

Algebraic Fusion of Functions with an Accumulating Parameter and Its Improvement

Shin-ya Katsumata

Research Institute for Mathematical Sciences
Kyoto University
Kyoto 606-8502, Japan
sinya@kurims.kyoto-u.ac.jp

Susumu Nishimura *

Department of Mathematics, Faculty of Science
Kyoto University
Kyoto 606-8502, Japan
susumu@math.kyoto-u.ac.jp

Abstract

We present a unifying solution to the problem of fusion of functions, where both the producer function and the consumer function have one accumulating parameter. The key idea in this development is to formulate the producer function as a function which computes over a monoid of *data contexts*. Upon this formulation, we develop a fusion method called *algebraic fusion* based on the elementary theory of universal algebra and monoids. The producer function is fused with a monoid homomorphism that is derived from the definition of the consumer function, and is turned into a higher-order function f that computes over the monoid of endofunctions.

We then introduce a general concept called *improvement*, in order to reduce the cost of computing over the monoid of endofunctions (i.e., function closures). An improvement of the function f via a monoid homomorphism h is a function g that satisfies $f = h \circ g$. This provides a principled way of finding a first-order function representing a solution to the fusion problem. It also presents a clean and unifying account for varying fusion methods that have been proposed so far. Furthermore, we show that our method extends to support partial and infinite data structures, by means of an appropriate monoid structure.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Algebraic approaches to semantics; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Program and recursion schemes

General Terms Languages, Theory

Keywords shortcut fusion; accumulating parameter; data contexts; monoids and monoid homomorphisms; higher-order removal; partial and infinite data structures.

* Partly supported by the Grant-in-Aid for Young Scientists (B), MEXT, Japan.

This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06), DOI: <http://doi.acm.org/10.1145/1159803.1159835> ICFP'06 September 16–21, 2006, Portland, Oregon, USA. Copyright © 2006 ACM 1-59593-309-3/06/0009...\$5.00.

1. Introduction

Modular programming is a commonly approved programming style that encourages us to solve a problem by combining solutions to subproblems. Functional programming languages exploit a higher level of modularity, in which any type of data can be a glue that composes functions together. While this adds an extra flexibility to the modular programming discipline, it can cause inefficiency because of possible additional cost in manipulating the intermediate data.

As a typical scenario, consider two list-processing functions $p, c : \text{list} \rightarrow \text{list}$ composed as $c \circ p : \text{list} \rightarrow \text{list}$. The functions p and c are called *producer* and *consumer*, respectively, as the function p produces an intermediate list that is consumed by c . Although the composed presentation certainly provides a modular solution, we would prefer a single monolithic function that does not produce the intermediate list, as this function exhibits improved behavior in the usage of heap memory.

The conflicting requirements of modularity and improved memory usage have led to the development of a family of program transformation techniques, called *fusion*, which automatically derive such single monolithic functions that perform the same computation as the composition $c \circ p$. Among those fusion transformation techniques, the so called *calculational* approaches have been the most successful ones. A calculational method improves programs by stepwise applications of a set of equational laws on a few combinators (e.g., generic recursion operators such as `foldr` for lists). The calculational methods are easier to implement, as the applicability of the laws can be judged locally without a global analysis of programs. Examples of such calculational methods include the promotion theorem [15], shortcut fusion [9] and its derivatives [8, 19, 11, 6].

These techniques, however, do not give a satisfactory result when applied to functions with an extra argument, called *accumulating parameter*, on which temporary data is accumulated during recursion. A relevant shortcut fusion law can be applied to functions with additional parameters, but its result is a higher-order function whose evaluation involves expensive operations on function closures for each recursion step. Recent research developments have seen several alternative fusion methods that generate first-order programs when they are applied to functions with an accumulating parameter, including fusion law for the `dmap` combinator [13], the lazy composition technique [20], and a higher-order removal method applied to the result of shortcut fusion [17].

In this paper, we propose a new fusion technique called *algebraic fusion*. The merits of algebraic fusion over these existing methods are addressed below.

