use_2 v) p', ds')
```

we can apply Law 3.1 to *semantics* obtaining a deforested circular definition, which, when inlined, gives the following:

```
semantics p
 = mdo{
   (m, env) ← let
     gk ds [] = return (return [], ds)
     gk ds (Decl v : p)
       = do (p', ds') ← gk (v : ds) p
            if (v ∈ ds)
            then do put ("Duplicate:decl" ++ v)
                    return (do r ← p'
                               return (v : r)
                           , ds')
            else return (p', ds')
     gk ds (Use v : p)
       = do (p', ds') ← gk ds p
            return (do r ← p'
                       if (v ∈ env)
                       then return r
                       else do put ("Missing:" ++ v)
                               return (v : r)
                   , ds')
     in gk [] p;
   m; }
```

Regarding the above program we may notice that it does not construct the intermediate $Prog_2$ structure that was used in the original *semantics* program to glue the functions *duplicate* and *missing* together. The deforestation of such structure was achieved introducing a circular definition on the global environment of an input program (i.e., on the list of all the variables declared in such program). Indeed, *env* is used, in the call to *gk*, whenever an instruction *Use v* is found in an input program, to check whether *v* is an element of *env* or not. However, *env* is also a result of the same *gk* call, which means that *env* is used at the same time it is being constructed, hence the circular definition.

4. Calculating circular programs, generically

In this section, we show that the definition of the program schemes *pfold* and *mbuildp*, and the *pfold/mbuildp* laws, presented for lists in the previous sections, are instances of generic definitions valid for a wide class of datatypes.

We write $\pi_1 :: (a, b) \to a$ and $\pi_2 :: (a, b) \to b$ to denote the product projections. Let $(f \times g)\ (x, y) = (f\ x, g\ y)$.

4.1 Data types

Generically, the structure of datatypes can be captured using the concept of a *functor*. A functor consists of a type constructor F and a function $map_F :: (a \to b) \to (F\ a \to F\ b)$, which preserves identities and compositions: $map_F\ id = id$ and $map_F\ (f \circ g) = map_F\ f \circ map_F\ g$. A standard example of a functor is that formed by the list type constructor and the well-known *map* function.

Semantically speaking, recursive datatypes correspond to least fixed points of functors. Given a datatype declaration it is possible to derive a functor F such that the datatype is the least solution to the equation $\tau \cong F\tau$. It is usual to write μF to denote the type corresponding to that solution. The isomorphism between μF and $F\ \mu F$ is provided by two strict functions, called $in_F :: F\ \mu F \to \mu F$ and $out_F :: \mu F \to F\ \mu F$, inverses of each other, and that pack the constructors and destructors of the datatype, respectively (see [1, 12] for details).

Let us consider, for example, a datatype for simple arithmetic expressions:

```
data Exp = Num Int | Add Exp Exp
```

Corresponding to this datatype we can derive a functor E such that:

```
data E a = FNum Int | FAdd a a

map_E :: (a → b) → E a → E b
map_E f (FNum n)    = FNum n
map_E f (FAdd a a') = FAdd (f a) (f a')
```

In this case, $\mu E \cong Exp$ and

```
in_E :: E μE → μE
in_E (FNum n)    = Num n
in_E (FAdd e e') = Add e e'

out_E :: μE → E μE
out_E (Num n)    = FNum n
out_E (Add e e') = FAdd e e'
```

In the case of lists, the structure is captured by a bifunctor L (a functor on two variables) because of the presence of the type parameter. That is, $\mu(L\ a) \cong [a]$.

```
data L a b = FNil | FCons a b

map_La :: (b → c) → L a b → L a c
map_La f FNil          = FNil
map_La f (FCons a b) = FCons a (f b)
```

4.2 Fold

Given a functor F (signature of a datatype) and a function $h :: F\ a \to a$, *fold* (or *catamorphism*) [3, 12] is defined as the least function such that

$$fold :: (F\ a \to a) \to \mu F \to a$$
$$fold\ h \circ in_F = h \circ map_F\ (fold\ h)$$

A function $h :: F\ a \to a$ is called an *F-algebra*. If functor F is given by n constructors,

$$\mathbf{data}\ F\ a = FC_1\ \sigma_{1,1} \cdots \sigma_{1,r_1}\ \mid \cdots \mid\ FC_n\ \sigma_{n,1} \cdots \sigma_{n,r_n}$$

then an algebra $h :: F\ a \to a$ has n component functions $(h_1, \ldots, h_n)$, each with type $h_i :: \sigma_{i,1} \to \cdots \to \sigma_{i,r_i} \to a$. For example, an algebra corresponding to the functor E is a function $h :: E\ a \to a$ of the form:

$$h\ (FNum\ n) = num\ n$$
$$h\ (FAdd\ a\ a') = add\ a\ a'$$

with component functions $num :: Int \to a$ and $add :: a \to a \to a$. In the specific instance of fold for this datatype,

$$fold_E :: (Int \to a, a \to a \to a) \to Exp \to a$$
$$fold_E\ (num, add)\ (Num\ a) = num\ a$$
$$fold_E\ (num, add)\ (Add\ e\ e') = add\ (fold_E\ (num, add)\ e)\ (fold_E\ (num, add)\ e')$$

we write an E-algebra simply as a pair (num, add).

