Shortcut Fusion of Monadic Programs

Cecilia Manzino[1], **Alberto Pardo**[2]

[1] Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario, Argentina

[2]Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay

`ceciliam@fceia.unr.edu.ar, pardo@fing.edu.uy`

Abstract. *Functional programs often combine separate parts of the program using intermediate data structures for communicating results. Programs so defined are easier to understand and maintain, but suffer from inefficiency problems due to the generation of those data structures. In response to this problematic, some program transformation techniques have been studied with the aim to eliminate the intermediate data structures that arise in function compositions. One of these techniques is known as shortcut fusion. This technique has usually been studied in the context of purely functional programs. In this work we propose an extension of shortcut fusion that is able to eliminate intermediate data structures generated in the presence of monadic effects. The extension to be presented can be uniformly defined for a wide class of data types and monads.*

1. Introduction

Functional programs often combine separate parts of the program using intermediate data structures for communicating results. Programs so defined have many benefits, such as clarity, modularity, and maintainability, but suffer from inefficiencies caused by the generation of those data structures. In response to this problematic, some program transformation techniques have been developed aiming at the elimination of the intermediate data structures. One of these techniques, known as shortcut fusion (or shortcut deforestation) [Gill et al. 1993], has mainly been studied in the context of purely functional programs.

The aim of this paper is the proposal of an extension of shortcut fusion for programs with monadic effects. The goal is to achieve fusion of monadic programs, maintaining the global effects. Like standard deforestation, we will be interested in eliminating the intermediate data structures generated in function compositions, but with the difference that now those intermediate structures are produced as the result of monadic computations. An important feature of the extension to be presented is that it is generic, in the sense that it is given by a uniform, single definition that can then be instantiated to a wide class of data types and monads.

This work has strong connections with previous work on fusion techniques for recursion schemes for programs with effects [Pardo 2001, Pardo 2005]. The main difference is that in the present work we adopt a shortcut fusion approach based on parametricity properties of polymorphic functions, known in functional programming as free theorems [Wadler 1989]. The results of this paper were preliminary presented in [Manzino 2005]. Throughout we will use Haskell notation, assuming a cpo semantics (in terms of pointed cpos), but without the presence of the seq function [Johann and Voigtländer 2004].

The paper is organized as follows. We start in Section 2 with a review of the concept of shortcut fusion. In Section 3, by means of specific examples, we show the extension of shortcut fusion to programs with effects. The generic constructions that give rise to the specific program schemes and laws presented in Sections 2 and 3 are developed in Section 4; a proof of the monadic shortcut fusion law is also presented. Section 5 summarizes related work, and Section 6 concludes the paper.

2. Shortcut fusion

Shortcut fusion [Gill et al. 1993] is a program transformation technique originally proposed for lists, but that can be defined for other datatypes as well. Given a function composition $c \circ p$, the idea of shortcut fusion is to eliminate the intermediate data structure produced by p (the *producer*) and consumed by c (the *consumer*) by a suitable combination of their definitions. We analyse the case of lists and arithmetic expressions.

Lists Shortcut fusion requires the consumer to be given by a structural recursive definition that treats all elements of a list in a uniform manner. This is captured by a recursion scheme called fold:[1]

$$\begin{array}{ll}
\mathit{fold}_L & :: (b, a \to b \to b) \to [a] \to b \\
\mathit{fold}_L\ (\mathit{nil}, \mathit{cons})\ [\,] & = \mathit{nil} \\
\mathit{fold}_L\ (\mathit{nil}, \mathit{cons})\ (a : \mathit{as}) & = \mathit{cons}\ a\ (\mathit{fold}_L\ (\mathit{nil}, \mathit{cons})\ \mathit{as})
\end{array}$$

A fold is a function that traverses the input list and replaces the occurrences of the list constructors $[\,]$ and $(:)$ by nil and cons, respectively. For example, $\mathit{fold}_L\ (\mathit{nil}, \mathit{cons})$ applied to the list $1 : 2 : 3 : [\,]$ returns the expression $\mathit{cons}\ 1\ (\mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil}))$.

The producer, on the other hand, is required to be able to show that the list constructors can be abstracted from the process that generates the intermediate list. This is expressed by a function called build:

$$\begin{array}{l}
\mathit{build}_L \quad :: (\forall\ b\ .\ (b, a \to b \to b) \to b) \to [a] \\
\mathit{build}_L\ g = g\ ([\,], (:))
\end{array}$$

For example, the list $1 : 2 : 3 : [\,]$ can be written as $\mathit{build}_L\ (\lambda(n, c) \to c\ 1\ (c\ 2\ (c\ 3\ n)))$.

With the forms required to the producer and the consumer it is now possible to state the following fusion law, known as *shortcut fusion*.

Law 1 (fold/build for lists)

$$\mathit{fold}_L\ (\mathit{nil}, \mathit{cons})\ (\mathit{build}_L\ g) = g\ (\mathit{nil}, \mathit{cons})$$

The intuition behind this law is the following: since g explicitly exhibits that the intermediate list generation relies on the constructors $[\,]$ and $(:)$, and those constructors are immediately replaced by nil and cons by the fold, then the final result corresponds to g applied directly to nil and cons.