- Algebraic fusion is based on the theory of monoids and universal algebra and provides a clean, universal account of the intricate business in dealing with accumulating parameter. The direct output of algebraic fusion is a higher-order program, which is presented as a monoid of function compositions. We will show that the existing methods mentioned above can be explained uniformly in terms of monoid homomorphisms: every particular fusion method can be recognized as a mapping, by means of an appropriate monoid homomorphism, between the monoid of function space and another monoid of first-order objects whose multiplication is efficiently implementable.
- Throughout the transformation process, all the transformation tasks are carried out in a calculational way. Algebraic fusion has only one local fusion law that is similar to shortcut fusion and its derivatives. This is in contrast to some precursors [20, 17], which handle accumulating parameters by non-local transformation rules that introduce a circular let construct in order to maintain cyclic dependency in the target first-order program. Putting this cyclicity construction in an appropriate monoid, we can derive the desired first-order program by a simple calculation on the monoid and monoid homomorphism. This helps greatly in understanding the essence behind the intricate transformation process of these precursors.
- Our fusion law is justified by a straightforward equational reasoning. It is also simple to give a justification on a particular transformation process in our framework: it is sufficient to find suitable monoids and monoid homomorphisms and to prove that the homomorphism condition is fulfilled. This remarkable simplicity gives rise to a uniform account of various fusion transformation tasks, including those which eliminate partial and infinite data structures.
- A significant observation in this paper is that it is crucial to keep an eye on the monoid structure of programs throughout the transformation process. As we shall discuss in Section 3.6, a variant of shortcut fusion produces the same higher-order program as algebraic fusion does. However, algebraic fusion preserves the monoid structure derived from the original program in an explicit form, which makes the subsequent improvement process readily applicable.

Despite of the fruitful merits described above, we note that algebraic fusion has some weaknesses as compared to its counterpart: the popular shortcut fusion method. One major weakness is that the producer must be a function that computes over a monoid. Thus algebraic fusion can only deal with a limited subclass of programs, while its counterpart shortcut fusion can deal with more. However, the aim of this paper is to provide a clean account of the intricate business in dealing with accumulating parameters and the class of functions we consider still includes a sufficiently large set of program instances that use accumulating parameters.

The rest of the paper is organized as follows: Section 2 informally introduces algebraic fusion and demonstrates how a result of algebraic fusion can be improved by a monoid homomorphism. In Section 3, we give a formal definition of algebraic fusion and shows its correctness. The relationship with shortcut fusion and its derivatives is also mentioned. In Section 4, we exhibit how the results of algebraic fusion can be improved for varying instances. Section 5 exploits a more sophisticated monoid supporting partial and infinite data structures and shows that the second author's previous work is another instance of improvement. We discuss related work in Section 6 and conclude in Section 7.

2. Algebraic Fusion by Example

Algebraic fusion is a method to fuse the following two functions called *producer* and *consumer*:

$$\text{prod} \in T_{\Delta} \rightarrow T_{\Sigma} \rightarrow T_{\Sigma} \quad \text{cons} \in T_{\Sigma} \rightarrow D,$$

where T_{Δ} and T_{Σ} are sets of (total and finite) Δ - and Σ -terms for some signatures Δ and Σ , respectively. The result of the algebraic fusion of *prod* and *cons* is the following function

$$\text{fuse} \in T_{\Delta} \rightarrow D \rightarrow D$$

satisfying

$$\text{fuse } t (\text{cons } u) = \text{cons } (\text{prod } t u).$$

The function *fuse* directly calculates values in D from Δ -terms without creating intermediate Σ -terms which are passed from *prod* to *cons* during the computation of the r.h.s. of the above equation.

We demonstrate how algebraic fusion and its improvement work by means of a small example, *rev* composed of itself, where *rev* is the iterative list reverse function defined by the following set of recursive equations:

$$\begin{aligned} \text{rev} & \in T_{\text{list}} \rightarrow T_{\text{list}} \rightarrow T_{\text{list}} \\ \text{rev } [] & x = x \\ \text{rev } (a :: l) & x = \text{rev } l (a :: x). \end{aligned}$$

The fusion process exhibited in this section is a particular instance of the general method, where $\Delta = \Sigma = \text{list}$, $\text{prod} = \text{cons} = \text{rev}$ and $D = T_{\text{list}} \rightarrow T_{\text{list}}$; here *list* is the signature $\{[]^0, a :: -^1\}$ of cons lists (a ranges over some set A). We assume that readers are familiar with the concept of monoid and monoid homomorphism. We also write *revrev* for the result *fuse* of algebraic fusion in the present particular example.