For the list datatype we can do something similar, we can write an algebra $h :: L\ a\ b \to b$ as a pair $(nil, cons)$. Then,

$$fold_L :: (b, a \to b \to b) \to [a] \to b$$
$$fold_L\ (nil, cons)\ [\,] = nil$$
$$fold_L\ (nil, cons)\ (a : as) = cons\ a\ (fold_L\ (nil, cons)\ as)$$

The same can be done with algebras associated to other datatypes.

Fold enjoys many algebraic laws that are useful for program transformation. A well-known example is *shortcut fusion* [14, 13, 23] (also known as the *fold/build* rule), a law that is derived from a free theorem [26].

LAW 4.1 (FOLD/BUILD). *For strict h,*

$$fold\ h \circ build\ g = g\ h$$

where

$$build :: (\forall\ a\ .\ (F\ a \to a) \to c \to a) \to c \to \mu F$$
$$build\ g = g\ in_F$$

The standard example is the case for lists:

LAW 4.2 (FOLD/BUILD FOR LISTS).

$$fold_L\ (nil, cons) \circ build_L\ g = g\ (nil, cons)$$

where

$$build_L :: (\forall\ b\ .\ (b, a \to b \to b) \to c \to b) \to c \to [a]$$
$$build_L\ g = g\ ([\,], (:))$$

4.3 Fold with parameters

Some recursive functions use context information in the form of constant parameters for their computation. We are interested in functions of the form $f :: (\mu F, z) \to a$, defined by structural recursion on μF and with context information of type z. Our method will assume that consumers are functions of this kind.

Such functions can be defined in different ways. One alternative consists of fixing the value of the context information; this can be written as a fold, $f\ (t, z) = fold\ h\ t$, such that a reference to the value z may eventually occur in the algebra h. Another alternative is to define a function of type $\mu F \to (z \to a)$ in terms of a higher-order fold.

In this work, we consider a third alternative consisting of defining $f :: (\mu F, z) \to a$ in terms of a program scheme called *pfold* (a fold with parameters) [19]. The definition of pfold relies on the concept of *strength* of a functor F, a polymorphic function $\tau_F :: (F\ a, z) \to F\ (a, z)$ that satisfies certain coherence axioms (see [19, 6] for details). The strength distributes the value of type z to the variable positions (of type a) of the functor. For instance, the strength corresponding to functor E is given by:

$$\tau_E :: (E\ a, z) \to E\ (a, z)$$
$$\tau_E\ (FNum\ n, z) = FNum\ n$$
$$\tau_E\ (FAdd\ a\ a', z) = FAdd\ (a, z)\ (a', z)$$

The strength plays an important role in the definition of pfold as it represents the distribution of the context information to the recursive calls.

Let $(f \vartriangle g)\ x = (f\ x, g\ x)$. Given a functor F and a function $h :: (F\ a, z) \to a$, *pfold* [19] is defined as the least function such that:

$$pfold\ h \circ (in_F \times id) = h \circ ((map_F\ (pfold\ h) \circ \tau_F) \vartriangle \pi_2)$$

A function h is something similar to an algebra, but it also accepts the value of the parameters. In fact, if functor F is given by n constructors,

$$\mathbf{data}\ F\ a = FC_1\ \sigma_{1,1} \cdots \sigma_{1,r_1}\ \mid \cdots \mid\ FC_n\ \sigma_{n,1} \cdots \sigma_{n,r_n}$$

then, similar to an algebra, $h :: (F\ a, z) \to a$ has n component functions $(h_1, \ldots, h_n)$, now each with type $h_i :: \sigma_{i,1} \to \cdots \to \sigma_{i,r_i} \to z \to a$. Like with the algebras, in the specific instances of pfold we write the tuple of components functions instead of h. For example, for the E functor,

$$h :: (E\ a, z) \to a$$
$$h\ (FNum\ n, z) = hnum\ n\ z$$
$$h\ (FAdd\ a\ a', z) = hadd\ a\ a'\ z$$

where $hnum :: Int \to z \to a$ and $hadd :: a \to a \to z \to a$ are the component functions. Pfold for expressions is then defined by:

$$pfold_E :: (Int \to z \to a, a \to a \to z \to a) \to (Exp, z) \to a$$
$$pfold_E\ (hnum, hadd) = p_E$$
$$\mathbf{where}$$
$$p_E\ (Num\ n, z) = hnum\ n\ z$$
$$p_E\ (Add\ e\ e', z) = hadd\ (p_E\ (e, z))\ (p_E\ (e', z))\ z$$

The following equation shows one of the possible relationships between pfold and fold. For h with component functions $(h_1, \ldots, h_n)$, fixing the value z we have that:

$$pfold\ h\ (t, z) = fold\ k\ t\ \ \mathbf{where}\ \ k_i\ \bar{x} = h_i\ \bar{x}\ z \qquad (1)$$

Observe that k is an algebra with components $(k_1, \ldots, k_n)$. We denote by $\bar{x}$ a sequence of values $x_1 \cdots x_{r_i}$.