To see an example, consider the following definition of factorial:

[1]The fold for lists is known as *foldr* in the functional programming jargon [Bird 1998].

$$fact\ n = product\ (down\ n)$$

$$\begin{aligned}
&product && :: [Int] \to Int \\
&product\ [\,] && = 1 \\
&product\ (a : as) && = a * product\ as
\end{aligned}$$

$$\begin{aligned}
&down && :: Int \to [Int] \\
&down\ 0 && = [\,] \\
&down\ n && = n : down\ (n-1)
\end{aligned}$$

Given n, we first compute the list of numbers between n and 1 and then calculate their product. However, as we all know, it is not necessary to produce an intermediate list to compute factorial. The listless definition can be obtained by shortcut fusion. To do so we need to express the $product$ and $down$ in terms of $fold$ and $build$, respectively.

$$product = fold_L\ (1, (*))$$

$$\begin{aligned}
&down\ n = build_L\ (gdown\ n) \\
&\quad \mathbf{where}\ gdown\ 0\ (nil, cons) = nil \\
&\qquad\qquad\ \ gdown\ n\ (nil, cons) = cons\ n\ (gdown\ (n-1)\ (nil, cons))
\end{aligned}$$

By applying Law 1 we obtain $fact\ n = gdown\ n\ (1, (*))$, which corresponds to the usual definition of factorial: $fact\ 0 = 1$ and $fact\ n = n * fact\ (n-1)$.

Arithmetic expressions Consider a datatype for simple arithmetic expressions formed by numerals and addition.

$$\mathbf{data}\ Exp = Num\ Int \mid Add\ Exp\ Exp$$

The fold and build functions for this datatype are defined as follows:

$$\begin{aligned}
&fold_E && :: (Int \to a, a \to a \to a) \to Exp \to a \\
&fold_E\ (num, add)\ (Num\ n) && = num\ n \\
&fold_E\ (num, add)\ (Add\ e\ e') && = add\ (fold_E\ (num, add)\ e)\ (fold_E\ (num, add)\ e')
\end{aligned}$$

$$\begin{aligned}
&build_E\ :: (\forall\ a\ .\ (Int \to a, a \to a \to a) \to a) \to Exp \\
&build_E\ g = g\ (Num, Add)
\end{aligned}$$

Law 2 (fold/build for expressions)

$$fold_E\ (num, add)\ (build_E\ g) = g\ (num, add)$$

3. Monadic shortcut fusion

In functional programming, monads are a powerful mechanism to structure programs that produce effects, such as exceptions, state, or input/output [Wadler 1995]. A *monad* is usually presented as a triple formed by a type constructor m, a polymorphic function $return$ and a polymorphic operator $(\gg\!=)$ (often called *bind*), such that certain laws are satisfied [Wadler 1995]. In Haskell, a monad can be defined in terms of a class:

$$\mathbf{class}\ Monad\ m\ \mathbf{where}$$

$$\begin{array}{ll} return & :: a \to m\ a \\ (\gg\!=) & :: m\ a \to (a \to m\ b) \to m\ b \\ (\gg) & :: m\ a \to m\ b \to m\ b \\ \multicolumn{2}{l}{m \gg m' = m \gg\!= \lambda\_ \to m'} \end{array}$$

With the aim at improving readability of monadic programs, Haskell provides a special syntax called the *do notation*. It is defined by the following translation rules:

$$\begin{array}{rcl} \mathbf{do}\ \{x \leftarrow m; m'\} & = & m \gg\!= \lambda x \to \mathbf{do}\ \{m'\} \\ \mathbf{do}\ \{m; m'\} & = & m \gg \mathbf{do}\ \{m'\} \\ \mathbf{do}\ \{m\} & = & m \end{array}$$

Associated with every monad it is possible to define a map function, which together with the type constructor m satisfies to be a functor in the sense we will define in Section 4.

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

Before introducing a law corresponding to monadic shortcut fusion, we first analyse what happens when we consider compositions of effectful functions. After that we present a fusion law that considers a restricted form of monadic compositions in which only the function that generates the intermediate data structure may produce an effect, while the consumer is purely functional. The conception of this law has strong connections with similar laws developed for monadic recursion schemes like the monadic versions of fold and hylomorphism [Pardo 2001, Pardo 2005].

3.1. Fusion of effectful functions

Computations ordering is what makes effectful functions more difficult to be fused. In fact, the main difference with fusion of pure programs is that, when fusing two monadic functions, we must ensure the preservation of the order in which monadic computations are performed. Like in the case of purely functional programs, fusion laws for monadic programs rely on the representation of the involved functions in terms of recursion schemes and the properties those representations require to make fusion possible. For example, when the monadic versions of fold and hylomorphism [Pardo 2001, Pardo 2005] are used as representation, a strong condition to the monad, namely, commutativity, is required in order to make fusion possible. A monad is said to be *commutative* if the order in which computations are performed is irrelevant. The essential property is the following: $\mathbf{do}\ \{a \leftarrow m; b \leftarrow m'; return\ (a, b)\} = \mathbf{do}\ \{b \leftarrow m'; a \leftarrow m; return\ (a, b)\}$. Cases like the state reader (also known as the environment monad) or the identity monad are commutative, while monads like state or IO are not.

We analyze examples on lists and binary trees; the case on trees will raise the necessity of a commutativity condition for the monad.