One of the key observations in algebraic fusion is to regard a producer with an accumulating parameter as a function *Prod* from T_{Δ} to the set of Σ -contexts. A Σ -context is a Σ -term that may contain *holes* denoted by $[-]$; more formally, it is simply a $\Sigma \cup \{[-]^0\}$ -term. For example, $1 :: (2 :: (3 :: [-]))$ is a list context.

Let us tentatively write C_{list} for the set of list contexts. Given two list contexts c, c' , we write $c \cdot c'$ for the list context obtained by filling a hole in c with c' . For instance, the hole filling $(1 :: (2 :: (3 :: [-]))) \cdot (4 :: (5 :: [-]))$ results in $1 :: (2 :: (3 :: (4 :: (5 :: [-])))$). A similar operation with $- \cdot -$ is the function $\text{fill} : C_{\text{list}} \rightarrow T_{\text{list}} \rightarrow T_{\text{list}}$, which takes a list context k and a list l and then returns a list obtained by filling a hole in k with l (if k has no hole, $\text{fill}(k)(l) = k$). The context-filling operation $- \cdot -$ and $[-]$ obey the axioms of monoids. Thus the triple $(C_{\text{list}}, [-], - \cdot -)$ forms the monoid of list contexts. Here we also introduce another monoid called *function space monoid* $A \Rightarrow A$ over some set A . It is a triple $(A \rightarrow A, \text{id}, - \circ -)$, where $- \circ -$ is the function composition.

The first step of algebraic fusion is to represent the producer side of *rev* by means of a function that calculates list contexts. We observe that the result of the application of *rev* to $[a_1, \dots, a_n]$ can be represented by a list context and fill:

$$\begin{aligned} \text{rev } [a_1, \dots, a_n] &= \lambda x. a_n :: (\dots :: (a_1 :: x) \dots) \\ &= \text{fill } (a_n :: (\dots :: (a_1 :: [-]))) \end{aligned}$$

where the underlined list context can be computed by the application of the following recursive function *Rev* to $[a_1, \dots, a_n]$:

$$\begin{aligned} \text{Rev} & \in T_{\text{list}} \rightarrow C_{\text{list}} \\ \text{Rev } [] &= [-] \\ \text{Rev } (a :: l) &= \text{Rev } l \cdot (a :: [-]). \end{aligned}$$

To summarize, *rev* can be represented by *Rev* and *fill*:

$$\text{rev} = \text{fill} \circ \text{Rev}.$$

The next step of algebraic fusion is to extend the domain of the consumer side of *rev* to C_{list} . This extension is done by adding to the

recursive definition of rev (i) an extra parameter w and (ii) an extra line that handles the case where an input is a hole. The following recursive function $\overline{\text{rev}}$ is the result of this extension.

$$\begin{aligned} \overline{\text{rev}} & \in C_{\text{list}} \rightarrow (T_{\text{list}} \rightarrow T_{\text{list}}) \rightarrow (T_{\text{list}} \rightarrow T_{\text{list}}) \\ \overline{\text{rev}} [] & w = \lambda x.x \\ \overline{\text{rev}} (a :: l) & w = \lambda x.\overline{\text{rev}} l w (a :: x) \\ \overline{\text{rev}} [-] & w = w \end{aligned}$$

The extra parameter is simply passed around in the recursive calls of $\overline{\text{rev}}$ and replaces every hole in the input. This extension satisfies the following property that is crucial to algebraic fusion:

$$\begin{aligned} \overline{\text{rev}} [-] &= \text{id} \\ \overline{\text{rev}} (k \cdot k') &= \overline{\text{rev}} k \circ \overline{\text{rev}} k' \end{aligned}$$

That is, $\overline{\text{rev}}$ is a *monoid homomorphism* from the monoid of list contexts to the function space monoid $(T_{\text{list}} \rightarrow T_{\text{list}}) \Rightarrow (T_{\text{list}} \rightarrow T_{\text{list}})$.