The laws for deriving monadic circular programs we are proposing are a monadic extension of the following one, devoted to the derivation of purely functional circular programs [9]. It takes a composition of a producer $p :: a \to (t, z)$ followed by a consumer $c :: (t, z) \to b$, and returns an equivalent deforested, circular program that performs a single traversal over the input value. The law assumes that t is a recursive datatype, the consumer c is defined by structural recursion on t, and the intermediate value of type z is a constant parameter for c; in other words, it assumes that c is a pfold. In addition, it is required that p is a "good producer", in the sense that it is possible to express it in terms of a kind of *build*. Examples of the use of this law and a proof can be found in [9].

LAW 4.3 (PFOLD/BUILDP). *Let h with components $(h_1, \ldots, h_n)$.*

$$
\begin{aligned}
&\mathit{pfold}\ h\ (\mathit{buildp}\ g\ c) \\
&\quad = \mathbf{let}\ (v, z) = g\ k\ c \\
&\qquad\quad k_i\ \bar{x} = h_i\ \bar{x}\ z \\
&\quad \mathbf{in}\ v
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{buildp} :: (\forall\ a\ .\ (F\ a \to a) \to c \to (a, z)) \to c \to (\mu F, z) \\
&\mathit{buildp}\ g = g\ \mathit{in}_F
\end{aligned}
$$

4.4 Extended shortcut fusion

Shortcut fusion laws for monadic programs can be obtained as a special case of an extended form of shortcut fusion that captures the case when the intermediate data structure is generated as part of another structure given by a functor [18, 11]. Such extension is based on an extended form of build: Given a functor F (signature of a datatype) and another functor N, we can define

$$
\begin{aligned}
&\mathit{build}_N :: (\forall\ a\ .\ (F\ a \to a) \to c \to N\ a) \to c \to N\ \mu F \\
&\mathit{build}_N\ g = g\ \mathit{in}_F
\end{aligned}
$$

This is a natural extension of the standard *build*. In fact, *build* can be obtained from build_N by considering the identity functor as N. Moreover, *buildp* is also a particular case obtained by considering the functor $N\ a = (a, z)$.

Using build_N it is possible to state an extended form of shortcut fusion (see [18, 11] for a proof):

LAW 4.4 (EXTENDED FOLD/BUILD). *For strict h and strictness preserving N,*

$$\mathit{map}_N\ (\mathit{fold}\ h) \circ \mathit{build}_N\ g = g\ h$$

Similarly, we can also consider an extension for *buildp*:

$$
\begin{aligned}
&\mathit{buildp}_N \quad :: (\forall\ a\ .\ (F\ a \to a) \to c \to N\ (a, z)) \\
&\qquad\qquad \to c \to N\ (\mu F, z) \\
&\mathit{buildp}_N\ g = g\ \mathit{in}_F
\end{aligned}
$$

with the following shortcut fusion law:

LAW 4.5 (EXTENDED FOLD/BUILDP). *For strict h and strictness-preserving N,*

$$\mathit{map}_N\ (\mathit{fold}\ h \times \mathit{id}) \circ \mathit{buildp}_N\ g = g\ h$$

Proof By considering $N'\ a = N\ (a, z)$, we have that $\mathit{buildp}_N\ g = \mathit{build}_{N'}\ g$ and $\mathit{map}_{N'}\ f = \mathit{map}_N\ (f \times \mathit{id})$. Then, the left-hand side of the equation can be rewritten as: $\mathit{map}_{N'}\ (\mathit{fold}\ h) \circ \mathit{build}_{N'}\ g$. Finally, we apply Law 4.4. □

4.5 Monadic shortcut fusion

The case we are interested in is when the functor N is the composition of a monad m with a product: For some type z,

$$N\ a = m\ (a, z) \qquad \text{and} \qquad \mathit{map}_N\ f = \mathit{mmap}\ (f \times \mathit{id})$$

where

$$
\begin{aligned}
&\mathit{mmap} \quad :: \mathit{Monad}\ m \Rightarrow (a \to b) \to (m\ a \to m\ b) \\
&\mathit{mmap}\ f\ m = \mathbf{do}\ \{a \leftarrow m; \mathit{return}\ (f\ a)\}
\end{aligned}
$$

is the map function for the monad m. The producer then corresponds to a monadic version of buildp:

$$
\begin{aligned}
&\mathit{mbuildp} :: \mathit{Monad}\ m \\
&\quad \Rightarrow (\forall\ a\ .\ (F\ a \to a) \to c \to m\ (a, z)) \\
&\qquad \to c \to m\ (\mu F, z) \\
&\mathit{mbuildp}\ g = g\ \mathit{in}_F
\end{aligned}
$$

A monadic shortcut fusion law can be directly obtained as an instance of Law 4.5. We unfold the definition of mmap so that we get a formulation in terms of do-notation:

LAW 4.6 (FOLD/MBUILDP). *For strict k and strictness preserving mmap,*

$$\mathbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}\ g\ c; \mathit{return}\ (\mathit{fold}\ k\ t, z)\} = g\ k\ c$$

This is a version for $\mathit{mbuildp}$ of the shortcut fusion law introduced by Manzino and Pardo [18].