Lists Consider the following composition of two effectful functions:

$$\begin{array}{ll} displaySeq & :: Show\ a \Rightarrow [IO\ a] \to IO\ () \\ displaySeq\ ms & = \mathbf{do}\ \{xs \leftarrow sequence\ ms; display\ xs\} \end{array}$$

$$\begin{array}{ll}
\mathit{sequence} & :: [\mathit{IO}\ a] \rightarrow \mathit{IO}\ [a] \\
\mathit{sequence}\ [\,] & = \mathit{return}\ [\,] \\
\mathit{sequence}\ (m : \mathit{ms}) & = \mathbf{do}\ \{x \leftarrow m; \mathit{xs} \leftarrow \mathit{sequence}\ \mathit{ms}; \mathit{return}\ (x : \mathit{xs})\}
\end{array}$$

$$\begin{array}{ll}
\mathit{display} & :: \mathit{Show}\ a \Rightarrow [a] \rightarrow \mathit{IO}\ () \\
\mathit{display}\ [\,] & = \mathit{return}\ () \\
\mathit{display}\ (x : \mathit{xs}) & = \mathbf{do}\ \{\mathit{display}\ \mathit{xs}; \mathit{putStr}\ (\mathit{show}\ x)\}
\end{array}$$

$$\mathit{put}\ x = \mathbf{do}\ \{\mathit{putStr}\ (\mathit{show}\ x); \mathit{return}\ x\}$$

The *sequence* function executes a list of IO computations from left-to-right, collecting their results in a list, while *display* prints the elements of a list in reverse order. For example, when applied to the list $[\mathit{put}\ 1, \mathit{put}\ 2, \mathit{put}\ 3]$, *displaySeq* produces the string `"123321"` in the standard output.

A definition of *displaySeq* that avoids the generation of the intermediate list can be derived by case analysis. Two cases have to be considered:

$$\begin{array}{ll}
\mathit{displaySeq}\ [\,] & = \mathit{return}\ () \\
\mathit{displaySeq}\ (m : \mathit{ms}) & = \mathbf{do}\ \{x \leftarrow m; \mathit{displaySeq}\ \mathit{ms}; \mathit{putStr}\ (\mathit{show}\ x)\}
\end{array}$$

In this case fusion succeeds because the computations are in a “suitable” order. This situation can be captured by the following shortcut fusion law, presented by Meijer and Jeuring [Meijer and Jeuring 1995], which is associated to a recursion scheme called monadic fold.

Law 3 (mfold/mbuild for lists)

$$\mathbf{do}\ \{\mathit{as} \leftarrow \mathit{mbuild}_L\ g; \mathit{mfold}_L\ (\mathit{mnil}, \mathit{mcons})\ \mathit{as}\} = g\ (\mathit{mnil}, \mathit{mcons})$$

where

$$\begin{array}{l}
\mathit{mfold}_L :: \mathit{Monad}\ m \Rightarrow (m\ b, a \rightarrow b \rightarrow m\ b) \rightarrow [a] \rightarrow m\ b \\
\mathit{mfold}_L\ (\mathit{mnil}, \mathit{mcons})\ [\,] \quad = \mathit{mnil} \\
\mathit{mfold}_L\ (\mathit{mnil}, \mathit{mcons})\ (a : \mathit{as}) = \mathbf{do}\ \{y \leftarrow \mathit{mfold}_L\ (\mathit{mnil}, \mathit{mcons})\ \mathit{as}; \mathit{mcons}\ a\ y\}
\end{array}$$

$$\begin{array}{l}
\mathit{mbuild}_L \quad :: \mathit{Monad}\ m \Rightarrow (\forall\ b\ .\ (m\ b, a \rightarrow b \rightarrow m\ b) \rightarrow m\ b) \rightarrow m\ [a] \\
\mathit{mbuild}_L\ g = g\ (\mathit{return}\ [\,], \lambda a\ \mathit{as} \rightarrow \mathit{return}\ (a : \mathit{as}))
\end{array}$$

By writing *sequence* and *display* in terms of mbuild_L and mfold_L, respectively, we arrive at the same recursive definition of *displaySeq* shown before.

Trees Now, we present a similar example on trees but that requires the monad to be commutative.

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Join}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\begin{array}{ll}
\mathit{displaySeq}_T & :: \mathit{Show}\ a \Rightarrow \mathit{Tree}\ (\mathit{IO}\ a) \rightarrow \mathit{IO}\ () \\
\mathit{displaySeq}_T\ \mathit{ms} & = \mathbf{do}\ \{t \leftarrow \mathit{seq}_T\ \mathit{ms}; \mathit{display}_T\ t\}
\end{array}$$

$$\begin{array}{ll}
\mathit{seq}_T & :: \mathit{Tree}\ (\mathit{IO}\ a) \rightarrow \mathit{IO}\ (\mathit{Tree}\ a) \\
\mathit{seq}_T\ (\mathit{Leaf}\ m) & = \mathbf{do}\ \{a \leftarrow m; \mathit{return}\ (\mathit{Leaf}\ a)\}
\end{array}$$

$$seq_T\ (Join\ ml\ mr) = \mathbf{do}\ \{l \leftarrow seq_T\ ml; r \leftarrow seq_T\ mr; return\ (Join\ l\ r)\}$$

$$\begin{array}{ll} display_T & :: Show\ a \Rightarrow Tree\ a \rightarrow IO\ () \\ display_T\ (Leaf\ a) & = putStr\ (show\ a) \\ display_T\ (Join\ l\ r) & = \mathbf{do}\ \{display_T\ l; display_T\ r\} \end{array}$$

The seq_T function executes from left-to-right the IO computations stored in the leaves of a tree, while $display_T$ prints the elements that result from those computations. For example, when applied to the tree $Join\ (Leaf\ (put\ 1))\ (Join\ (Leaf\ (put\ 2))\ (Leaf\ (put\ 3)))$, $displaySeq_T$ produces the string `"123123"` in the standard output.