The final step of algebraic fusion is to extract a recursive function that performs the same computation as $\overline{\text{rev}} \circ \text{Rev}$. The computation of this composition involves the creation of intermediate data structures passed from Rev to $\overline{\text{rev}}$, but it can be eliminated due to the shape of Rev and the property of $\overline{\text{rev}}$ being a monoid homomorphism. To see this, we apply $\overline{\text{rev}}$ to both sides of the recursive definition of Rev , and obtain the following equations:

$$\begin{aligned} \overline{\text{rev}} (\text{Rev} []) &= \overline{\text{rev}} [-] \\ &= \text{id} \\ \overline{\text{rev}} (\text{Rev} (a :: l)) &= \overline{\text{rev}} (\text{Rev} l \cdot (a :: [-])) \\ &= (\overline{\text{rev}} (\text{Rev} l)) \circ (\overline{\text{rev}} (a :: [-])) \\ &= (\overline{\text{rev}} (\text{Rev} l)) \circ (\lambda w.x.w (a :: x)) \end{aligned}$$

By folding (in the sense of [2]) the function calls of the form $\overline{\text{rev}}(\text{Rev } -)$, we obtain a recursive function revrev :

$$\begin{aligned} \text{revrev} & \in T_{\text{list}} \rightarrow (T_{\text{list}} \rightarrow T_{\text{list}}) \rightarrow (T_{\text{list}} \rightarrow T_{\text{list}}) \\ \text{revrev} [] & w = w \\ \text{revrev} (a :: l) & w = \text{revrev} l (\lambda x.w (a :: x)). \end{aligned}$$

This is the result of algebraic fusion of rev and rev . The function revrev satisfies the equation:

$$\text{revrev} l (\text{rev } y) = \text{rev} (\text{rev} l y), \quad (1)$$

where the initial accumulating parameter y to the producer rev is preprocessed by the consumer before it is passed to the fused function revrev .

In the present fusion example, we can substitute the expensive computation over the function space monoid $(T_{\text{list}} \rightarrow T_{\text{list}}) \Rightarrow (T_{\text{list}} \rightarrow T_{\text{list}})$ with a less expensive one over another monoid as follows. Let $(T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$ be the *opposite* of the function space monoid $T_{\text{list}} \Rightarrow T_{\text{list}}$, i.e., the monoid whose multiplication $- \bullet -$ is defined by $f \bullet g = g \circ f$. It is easy to show that $h = \lambda g.\lambda f.(f \circ g)$ is a monoid homomorphism from $(T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$ to $(T_{\text{list}} \rightarrow T_{\text{list}}) \Rightarrow (T_{\text{list}} \rightarrow T_{\text{list}})$. Let us define a recursive function $\text{app} \in T_{\text{list}} \rightarrow T_{\text{list}} \rightarrow T_{\text{list}}$ as follows:

$$\begin{aligned} \text{app} [] &= \lambda w.w \\ \text{app} (a :: l) &= \text{app} l \bullet (\lambda w.a :: w) \\ &= \lambda w.a :: (\text{app} l w). \end{aligned}$$

Now we apply h to both sides of app . Since h is a monoid homomorphism, we obtain the following equations:

$$\begin{aligned} h (\text{app} []) &= h(\lambda w.w) = \text{id} \\ h (\text{app} (a :: l)) &= h ((\text{app} l) \bullet (\lambda w.a :: w)) \\ &= (h (\text{app} l)) \circ (\lambda w.x.w (a :: x)) \end{aligned}$$

From this, we conclude that the recursive function obtained by folding $h (\text{app } -)$ coincides with revrev . Therefore we have

$$h \circ \text{app} = \text{revrev}. \quad (2)$$

We call app the *improvement* of revrev . The above equation implies that the value of $\text{revrev}(x)$ can be calculated by $\text{app}(x)$, which does not involve any closure creations during its recursion. Furthermore, this equation gives a property that holds between rev and app :

$$\begin{aligned} \text{rev} (\text{rev } s t) u & \\ &= \text{revrev } s (\text{rev } t) u \quad \text{by (1)} \\ &= h (\text{app } s) (\text{rev } t) u \quad \text{by (2)} \\ &= \text{rev } t (\text{app } s u) \quad \text{by definition of } h. \end{aligned}$$

Here we note that the above equational derivation is valid only if the input lists are total (that is, no divergent computation is contained) and finite. If we allow partial and infinite lists as inputs, the last equation should be replaced by an inequation

$$\text{rev} (\text{rev } s t) u \sqsubseteq \text{rev } t (\text{app } s u),$$

which implies that the computation of the r.h.s. may terminate while that of the l.h.s. diverges. Our theory correctly captures this subtle point appearing in algebraic fusion and improvement. We will return to this topic in Section 5.