Using this law we can state a first monadic extension of the pfold/buildp rule. Observe that the introduction of the circularity on z requires the monad to be *recursive* [8] as it is necessary the use of a circular binding within a monadic computation. In Haskell terms, this can be expressed using the **mdo**-notation provided that the monad is an instance of the $\mathit{MonadFix}$ class.

LAW 4.7 (PFOLD/MBUILDP). *Let m be a recursive monad with strictness-preserving mmap. For h with components $(h_1, \ldots, h_n)$ and strict,*

$$
\begin{aligned}
&\mathbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}\ g\ c; \mathit{return}\ (\mathit{pfold}\ h\ (t, z))\} \\
&= \\
&\mathbf{mdo}\{(v, z) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ g\ k\ c; \mathit{return}\ v\}
\end{aligned}
$$

Proof

$$
\begin{aligned}
&\mathbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}\ g\ c; \mathit{return}\ (\mathit{pfold}\ h\ (t, z))\} \\
=\ &\{\ (1)\ \} \\
&\mathbf{do}\ (t, z) \leftarrow \mathit{mbuildp}\ g\ c \\
&\quad \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t) \\
=\ &\{\ \text{definition of } \pi_1\ \} \\
&\mathbf{do}\ (t, z) \leftarrow \mathit{mbuildp}\ g\ c \\
&\quad \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ \mathit{return}\ (\pi_1\ (\mathit{fold}\ k\ t, z)) \\
=\ &\\
&\mathbf{do}\ (t, z) \leftarrow \mathit{mbuildp}\ g\ c \\
&\quad (v, z') \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z) \\
&\quad \mathit{return}\ v \\
=\ &\{\ \text{circularity introduction}\ \} \\
&\mathbf{mdo}\ (t, z) \leftarrow \mathit{mbuildp}\ g\ c \\
&\quad (v, z') \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z'\ \mathbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z) \\
&\quad \mathit{return}\ v \\
=\ &\{\ \text{Law 4.6}\ \} \\
&\mathbf{mdo}\{(v, z') \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z'\ \mathbf{in}\ g\ k\ c; \mathit{return}\ v\}
\end{aligned}
$$

□

The circularity introduced is safe and therefore computations can be ordered under lay evaluation. In fact, to compute v it is necessary to compute z', which in turn depends on z. That is, even with the introduction of the circularity, the computation of the final value depends on the computation of the second value computed by the $\mathit{mbuildp}\ g\ c$.

When the consumer is also an effectful function, it is possible to state two other laws, similar to Laws 4.6 and 4.7, respectively, but that deal with fusion of effectful functions. The formulation of these laws follow the approach presented by Chitil [5] and Ghani and Johann [11].