In this case, we would like to eliminate the intermediate tree that is generated by seq_T and consumed by $display_T$. Like for lists, we proceed by case analysis.

$$displaySeq_T\ (Leaf\ m) = \mathbf{do}\ \{a \leftarrow m; putStr\ (show\ a)\}$$

However, in the case of a join node:

$$\begin{array}{l} displaySeq_T\ (Join\ ml\ mr) \\ \quad = \mathbf{do}\ \{l \leftarrow seq_T\ ml; r \leftarrow seq_T\ mr; display_T\ l; display_T\ r\} \end{array}$$

We get stuck at this point as it is not possible to reorder the terms in the do-expression so that to introduce a recursive call to $displaySeq_T$ (a change in the order of the IO computations would produce a different output).

Taking a slightly different approach, Chitil [Chitil 2000] and Ghani and Johann [Ghani and Johann 2008] give a shortcut fusion law that permits fusion of effectful functions without requiring commutativity of the monad. The law presented in [Ghani and Johann 2008] is related with the shortcut fusion law to be introduced next.

3.2. Fusion with pure functions

Now we focus our attention on a restricted form of compositions involving effects. Concretely, we will consider compositions between a monadic producer p and the lifting of a fold: $\mathbf{do}\ \{t \leftarrow p\ x; return\ (fold\ h\ t)\}$. These are compositions where the effect is produced by the first function and only propagated by the second one. We introduce a shortcut fusion law for this kind of monadic compositions by means of specific examples.

Lists Consider the following composition:

$$\begin{array}{ll} sumSeq & :: Num\ a \Rightarrow [IO\ a] \rightarrow IO\ a \\ sumSeq\ ms & = \mathbf{do}\ \{xs \leftarrow sequence\ ms; return\ (sum\ xs)\} \end{array}$$

$$\begin{array}{ll} sum & :: Num\ a \Rightarrow [a] \rightarrow a \\ sum\ [\,] & = 0 \\ sum\ (a : as) & = a + sum\ as \end{array}$$

For example, when applied to the list $[put\ 1, put\ 2, put\ 3]$, $sumSeq$ returns a computation that yields `6` as result and prints the string `"123"` in the standard output. A recursive definition can be derived for $sumSeq$:

$$\begin{aligned}
&\mathit{sumSeq}\ [\,] &&= \mathit{return}\ 0\\
&\mathit{sumSeq}\ (m : \mathit{ms}) &&= \mathbf{do}\ \{x \leftarrow m; y \leftarrow \mathit{sumSeq}\ \mathit{ms}; \mathit{return}\ (x + y)\}
\end{aligned}$$

In this case, we can observe that fusion simply performs the substitution of the intermediate list constructors by corresponding actions in function sum. That is, it is a substitution between purely functional objects; no effects are involved.

This transformation can be captured by a shortcut fusion law associated with fold where we have to reflect the fact that the producer may be an effectful function and that the consumer (a fold) must appear lifted.

Law 4 (fold/mbuild for lists)

$$\mathbf{do}\ \{\mathit{as} \leftarrow \mathit{mbuild}_L\ g; \mathit{return}\ (\mathit{fold}_L\ (\mathit{nil}, \mathit{cons})\ \mathit{as})\} = g\ (\mathit{nil}, \mathit{cons})$$

where

$$\begin{aligned}
&\mathit{mbuild}_L\quad :: \mathit{Monad}\ m \Rightarrow (\forall\ b\ .\ (b, a \to b \to b) \to m\ b) \to m\ [a]\\
&\mathit{mbuild}_L\ g = g\ ([\,], (:))
\end{aligned}$$

The recursive definition of sumSeq can then be obtained by first writing sum and $\mathit{sequence}$ in terms of fold and the monadic build, respectively, and then applying Law 4.

$$\mathit{sum} = \mathit{fold}_L\ (0, (+))$$

$$\begin{aligned}
&\mathit{sequence}\ \mathit{ms} = \mathit{mbuild}_L\ (\mathit{gseq}\ \mathit{ms})\\
&\quad \mathbf{where}\\
&\quad\quad \mathit{gseq}\ [\,]\ (n, c) \qquad\quad = \mathit{return}\ n\\
&\quad\quad \mathit{gseq}\ (m : \mathit{ms})\ (n, c) = \mathbf{do}\ \{x \leftarrow m; y \leftarrow \mathit{gseq}\ \mathit{ms}\ (n, c); \mathit{return}\ (c\ x\ y)\}
\end{aligned}$$

Parsing Shortcut fusion is well suited to be used in the context of monadic parsers [Hutton and Meijer 1998]. A parser usually returns an abstract syntax tree which is then consumed by another function that performs the semantic actions. Using shortcut fusion, we show how these two phases can be merged together. To illustrate this, we present a simple parser that recognizes natural numbers. We adopt the usual definition of the parser monad (see [Hutton and Meijer 1998] for more details):

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

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

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

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

$$\begin{aligned}
&(\oplus) \qquad\qquad\quad :: \mathit{Parser}\ a \to \mathit{Parser}\ a \to \mathit{Parser}\ a\\
&(P\ p) \oplus (P\ q) = P\ (\lambda \mathit{cs} \to \mathbf{case}\ p\ \mathit{cs} +\!\!+\ q\ \mathit{cs}\ \mathbf{of}
\end{aligned}$$

```
                              []       → []
                              (x : xs) → [x])

item :: Parser Char
item = P (λcs → case cs of
                  ""       → []
                  (c : cs) → [(c, cs)])
```

Alternatives are represented by a deterministic choice operator ($\oplus$), 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.