The discussion so far suggests that there is a uniform way of replacing expensive computation over function space with other domains of moderate computation cost: Find a suitable monoid \mathcal{M} , whose multiplication has a less expensive computation cost than that of the function space monoid, and a recursive function Q such that there exists a monoid homomorphism h from \mathcal{M} to the function space monoid that makes $h \circ Q$ coincide with the fused function. Then $h \circ Q$ may give a more efficient implementation.

In the following sections, we formulate the aforementioned ideas by means of the elementary theory of universal algebra and monoids, and demonstrate that our approach can uniformly represent several existing fusion methods and program transformations.

3. Algebraic Fusion

We carry out the development of algebraic fusion for programs represented by set-theoretic functions. The correctness of the fusion will be proved as an equality about set-theoretic functions.

3.1 Notations

We use Σ, Δ for ranging over single-sorted first-order signatures. By $o \in \Sigma^{(n)}$ we mean that o is an n -ary operator in Σ . We write T_{Σ} for the set of closed Σ -terms. The initial Σ -algebra homomorphism (a.k.a. catamorphism) with respect to a Σ -algebra (D, δ) will be denoted by $\llbracket \delta \rrbracket \in T_{\Sigma} \rightarrow D$. We fix a finite set A (ranged over by a) for the elements of lists. The followings are pre-defined signatures:

$$\text{nat} = \{Z^0, S^1\} \quad \text{tree} = \{L^0, N^2\} \quad \text{list} = \{[]^0, a :: ^1\}.$$

We write \mathcal{M}, \mathcal{N} to range over monoids.

3.2 Σ -contexts

The object which plays a central role in algebraic fusion is the monoid of Σ -contexts, which we introduce below.

For a signature Σ , by Σ^+ we mean the signature extended with a nullary constant $[-]^0$ denoting a hole. We call Σ^+ -terms Σ -contexts in this paper, and write C_{Σ} for the set of Σ -contexts instead of T_{Σ^+} .

We can think of a Σ -context k as a function $\lambda t.k[t] \in T_{\Sigma} \rightarrow T_{\Sigma}$, where $k[t]$ is the Σ -term obtained by filling all holes in k with t . We define $\text{fill}_{\Sigma} \in C_{\Sigma} \rightarrow T_{\Sigma} \rightarrow T_{\Sigma}$ as the mapping from a Σ -context k to the function $\lambda t.k[t]$. The subscript of fill may be omitted when it is clear from the context.

A natural operation over Σ -contexts is to fill all holes in a Σ -context k with some other Σ -context k' . We write $k \cdot k'$ for this context-filling operation. This operation and the hole $[-]$ obey the axioms of monoid, i.e., $- \cdot -$ is associative and $[-]$ is the unit of $- \cdot -$. We thus obtain the monoid $C_\Sigma = (C_\Sigma, [-], - \cdot -)$ of Σ -contexts.

Another monoid that we often refer to is the *function space monoid*. For any set A , the identity function $\text{id} : A \rightarrow A$ and the composition $f \circ g$ of functions $f, g \in A \rightarrow A$ obey the axioms of monoids. We write $A \Rightarrow A$ for the triple $(A \rightarrow A, \text{id}, - \circ -)$ and call it the *function space monoid*.

It is easy to see that fill_Σ is a monoid homomorphism from C_Σ to $T_\Sigma \Rightarrow T_\Sigma$.

3.3 Polynomial Σ -Algebras over Monoids

The concept of polynomial Σ -algebras over monoids is the technical vehicle in this paper. Intuitively, it is a special Σ -algebra such that the carrier is a monoid and the algebra structure is given by polynomials over the monoid. Polynomial Σ -algebras have favorable properties for reasoning and transforming recursive programs, as we demonstrated below.

Let $\mathcal{M} = (M, e, \star)$ be a monoid, where M , e , and \star stand for the carrier set, the unit and the multiplication, respectively. A *polynomial P* over \mathcal{M} is a formal expression

$$P[X_1, \dots, X_n] = c_1 \star X_{i_1} \star c_2 \star X_{i_2} \dots \star X_{i_l} \star c_{l+1}$$

where n, l are natural numbers, $c_1, \dots, c_{l+1} \in M$ are called *coefficients*, and $1 \leq i_1, \dots, i_l \leq n$ are indexes of the formal parameter variables. We omit writing the unit coefficients unless they appear solely.