LAW 4.8 (EFFECTFUL FOLD/MBUILDP). *For strict* $k{::}F\ (m\ a) \to m\ a$ *and strictness preserving* $mmap$*,*

$$\begin{array}{l}\quad \mathbf{do}\ \{(t,z) \leftarrow mbuildp\ g\ c; v \leftarrow fold\ k\ t; return\ (v,z)\} \\ = \\ \quad \mathbf{do}\ \{(m,z) \leftarrow g\ k\ c; v \leftarrow m; return\ (v,z)\}\end{array}$$

Proof

$$\begin{array}{l}\mathbf{do}\ \{(t,z) \leftarrow mbuildp\ g\ c; v \leftarrow fold\ k\ t; return\ (v,z)\} \\ \quad = \mathbf{do}\ (t,z) \leftarrow mbuildp\ g\ c \\ \qquad\quad (m,\_) \leftarrow return\ (fold\ k\ t, z) \\ \qquad\quad v \leftarrow m \\ \qquad\quad return\ (v,z) \\ \quad = \mathbf{do}\ (m,z) \leftarrow \mathbf{do}\ (t,z) \leftarrow mbuildp\ g\ c \\ \qquad\qquad\qquad\quad return\ (fold\ k\ t, z) \\ \qquad\quad v \leftarrow m \\ \qquad\quad return\ (v,z) \\ \quad = \mathbf{do}\ \{(m,z) \leftarrow g\ k\ c; v \leftarrow m; return\ (v,z)\}\end{array}$$

□

Using this law we can now state a shortcut fusion law for the derivation of monadic circular programs in those cases when both the producer and consumer are effectful functions. Again, like in Law 4.7, the monad is required to be recursive because of the introduction of a recursive binding within the monadic computation.

LAW 4.9 (EFFECTFUL PFOLD/MBUILDP). *Let* m *be a recursive monad with strictness-preserving* $mmap$*. For* $h :: (F\ (m\ a), z) \to m\ a$ *with components* $(h_1, \ldots, h_n)$ *and strict,*

$$\begin{array}{l}\quad \mathbf{do}\ \{(t,z) \leftarrow mbuildp\ g\ c; pfold\ h\ (t,z)\} \\ = \\ \quad \mathbf{mdo}\{(m,z) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ g\ k\ c; m\}\end{array}$$

Proof

$$\begin{array}{l}\quad \mathbf{do}\ \{(t,z) \leftarrow mbuildp\ g\ c; pfold\ h\ (t,z)\} \\ = \\ \quad \mathbf{do}\ (t,z) \leftarrow mbuildp\ g\ c \\ \qquad m \leftarrow return\ (pfold\ h\ (t,z)) \\ \qquad m \\ = \quad \{\ \text{Law 4.7}\ \} \\ \quad \mathbf{mdo}\{(m,z) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ g\ k\ c; m\}\end{array}$$

□

5. Higher-order shortcut fusion

There exists an alternative way to transform compositions between *pfold* and *buildp* such that, instead of circular programs, higher-order programs are obtained as result. Obtaining higher-order programs is attractive since the execution of such programs is not restricted to a lazy evaluation setting. Furthermore, the running performance of a higher-order program is often better than the performance of its circular equivalent.

The alternative transformation presented in this section is based on the fact that every *pfold* can be expressed in terms of a higher-order fold: For $h :: (F\ a, z) \to a$,

$$pfold\ h = apply \circ (fold\ \varphi_h \times id) \tag{2}$$

where $\varphi_h :: F\ (z \to a) \to (z \to a)$ is given by

$$\varphi_h = curry\ (h \circ ((map_F\ apply \circ \tau_F) \vartriangle \pi_2))$$

and $apply :: (a \to b, a) \to b$ by $apply\ (f,x) = f\ x$. Therefore, $fold\ \varphi_h :: \mu F \to (z \to a)$ is the curried version of $pfold\ h$.

With this relationship at hand we can state the following shortcut fusion law, which is the instance to our context of a more general program transformation technique called lambda abstraction [20]. The specific case of this law when lists are the intermediate structure was recently introduced by Voigtländer [25].

LAW 5.1 (HIGHER-ORDER PFOLD/BUILDP). *For left-strict* h*,*[3]

$$pfold\ h \circ buildp\ g = apply \circ g\ \varphi_h$$

Like in the derivation of circular programs, $g\ \varphi_h$ returns a pair, but now composed by a function of type $z \to a$ and an object of type z. The final result then corresponds to the application of the function to the object. That is,

$$pfold\ h\ (buildp\ g\ c) = \mathbf{let}\ (f,z) = g\ \varphi_h\ c\ \mathbf{in}\ f\ z$$

Of course, this law can be generalized in the sense of extended shortcut fusion.

LAW 5.2. *For left-strict* h *and strictness-preserving* N*,*

$$map_N\ (pfold\ h) \circ buildp_N\ g = map_N\ apply \circ g\ \varphi_h$$

Proof

$$\begin{array}{l}\quad map_N\ (pfold\ h) \circ buildp_N\ g \\ = \quad \{\ (2)\ \} \\ \quad map_N\ apply \circ map_N\ (fold\ \varphi_h \times id) \circ buildp_N\ g \\ = \quad \{\ \text{Law 4.5}\ \} \\ \quad map_N\ apply \circ g\ \varphi_h\end{array}$$

□

Like in Subsection 4.5, we can consider the particular case when N is the functor of a monad. Unlike the transformation to circular programs, now we do not need to require the monad to be recursive. Again, in first place we state the case when the *pfold* is a pure function.