Suppose we want to parse a string formed by digits and return a list containing their integer conversion. For example, given the string `"123"` the parser returns the list `[1,2,3]`.

```
digits :: Parser [Int]
digits = do { d ← digit; ds ← digits; return (d : ds) } ⊕ return []

digit :: Parser Int
digit = do { c ← item; if isDigit c then return (ord c − ord '0') else pzero }

isDigit c = (c ⩾ '0') ∧ (c ⩽ '9')
```

We want to test whether the number represented by the list of digits is divisible by 3, but without computing the number itself. It is well known that a number is divisible by 3 if the sum of its digits is also divisible by 3.

```
sumDigits :: Parser Int
sumDigits = do { ds ← digits; return (sum ds) }

divby3 :: Parser Bool
divby3 = do { n ← sumDigits; return (n `mod` 3 == 0) }
```

Since *digits* can be written as a monadic build,

```
digits = mbuild_L gdig
  where gdig (nil, cons)
          = do { d ← digit; ds ← gdig (nil, cons); return (cons d ds) }
          ⊕ return nil
```

and *sum* is a fold, we can apply Law 4, obtaining the following monolithic definition:

```
sumDigits = do { d ← digit; y ← sumDigits; return (d + y) } ⊕ return 0
```

Arithmetic Expressions Let us now consider a parser for arithmetic expressions. The parser takes a string containing an arithmetic expression and returns an abstract syntax tree of type *Exp*. For example, given the string `"1+2+3"` the parser returns the term *Add* (*Num* 1) (*Add* (*Num* 2) (*Num* 3)).

$$
\begin{aligned}
&\mathit{expression} :: \mathit{Parser}\ \mathit{Exp}\\
&\mathit{expression} = \mathbf{do}\ \{\, n \leftarrow \mathit{number}; \mathit{plusop};\\
&\qquad\qquad\qquad e \leftarrow \mathit{expression}; \mathit{return}\ (\mathit{Add}\ (\mathit{Num}\ n)\ e)\}\\
&\qquad\qquad \oplus\ \mathbf{do}\ \{\, n \leftarrow \mathit{number}; \mathit{return}\ (\mathit{Num}\ n)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{number} :: \mathit{Parser}\ \mathit{Int}\\
&\mathit{number} = \mathbf{do}\ \{\, (n,p) \leftarrow \mathit{numpow10}; \mathit{return}\ n\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{numpow10} :: \mathit{Parser}\ (\mathit{Int}, \mathit{Int})\\
&\mathit{numpow10} = \mathbf{do}\ \{\, d \leftarrow \mathit{digit}; (n,p) \leftarrow \mathit{numpow10}; \mathit{return}\ (d * p + n, 10 * p)\}\\
&\qquad\qquad \oplus\ \mathit{return}\ (0,1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{plusop} :: \mathit{Parser}\ ()\\
&\mathit{plusop} = \mathbf{do}\ \{\, c \leftarrow \mathit{item}; \mathbf{if}\ c == \texttt{'+'}\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else}\ \mathit{pzero}\}
\end{aligned}
$$

Given an arithmetic expression, we want to evaluate it.

$$
\begin{aligned}
&\mathit{evalexp} :: \mathit{Parser}\ \mathit{Int}\\
&\mathit{evalexp} = \mathbf{do}\ \{\, e \leftarrow \mathit{expression}; \mathit{return}\ (\mathit{eval}\ e)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval} && :: \mathit{Exp} \rightarrow \mathit{Int}\\
&\mathit{eval}\ (\mathit{Num}\ n) && = n\\
&\mathit{eval}\ (\mathit{Add}\ e\ e') && = \mathit{eval}\ e + \mathit{eval}\ e'
\end{aligned}
$$

Function $\mathit{evalexp}$ generates an intermediate expression that we would like to eliminate with fusion. The monadic shortcut fusion law in this case is the following:

Law 5 (fold/mbuild for expressions)

$$\mathbf{do}\ \{\, e \leftarrow \mathit{mbuild}_E\ g; \mathit{return}\ (\mathit{fold}_E\ (\mathit{num}, \mathit{add})\ e)\} = g\ (\mathit{num}, \mathit{add})$$

where

$$
\begin{aligned}
&\mathit{mbuild}_E && :: \mathit{Monad}\ m \Rightarrow (\forall\ a\ .\ (\mathit{Int} \rightarrow a, a \rightarrow a \rightarrow a) \rightarrow m\ a) \rightarrow m\ \mathit{Exp}\\
&\mathit{mbuild}_E\ g = g\ (\mathit{Num}, \mathit{Add})
\end{aligned}
$$