Example 3.1 Examples of polynomials are:

$$\begin{aligned} \text{Rev}_{a::[-]}[X] &= X \cdot (a :: [-]) && (\text{over } C_{\text{list}}) \\ \text{Count}_{N(-,-)}[X_1, X_2] &= S [-] \cdot X_1 \cdot X_2 && (\text{over } C_{\text{nat}}) \end{aligned}$$

There are a few basic facts about polynomials over monoids. Let P be an n -variable polynomial over $\mathcal{M} = (M, e, \star)$ and $h : \mathcal{M} \rightarrow \mathcal{N}$ be a monoid homomorphism. Then,

1. P determines a function $\text{fun}(P) : M^n \rightarrow M$ given by
$$\text{fun}(P)(x_1, \dots, x_n) = P[x_1/X_1, \dots, x_n/X_n],$$
2. by mapping each coefficient of P with h , we obtain a polynomial $h(P)$ over \mathcal{N} , and
3. since h is a homomorphism, we have
$$\text{fun}(h(P))(h m_1, \dots, h m_n) = h \circ \text{fun}(P)(m_1, \dots, m_n).$$

Example 3.2 (continued from Example 3.1) Recall that fill_Σ is a monoid homomorphism from C_Σ to $T_\Sigma \Rightarrow T_\Sigma$. By mapping polynomials $\text{Rev}_{a::[-]}$ and $\text{Count}_{N(-,-)}$ with $\text{fill}_{\text{list}}$ and fill_{nat} respectively, we obtain two polynomials:

$$\begin{aligned} \text{fill}_{\text{list}}(\text{Rev}_{a::[-]}[X]) &= X \circ (\lambda x. a :: x) && (\text{over } T_{\text{list}} \Rightarrow T_{\text{list}}) \\ \text{fill}_{\text{nat}}(\text{Count}_{N(-,-)}[X_1, X_2]) &= (\lambda x. S x) \circ X_1 \circ X_2 && (\text{over } T_{\text{nat}} \Rightarrow T_{\text{nat}}) \end{aligned}$$

We introduce the concept of *polynomial Σ -algebras*.

Definition 3.3 A polynomial Σ -algebra P over a monoid $\mathcal{M} = (M, e, \star)$ is an operator-indexed family of polynomials $\{P_o\}_{o \in \Sigma}$ such that for each $o \in \Sigma^{(n)}$, P_o is an n -variable polynomial over \mathcal{M} . A polynomial Σ -algebra P over \mathcal{M} induces a Σ -algebra $(M, [\text{fun}(P_o)]_{o \in \Sigma})$, which we refer to by P as well.

We note that the concept of polynomial algebras can be defined for any algebraic objects which admit the concept of polynomials, such as monoids, groups, rings, etc.

Example 3.4 (continued from Example 3.2) An example of a polynomial list-algebra over C_{list} is the family

$$\text{Rev} = \{\text{Rev}_[] = [-], \text{Rev}_{a::[-]}[X] = X \cdot (a :: [-])\}.$$

This polynomial algebra induces the initial list-algebra homomorphism $\llbracket \text{Rev} \rrbracket \in T_{\text{list}} \rightarrow C_{\text{list}}$, which has the following recursive definition:

$$\begin{aligned} \llbracket \text{Rev} \rrbracket [] &= [-] \\ \llbracket \text{Rev} \rrbracket (a :: l) &= \llbracket \text{Rev} \rrbracket l \cdot (a :: [-]). \end{aligned}$$

The function $\llbracket \text{Rev} \rrbracket$ maps a list $[a_1, \dots, a_n]$ to a list context $a_n :: \dots :: a_1 :: [-]$.

Another example of a polynomial tree-algebra over C_{nat} is the family

$$\begin{aligned} \text{Count} &= \{\text{Count}_L = S [-], \\ &\text{Count}_{N(-,-)}[X_1, X_2] = S [-] \cdot X_1 \cdot X_2\}. \end{aligned}$$

The initial tree-algebra homomorphism $\llbracket \text{Count} \rrbracket \in T_{\text{tree}} \rightarrow C_{\text{nat}}$ has the following recursive definition:

$$\begin{aligned} \llbracket \text{Count} \rrbracket L &= S [-] \\ \llbracket \text{Count} \rrbracket N(l, r) &= S [-] \cdot \llbracket \text{Count} \rrbracket l \cdot \llbracket \text{Count} \rrbracket r. \end{aligned}$$