LAW 5.3 (HIGHER-ORDER PFOLD/MBUILDP). *For left-strict* h *and strictness-preserving* $mmap$*,*

$$\begin{array}{l}\quad \mathbf{do}\ \{(t,z) \leftarrow mbuildp\ g\ c; return\ (pfold\ h\ (t,z))\} \\ = \\ \quad \mathbf{do}\ \{(f,z) \leftarrow g\ \varphi_h\ c; return\ (f\ z)\}\end{array}$$

To see an example of the application of this law we consider again function *shift* from Section 2:

$$\begin{array}{l}shift :: Parser\ [Bit] \\ shift = \mathbf{do}\ (bs,s) \leftarrow bitstring \\ \qquad\qquad return\ (transform\ (bs,s))\end{array}$$

and write the higher-order fold corresponding to $transform$:

$$\begin{array}{l}transform_{ho} :: [Bit] \to (Bit \to [Bit]) \\ transform_{ho} = fold_L\ (\varphi_n, \varphi_c) \\ \quad \mathbf{where}\ \varphi_n \quad = \lambda\_ \to [\,] \\ \qquad\qquad \varphi_c\ b\ r = \lambda s \to (b \oplus s) : r\ s\end{array}$$

Then, by applying Law 5.3 we obtain:

$$\begin{array}{l}shift_{ho} = \mathbf{do}\ (f,s) \leftarrow g\ (\varphi_n, \varphi_c) \\ \qquad\qquad\quad return\ (f\ s)\end{array}$$

[3] By left-strict we mean strict on the first argument, that is, $h\ (\bot, z) = \bot$.

Inlining,

$$
\begin{array}{l}
shift_{ho} = \mathbf{do}\ (f, s) \leftarrow g_\varphi \\
\qquad\qquad\ \ return\ (f\ s) \\
\quad \mathbf{where}\ g_\varphi = \mathbf{do}\ b \quad\ \leftarrow bit \\
\qquad\qquad\qquad\ \ (f, s) \leftarrow g_\varphi \\
\qquad\qquad\qquad\ \ return\ (\lambda s' \rightarrow (b \oplus s') : f\ s', b \oplus s) \\
\qquad\qquad\quad \langle|\rangle\ return\ (\lambda\_ \rightarrow [\,], Z)
\end{array}
$$

Finally, we present the case when the consumer is also an effectful function.

LAW 5.4 (EFFECTFUL HIGHER-ORDER PFOLD/MBUILDP). *For left-strict* $h :: (F\ (m\ \ a), z) \rightarrow m\ \ a$ *and strictness-preserving* $mmap$*,*

$$
\begin{array}{l}
\quad \mathbf{do}\ \{(t, z) \leftarrow mbuildp\ g\ c; pfold\ h\ (t, z)\} \\
= \\
\quad \mathbf{do}\ \{(f, z) \leftarrow g\ \varphi_h\ c; f\ z\}
\end{array}
$$

Proof

$$
\begin{array}{l}
\quad \mathbf{do}\ \{(t, z) \leftarrow mbuildp\ g\ c; pfold\ h\ (t, z)\} \\
= \\
\quad \mathbf{do}\ (t, z) \leftarrow mbuildp\ g\ c \\
\qquad m \quad\ \leftarrow return\ (pfold\ h\ (t, z)) \\
\qquad m \\
= \qquad \{\ \text{Law 5.3}\ \} \\
\quad \mathbf{do}\ m \leftarrow \mathbf{do}\ \{(f, z) \leftarrow g\ \varphi_h\ c; return\ (f\ z)\} \\
\qquad m \\
= \\
\quad \mathbf{do}\ \{(f, z) \leftarrow g\ \varphi_h\ c; f\ z\}
\end{array}
$$

□

We conclude this section with an example of the application of this law. To this end we consider again the function *semantics* presented in Section 3:

$$
\begin{array}{l}
semantics :: Prog \rightarrow IO\ [String] \\
semantics\ p = \mathbf{do}\ (p', env) \leftarrow duplicate\ [\,]\ p \\
\qquad\qquad\qquad\ \ missing\ (p', env)
\end{array}
$$

To apply the law we need the expression of the algebra of the higher-order fold, which corresponds to the curried version of *missing*:

$$
\begin{array}{l}
missing_{ho} :: Prog_2 \rightarrow ([String] \rightarrow IO\ [String]) \\
missing_{ho} = fold_L\ (\varphi_n, \varphi_c) \\
\quad \mathbf{where} \\
\qquad \varphi_n = \lambda\_ \rightarrow return\ [\,] \\
\qquad \varphi_c\ (Dupl_2\ v)\ mr \\
\qquad\quad = \lambda env \rightarrow \mathbf{do}\ r \leftarrow mr\ env \\
\qquad\qquad\qquad\qquad return\ (v : r) \\
\qquad \varphi_c\ (Use_2\ v)\ mr \\
\qquad\quad = \lambda env \rightarrow \mathbf{do}\ r \leftarrow mr\ env \\
\qquad\qquad\qquad\qquad \mathbf{if}\ (v \in env) \\
\qquad\qquad\qquad\qquad \mathbf{then}\ return\ r \\