Now, if we write eval and $\mathit{expression}$ in terms of fold and build, respectively:

$$\mathit{eval} = \mathit{fold}_E\ (\mathit{id}, (+))$$

$$
\begin{aligned}
&\mathit{expression} = \mathit{mbuild}_E\ \mathit{gexp}\\
&\quad \mathbf{where}\ \mathit{gexp}\ (\mathit{num}, \mathit{add})\\
&\qquad\qquad = \mathbf{do}\ \{\, n \leftarrow \mathit{number}; \mathit{plusop};\\
&\qquad\qquad\qquad e \leftarrow \mathit{gexp}\ (\mathit{num}, \mathit{add}); \mathit{return}\ (\mathit{add}\ (\mathit{num}\ n)\ e)\}\\
&\qquad\qquad \oplus\ \mathbf{do}\ \{\, n \leftarrow \mathit{number}; \mathit{return}\ (\mathit{num}\ n)\}
\end{aligned}
$$

we can apply shortcut fusion (Law 5) to $\mathit{evalexp}$, obtaining the following definition:

$$
\begin{aligned}
&\mathit{evalexp} = \mathbf{do}\ \{\, n \leftarrow \mathit{number}; \mathit{plusop}; z \leftarrow \mathit{evalexp}; \mathit{return}\ (n + z)\}\\
&\qquad\quad \oplus\ \mathbf{do}\ \{\, n \leftarrow \mathit{number}; \mathit{return}\ n\}
\end{aligned}
$$

4. Shortcut fusion, generically

In this section, we show that the instances of fold, build, and shortcut fusion presented in the previous sections correspond to generic definitions valid for a wide class of datatypes.

4.1. Data types

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

Semantically, recursive datatypes are understood as least fixed points of functors. That is, given a datatype declaration it is possible to derive a functor F such that the datatype is the least solution to the equation $\tau \cong F\tau$. We write μF to denote the type corresponding to the least solution. The isomorphism between μF and $F\ \mu F$ is provided by two strict functions $\mathit{in}_F :: F\ \mu F \to \mu F$ and $\mathit{out}_F :: \mu F \to F\ \mu F$, inverses of each other. Function in_F packs the constructors of the datatype while out_F the destructors (for more details see e.g [Abramsky and Jung 1994, Gibbons 2002]).

For example, for the datatype of expressions we can derive a functor E such that:

$$\mathbf{data}\ E\ a = \mathit{FNum}\ \mathit{Int} \mid \mathit{FAdd}\ a\ a$$

$$\begin{array}{ll}
\mathit{map}_E & :: (a \to b) \to E\ a \to E\ b \\
\mathit{map}_E\ f\ (\mathit{FNum}\ n) & = \mathit{FNum}\ n \\
\mathit{map}_E\ f\ (\mathit{FAdd}\ a\ a') & = \mathit{FAdd}\ (f\ a)\ (f\ a')
\end{array}$$

In this case, $\mu E = \mathit{Exp}$ and

$$\begin{array}{ll}
\mathit{in}_E & :: E\ \mathit{Exp} \to \mathit{Exp} \\
\mathit{in}_E\ (\mathit{FNum}\ n) & = \mathit{Num}\ n \\
\mathit{in}_E\ (\mathit{FAdd}\ e\ e') & = \mathit{Add}\ e\ e'
\end{array}$$

$$\begin{array}{ll}
\mathit{out}_E & :: \mathit{Exp} \to E\ \mathit{Exp} \\
\mathit{out}_E\ (\mathit{Num}\ n) & = \mathit{FNum}\ n \\
\mathit{out}_E\ (\mathit{Add}\ e\ e') & = \mathit{FAdd}\ e\ e'
\end{array}$$

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 paremeter. That is, $\mu(L\,a) = [\,a\,]$.

$$\mathbf{data}\ L\ a\ b = \mathit{FNil} \mid \mathit{FCons}\ a\ b$$

$$\begin{array}{ll}
\mathit{map}_L & :: (a \to c) \to (b \to d) \to L\ a\ b \to L\ c\ d \\
\mathit{map}_L\ f\ g\ \mathit{FNil} & = \mathit{FNil} \\
\mathit{map}_L\ f\ g\ (\mathit{FCons}\ a\ b) & = \mathit{FCons}\ (f\ a)\ (g\ b)
\end{array}$$

4.2. Fold

Let F be a functor that captures the structure of a datatype. Given a function $h :: F\ a \to a$, *fold* [Gibbons 2002] is defined as the least function $\mathit{fold}_F\ h :: \mu F \to a$ such that:

$$fold_F\ h \circ in_F = h \circ F\ (fold_F\ h)$$

A function $h :: F\ a \to a$ is called an F-*algebra*. For example, an algebra corresponding to the functor E is a function $h :: E\ a \to a$ of the form:

$$\begin{aligned} h\ (FNum\ n) &= num\ n \\ h\ (FAdd\ a\ a') &= add\ a\ a' \end{aligned}$$

with $num :: Int \to a$ and $add :: a \to a \to a$. In the specific instance of fold for the Exp datatype we wrote an algebra h simply as a pair (num, add). For the list datatype we did something similar, in the fold for lists we wrote an algebra $h :: L\ a\ b \to b$ as a pair $(nil, cons)$. The same can be applied to any other inductive datatype.

An F-*homomorphism* between two algebras $h :: F\ a \to a$ and $k :: F\ b \to b$ is a function $f :: a \to b$ between the carriers that commutes with the operations. This is specified by the condition $f \circ h = k \circ F\ f$. Notice that fold is a homomorphism between the algebras in_F and h.