Definition 3.5 Let $h : \mathcal{M} \rightarrow \mathcal{N}$ be a monoid homomorphism and P be a polynomial Σ -algebra over \mathcal{M} . The image of P by h is the following polynomial Σ -algebra $h(P)$ over \mathcal{N} defined by

$$h(P) = \{h(P_o)\}_{o \in \Sigma}.$$

Proposition 3.6 For any monoid homomorphism $h : \mathcal{M} \rightarrow \mathcal{N}$ and polynomial Σ -algebra P over \mathcal{M} , h is a Σ -algebra homomorphism from P to $h(P)$, and we have

$$h \circ \llbracket P \rrbracket = \llbracket h(P) \rrbracket.$$

3.4 Conditions for Producers and Consumers

Recall that algebraic fusion is a method to fuse the following two functions called *producer* and *consumer*:

$$\text{prod} \in T_\Delta \rightarrow T_\Sigma \rightarrow T_\Sigma \quad \text{cons} \in T_\Sigma \rightarrow D.$$

However, we can not apply algebraic fusion to arbitrary combinations of a producer and a consumer. In this section, we introduce two conditions, one for the producers and the other for the consumers.

The producer prod should satisfy the following condition:

(C-prod-s) For each Δ -term $o(t_1, \dots, t_n)$, the recursive definition of the value $\text{prod}(o(t_1, \dots, t_n)) \in T_\Sigma \rightarrow T_\Sigma$ can be expressed as follows:

$$\text{prod}(o(t_1, \dots, t_n)) = (\text{fill } k_1) \circ (\text{prod } t_{i_1}) \cdots (\text{prod } t_{i_l}) \circ (\text{fill } k_{l+1})$$

where $1 \leq i_1, \dots, i_l \leq n$ are indexes of subterms and $k_1, \dots, k_{l+1} \in C_\Sigma$ are Σ -contexts representing accumulation of information. When $k_i = [-]$ for some $1 \leq i \leq l+1$, we omit writing $\text{fill } k_i$.

This condition precisely identifies what we usually refer to as a producer function with an accumulating parameter. The producer function accumulates information on the second argument but does not inspect into it during recursion; furthermore, the information is accumulated in a sequential manner.

Example 3.7 The function rev satisfies (C-prod-s); the value of $\text{rev} []$ and $\text{rev}(a :: l)$ can be expressed with function compositions,

list-contexts, and fill:

$$\begin{aligned}\text{rev } [] &= \lambda x.x = \text{fill } [-] \\ \text{rev } (a :: l) &= \lambda x.\text{rev } l (a :: x) \\ &= (\text{rev } l) \circ (\lambda x.a :: x) \\ &= (\text{rev } l) \circ \text{fill } (a :: [-]).\end{aligned}$$

Another example of a function satisfying (C-prod-s) is $\text{count} : T_{\text{tree}} \rightarrow T_{\text{nat}} \rightarrow T_{\text{nat}}$ which adds the number of leaves and nodes of a tree given in the first argument to the second argument:

$$\begin{aligned}\text{count } L \quad w &= S \ w \\ \text{count } (N(l, r)) \quad w &= S \ (\text{count } l \ (\text{count } r \ w)).\end{aligned}$$

To see that count satisfies the condition (C-prod-s), we transform the r.h.s. of the above definition into:

$$\begin{aligned}\text{count } L &= \lambda w.S \ w \\ &= \text{fill } (S \ [-]) \\ \text{count } (N(l, r)) &= \lambda w.S \ (\text{count } l \ (\text{count } r \ w)) \\ &= (\text{fill } (S \ [-])) \circ (\text{count } l) \circ (\text{count } r).\end{aligned}$$

On the other hand, the following function rp , which replaces the rightmost leaf of a tree in the first argument with the second argument, does not satisfy (C-prod-s):

$$\begin{aligned}\text{rp } L \quad w &= w \\ \text{rp } (N(l, r)) \quad w &= N(\text{rp } l \ L, \text{rp } r \ w),\end{aligned}$$

since it cannot be defined as the interleaving composition of functions and recursive calls.