\qquad\qquad\qquad\qquad \mathbf{else\ do}\ put\ (\texttt{"Missing:decl"} +\!\!+ v) \\
\qquad\qquad\qquad\qquad\qquad return\ (v : r)
\end{array}
$$

Then, by Law 5.4 we obtain:

$$
\begin{array}{l}
semantics_{ho} = \mathbf{do}\ (f, env) \leftarrow g\ (\varphi_n, \varphi_c) \\
\qquad\qquad\qquad\ \ f\ env
\end{array}
$$

Inlining this definition, we finally derive the higher-order program:

$$
\begin{array}{l}
semantics_{ho} = \mathbf{do}\ (f, env) \leftarrow g_\varphi \\
\qquad\qquad\qquad\ \ f\ env \\
\quad \mathbf{where} \\
\qquad g_\varphi\ ds\ [\,] = return\ (\lambda\_ \rightarrow return\ [\,], ds) \\
\qquad g_\varphi\ ds\ (Decl\ v : p) \\
\qquad\quad = \mathbf{do}\ (p', ds') \leftarrow g_\varphi\ (v : ds)\ p \\
\qquad\qquad\quad \mathbf{if}\ (v \in ds) \\
\qquad\qquad\quad \mathbf{then\ do}\ put\ (\texttt{"Duplicate:decl"} +\!\!+ v) \\
\qquad\qquad\qquad\qquad return\ (\lambda env \rightarrow \mathbf{do}\ r \leftarrow p'\ env \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ (v : r) \\
\qquad\qquad\qquad\qquad\qquad , ds') \\
\qquad\qquad\quad \mathbf{else}\ \ return\ (p', ds') \\
\qquad g_\varphi\ ds\ (Use\ v : p) \\
\qquad\quad = \mathbf{do}\ (p', ds') \leftarrow g_\varphi\ ds\ p \\
\qquad\qquad\quad return\ (\\
\qquad\qquad\qquad \lambda env \rightarrow \\
\qquad\qquad\qquad\quad \mathbf{do}\ r \leftarrow p'\ env \\
\qquad\qquad\qquad\qquad \mathbf{if}\ (v \in env) \\
\qquad\qquad\qquad\qquad \mathbf{then}\ return\ r \\
\qquad\qquad\qquad\qquad \mathbf{else\ do}\ put\ (\texttt{"Missing:decl"} +\!\!+ v) \\
\qquad\qquad\qquad\qquad\qquad return\ (v : r) \\
\qquad\qquad\qquad , ds')
\end{array}
$$

6. Conclusions and Future Work

6.1 Contributions

In this paper, we have presented shortcut fusion rules for the derivation of circular and higher-order monadic programs. The rules are generic, as they can be instantiated for a wide class of algebraic data types and monads. We have also shown two example applications which demonstrate the practical interest of the rules.

It is possible to obtain a prototype implementation of the rules using the rewrite rules (RULES pragma) of the Glasgow Haskell Compiler (GHC). Experimental results showing time and space comparisons for a set of examples are available in the webpage `http://www.fing.edu.uy/~pardo/ExtendedShortcut/`.

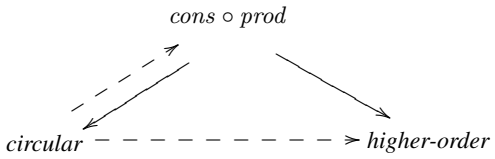
The main contributions of this paper can be summarized as follows:

(i) using an extension to shortcut fusion that captures the cases when the intermediate structures are generated as part of another structure given by a functor, we obtained shortcut fusion rules that transform compositions of monadic programs into circular programs. This extends to monadic programs the work we have presented for pure programs in [9].

(ii) we generalized the shortcut fusion rule for deriving pure higher-order programs presented in [25] so that it can be applied to compositions of programs with an arbitrary data type as intermediate structure.

(iii) we presented an extension of the above rule for the derivation of higher-order programs to the case of monadic programs.

The following diagram summarizes the results that have been achieved so far:



The arrow from $cons \circ prod$ to *circular programs* corresponds to the rules developed in this work and in [9]. The arrow from $cons \circ prod$ to *higher-order programs* corresponds to (ii) and (iii) above. The arrows from *circular programs* to $cons \circ prod$ and *higher-order programs* correspond to the transformations based on Attribute Grammar techniques presented in [10].