4.3. Shortcut fusion

Given a functor F, we can define a corresponding build operator:

$$\begin{aligned} &build_F\ \ :: (\forall\ a\ .\ (F\ a \to a) \to a) \to \mu F \\ &build_F\ g = g\ in_F \end{aligned}$$

Together with $fold$, $build$ enjoys the following fusion law [Takano and Meijer 1995], which is an instance of a free theorem [Wadler 1989].

Law 6 (fold/build) *For strict h,*[2]

$$fold_F\ h\ (build_F\ g) = g\ h$$

4.4. Monadic shortcut fusion

The shortcut fusion law 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. To state that law it is necessary to introduce an extended form of build. Given a functor F (signature of a datatype) and another functor N, we define:

$$\begin{aligned} &build_{F,N} :: (\forall\ a\ .\ (F\ a \to a) \to N\ a) \to N\ \mu F \\ &build_{F,N}\ g = g\ in_F \end{aligned}$$

When N is a monad we obtain a monadic build,

$$\begin{aligned} &mbuild_F :: Monad\ m \Rightarrow (\forall\ a\ .\ (F\ a \to a) \to m\ a) \to m\ \mu F \\ &mbuild_F\ g = g\ in_F \end{aligned}$$

[2]The strictness condition on h was not mentioned in the concrete instances of the law shown in Section 2 because a function defined by pattern matching is strict. That was the case of the algebras for expressions and lists considered in those instances.

On the other hand, the standard $build_F$ is obtained by considering the identity functor.

Law 7 (extended fold/build) *For strict h and strictness preserving N,*

$$map_N\ (fold_F\ h)\ (build_{F,N}\ g) = g\ h$$

Proof The free theorem associated with g's type states that, for all types b and b', algebras $\varphi :: F\ b \to b$ and $\psi :: F\ b' \to b'$, and strict function $f :: b \to b'$, the following holds $f \circ \varphi = \psi \circ map_F\ f \Rightarrow map_N\ f\ (g\ \varphi) = g\ \psi$. By considering $f = fold_F\ h$, $\varphi = in_F$ and $\psi = h$, we get $map_N\ (fold_F\ h)\ (g\ in_F) = g\ h$, because, again, the premise of the implication holds by definition of fold. Finally, we apply the definition of $build_{F,N}$ to obtain the law. The strictness on h is necessary for instantiation: if the algebra h is strict, then so is $fold_F\ h$, and we can instantiate f with $fold_F\ h$. The strictness-preserving assumption on the functor means that map_N preserves strict functions, i.e., if f is strict, then so is $map_N\ f$. This condition is necessary for stating the free theorem itself, and therefore it is inherited by the instantiation. $\square$

Monadic shortcut fusion is then obtained from this law by considering the functor associated with a monad m and by unfolding the corresponding $mmap$ function:

Law 8 (fold/mbuild) *For strict h and strictness preserving $mmap$,*

$$\mathbf{do}\ \{t \leftarrow mbuild_F\ g; return\ (fold_F\ h\ t)\} = g\ h$$

5. Related work

In [Pardo 2001, Pardo 2005], fusion laws for monadic versions of some recursion schemes (fold, unfold and hylomorphism) are presented. It is simple to see that so-called *acid rain laws* (a kind of fusion laws) associated with monadic folds and hylomorphisms are particular cases of monadic shortcut fusion. This is something that should not be surprising if we take into account that corresponding laws for purely functional versions of the same operators can be expressed in terms of standard shortcut fusion [Takano and Meijer 1995]. Let us consider, for example, the generic definition of monadic fold:

$$\begin{aligned} &mfold_F :: Monad\ m \Rightarrow (F\ a \to m\ a) \to \mu F \to m\ a \\ &mfold_F\ h = h \bullet \widehat{F}\ (mfold_F\ h) \circ out_F \end{aligned}$$

where $(f \bullet g)\ x = \mathbf{do}\ \{y \leftarrow f\ x; g\ y\}$, for monadic functions f and g, and $\widehat{F}\ f = dist_F \circ map_F\ f$, for monadic function f, such that $dist_F :: F\ (m\ a) \to m\ (F\ a)$ distributives the functor over the monad (see e.g. [Pardo 2005]). Consider the following acid rain law associated with monadic fold: For $\tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to m\ a)$, strict h and strictness-preserving $mmap$,

$$\mathbf{do}\ \{t' \leftarrow mfold_G\ (\tau\ in_F)\ t; return\ (fold_F\ h\ t')\} = mfold_G\ (\tau\ h)\ t$$

If we define $gmfold\ t\ \varphi = mfold_G\ (\tau\ \varphi)\ t$, then $mfold_G\ (\tau\ in_F)\ t = gmfold\ t\ in_F = mbuild_F\ (gmfold\ t)$, and therefore the acid rain law reduces to monadic shortcut fusion. For the acid rain law associated with monadic hylomorphism the situation is the same.