6.2 Future work

Various aspects of the ideas presented in this paper deserve further elaboration.

Multiple intermediate structure elimination The examples we presented in this paper consist of compositions of a single producer and consumer functions. We would like, however, to be able to achieve the same fusion goals for programs consisting in an arbitrary number of function compositions. Indeed, we are now studying how to generalize our work in order to optimize programs of the form $f_n \circ ... \circ f_1$ such that in each composition a data structure t_i and a value z_i are produced. We will describe such a generalization in a forthcoming paper, already under preparation.

Relation with Attribute Grammars Circular programs, monads and attribute grammars (AGs) are closely related [21]. Indeed, AG techniques are used to model and manipulate circular programs in order to derive efficient non-lazy equivalent programs [10], and several circular-based AG systems have been developed [22, 28]. In particular, we would like to express the transformations in [10] (the dashed arrows in the diagram above) in a calculational form, so that their correctness can be proved. Indeed, although these techniques are largely used by the AG community, their correctness remains to be formally proved! The techniques studied in this paper also serve the purpose of increasing the knowledge on the relation between circular programs and AGs, and, therefore, bring us close to our goal of building a proved correct AG system.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[2] R. Bird. *Introduction to Functional Programming using Haskell,* 2nd edition. Prentice-Hall, UK, 1998.

[3] Richard Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *Prentice-Hall Inernational Series in Computer Science*. Prentice-Hall, 1997.

[4] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf*, 21:239–250, 1984.

[5] O. Chitil. *Type-inference based deforestation of functional programs*. PhD thesis, RWTH Aachen, October 2000.

[6] R. Cockett and D. Spencer. Strong Categorical Datatypes I. In R.A.C. Seely, editor, *International Meeting on Category Theory 1991,* volume 13 of *Canadian Mathematical Society Conference Proceedings*, pages 141–169, 1991.

[7] Oege de Moor, Simon Peyton-Jones, and Eric van Wyk. Aspect-oriented compilers. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, LNCS, September 1999.

[8] L. Erkök and J. Launchbury. A Recursive do for Haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 29–37. ACM, 2002.

[9] João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell '07: Proceedings of the ACM SIGPLAN Haskell workshop*, pages 95–106, New York, NY, USA, 2007. ACM.

[10] João Paulo Fernandes and João Saraiva. Tools and Libraries to Model and Manipulate Circular Programs. In *Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM'07)*, pages 102–111. ACM Press, 2007.

[11] N. Ghani and P. Johann. Short Cut Fusion of Recursive Programs with Computational Effects. In *Symposium on Trends in Functional Programming (TFP 2008)*, 2008.

[12] J. Gibbons. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction,* LNCS 2297, pages 148–203. Springer-Verlag, January 2002.

[13] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, UK, 1996.

[14] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[15] Dennis Kugler Harald Baier and Marian Margraf. Elliptic Curve Cryptography Based on ISO 15946. Technical Report TR-03111, Federal Office for Information Security, 2007.

[16] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[17] Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 99–110, New York, NY, USA, 2004. ACM.

[18] C. Manzino and A. Pardo. Short Cut Fusion of Monadic Programs. In *Brazilian Symposium on Programming Languages (SBLP)*, 2008.

[19] A. Pardo. Generic Accumulations. In *IFIP WG2.1 Working Conf. on Generic Programming*, Dagstuhl, Germany, July 2002.

[20] Alberto Pettorossi and Andrzej Skowron. The lambda abstraction strategy for program derivation. In *Fundamenta Informaticae XII*, pages 541–561, 1987.

[21] Doaitse Swierstra. Tutorial on attribute grammars. In *Generative Programming and Component Engineering*, 1993.

[22] S. D. Swierstra, Arthur Baars, and Andres Löh. The UU-AG attribute grammar system.

[23] A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Functional Programming Languages and Computer Architecture'95*, 1995.

[24] Janis Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17:129–163, 2004. Previous version appeared in *ASIA-PEPM 2002*, Proceedings, pages 126–137, ACM Press, 2002.

[25] Janis Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *FLOPS '08: Proceedings of the 2008 International Symposium on Functional and Logic Programming*, pages 163–179. Springer-Verlag, 2008.

[26] P. Wadler. Theorems for free! In *4th International Conference on Functional Programming and Computer Architecture*, London, 1989.

[27] William Waite, Uwe Kastens, and Anthony M. Sloane. *Generating Software from Specifications*. Jones and Bartlett Publishers, Inc., USA, 2007.

[28] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, Eric Johnson, August Schwerdfeger, and Phil Russell. Tool Demonstration: Silver Extensible Compiler Frameworks and Modular Language Extensions for Java and C. In *SCAM*, page 161, 2006.