Chitil's PhD thesis [Chitil 2000] presents a generalized shortcut fusion law for the list case that is able to fuse effectful functions. We recall that law by giving its generic definition. Let $q :: \forall\ a\ .\ (F\ a \to a) \to (a \to b) \to c$ and $h :: F\ b \to b$. Then,

$$q\ in_F\ (fold_F\ h) = q\ h\ id$$

To see an example, consider again the function $displaySeq_T$. If we define,

$$q\ (leaf, join)\ f = \lambda ms \to \mathbf{do}\ \{t \leftarrow gseq_T\ ms\ (leaf, join); f\ t\}$$

$$\begin{aligned}
&gseq_T\ (Leaf\ m)\ (leaf, join) &&= \mathbf{do}\ \{a \leftarrow m; return\ (leaf\ a)\}\\
&gseq_T\ (Join\ ml\ mr)\ (leaf, join) &&= \mathbf{do}\ \{l \leftarrow gseq_T\ ml\ (leaf, join);\\
& && \qquad\qquad r \leftarrow gseq_T\ mr\ (leaf, join); return\ (join\ l\ r)\}
\end{aligned}$$

$$display_T = fold_T\ (putStr \circ show, \lambda ml\ mr \to \mathbf{do}\ \{ml; mr\})$$

$$\begin{aligned}
&fold_T &&:: (a \to b, b \to b \to b) \to Tree\ a \to b\\
&fold_T\ (leaf, join)\ (Leaf\ a) &&= leaf\ a\\
&fold_T\ (leaf, join)\ (Join\ l\ r) &&= join\ (fold_T\ (leaf, join)\ l)\ (fold_T\ (leaf, join)\ r)
\end{aligned}$$

then,

$$\begin{aligned}
&displaySeq_T\ ms\\
&\quad = \mathbf{do}\ \{t \leftarrow seq_T\ ms; display_T\ t\}\\
&\quad = q\ (Leaf, Join)\ (display_T\ t)\ ms\\
&\quad = q\ (putStr \circ show, \lambda ml\ mr \to \mathbf{do}\ \{ml; mr\})\ id\ ms\\
&\quad = \mathbf{do}\ \{m \leftarrow gseq_T\ (putStr \circ show, \lambda ml\ mr \to \mathbf{do}\ \{ml; mr\})\ ms; m\}
\end{aligned}$$

Observe that the obtained expression is formed by a function that returns a computation that yields computations as result, such that, the outer computation produces the effects of the producer (seq_T), while the inner computations produce the effects of the consumer ($display_T$). By inlining $f = gseq_T\ (putStr \circ show, \lambda ml\ mr \to \mathbf{do}\ \{ml; mr\})$ we get a clear picture of the generated computation.

$$\begin{aligned}
&f\ (Leaf\ m) &&= \mathbf{do}\ \{a \leftarrow m; return\ (putStr\ (show\ a))\}\\
&f\ (Join\ ml\ mr) &&= \mathbf{do}\ \{ml' \leftarrow f\ ml; mr' \leftarrow f\ mr; return\ (\mathbf{do}\ \{ml'; mr'\})\}
\end{aligned}$$

At the same time to us, but independently, Ghani and Johann [Ghani and Johann 2008] presented a shortcut fusion law that is able to fuse compositions of effectful programs. Like our monadic shortcut fusion law, their fusion law is also based on extended shortcut fusion (Law 7). The crucial difference with ours is that they consider a fold with monadic carrier as consumer. The law is the following: For strict h,

$$\mathbf{do}\ \{t \leftarrow mbuild_F\ g; fold_F\ h\ t\} = \mathbf{do}\ \{m \leftarrow g\ h; m\}$$

The left-hand side of the expression can be rewritten as $\mathbf{do}\ \{t \leftarrow mbuild_F\ g; m \leftarrow return\ (fold_F\ h\ t); m\}$, which by Law 8 is transformed to the right-hand side. It is interesting to see that the monadic expression obtained with this fusion law is exactly the same as the one produced by Chitil's law. In fact, if we define $q\ h\ f = \mathbf{do}\ \{t \leftarrow g\ h; f\}$, this law reduces to Chitil's.

6. Conclusions

This paper presented a shortcut fusion law tailored to a restricted form of compositions of programs with effects. The monadic shortcut fusion law introduced is simple, generic, and easy to apply in practice.

We have used the rewrite rules mechanism (RULES pragma) of the Glasgow Haskell Compiler (GHC) to obtain a prototype implementation of monadic shortcut fusion. Experimental results measuring time and space improvements for a set of examples are available in the webpage `http://www.fing.edu.uy/~pardo/MonadicShortcut/`.

Acknowledgements We would like to thank the referees for their helpful comments and suggestions.

References

Abramsky, S. and Jung, A. (1994). Domain theory. In *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press.

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

Chitil, O. (2000). *Type-inference based deforestation of functional programs*. PhD thesis, RWTH Aachen.

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

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

Gill, A., Launchbury, J., and Jones, S. P. (1993). A Shortcut to Deforestation. In *Conference on Functional Programming and Computer Architecture*.

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

Johann, P. and Voigtländer, J. (2004). Free theorems in the presence of seq. In *31st Symposium on Principles of Programming Languages*, pages 99–110. ACM.

Manzino, C. (2005). Monadic Shortcut Deforestation. Final year project, National University of Rosario, Argentina.

Meijer, E. and Jeuring, J. (1995). Merging Monads and Folds for Functional Programming. In *Advanced Functional Programming*, LNCS 925, pages 228–266. Springer-Verlag.

Pardo, A. (2001). Fusion of Recursive Programs with Computational Effects. *Theoretical Computer Science*, 260:165–207.

Pardo, A. (2005). Combining Datatypes and Effects. In *Advanced Functional Programming*, LNCS 3622, pages 171–209. Springer-Verlag.

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

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

Wadler, P. (1995). Monads for functional programming. In *Advanced Functional Programming*, LNCS 925. Springer-Verlag.