For the sake of simplicity in the following development of algebraic fusion, we use the following concise condition (C-prod) which is equivalent to (C-prod-s):

(C-prod) There exists a polynomial Δ -algebra Prod over C_{Σ} such that $\text{prod} = \text{fill} \circ \llbracket \text{Prod} \rrbracket$.

Example 3.8 (continued from Example 3.4 and 3.7) The functions rev and count satisfy (C-prod); polynomial algebras Rev and Count satisfy

$$\begin{aligned}\text{rev} &= \text{fill} \circ \llbracket \text{Rev} \rrbracket \\ \text{count} &= \text{fill} \circ \llbracket \text{Count} \rrbracket.\end{aligned}$$

Each consumer cons should be a recursive function which can be written as an initial algebra morphism:

(C-cons) $\text{cons} = \llbracket \delta \rrbracket$ for some Σ -algebra (D, δ) .

In other words, cons should be a recursive function whose defining equation for each $o \in \Delta^{(n)}$ is

$$\text{cons } (o(t_1, \dots, t_n)) = \delta_o(\text{cons } t_1, \dots, \text{cons } t_n),$$

where $\delta_o \in D^n \rightarrow D$ is the component at o of the algebra structure δ over D .

3.5 Algebraic Fusion

We are now ready to introduce algebraic fusion. Let $\text{prod} \in T_{\Delta} \rightarrow T_{\Sigma} \rightarrow T_{\Sigma}$ be a producer satisfying (C-prod) and $\text{cons} \in T_{\Sigma} \rightarrow D$ a consumer satisfying (C-cons).

We first extend the domain of cons from T_{Σ} to C_{Σ} . This is done by adding two things to the recursive definition of cons : (i) an extra parameter w , and (ii) a line which handles the case where an input is a hole. This extension yields the following recursive function $\overline{\text{cons}} \in C_{\Sigma} \rightarrow D \rightarrow D$:

$$\begin{aligned}\overline{\text{cons}} \ [-] \quad w &= w \\ \overline{\text{cons}} \ (o(k_1, \dots, k_n)) \quad w &= \delta_o(\overline{\text{cons}} \ k_1 \ w, \dots, \overline{\text{cons}} \ k_n \ w).\end{aligned}$$

The extra parameter w is distributed to each recursive call of $\overline{\text{cons}}$ with a subterm (when the arity of o is 0, w is discarded), and is returned only when an input is a hole. The extension $\overline{\text{cons}}$ satisfies the following two properties which are indispensable to algebraic fusion.

Proposition 3.9 For any $\text{cons} \in T_{\Sigma} \rightarrow D$ satisfying (C-cons),

1. the function $\overline{\text{cons}} \in C_{\Sigma} \rightarrow D \rightarrow D$ constructed as above is a monoid homomorphism from C_{Σ} to $D \Rightarrow D$, and
2. for any $k \in C_{\Sigma}$ and $t \in T_{\Delta}$, we have $\overline{\text{cons}} \ k \ (\text{cons } t) = \text{cons} \ (\text{fill } k \ t)$.

Next, we take the image (see Definition 3.5) of the polynomial Δ -algebra Prod mentioned in (C-prod) by $\overline{\text{cons}}$. The image is a polynomial Δ -algebra $\overline{\text{cons}}(\text{Prod})$ over $D \Rightarrow D$. Then, the *result* of the algebraic fusion of prod and cons is defined by the initial Δ -algebra homomorphism

$$\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket : T_{\Delta} \rightarrow D \rightarrow D.$$

The following theorem shows that algebraic fusion is correct.

Theorem 3.10 For any $t, u \in T_{\Delta}$, we have

$$\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket \ t \ (\text{cons } u) = \text{cons} \ (\text{prod } t \ u).$$

Proof Let $t, u \in T_{\Delta}$. Then,

$$\begin{aligned}\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket \ t \ (\text{cons } u) &= \overline{\text{cons}} \ (\llbracket \text{Prod} \rrbracket \ t) \ (\text{cons } u) && \text{by Proposition 3.6} \\ &= \text{cons} \ ((\text{fill} \circ \llbracket \text{Prod} \rrbracket) \ t \ u) && \text{by Proposition 3.9-2} \\ &= \text{cons} \ (\text{prod } t \ u) && \text{by (C-prod)}.\end{aligned}$$

□

Example 3.11 We have already checked that rev satisfies (C-prod) in Example 3.8 (and equivalently (C-prod-s) in Example 3.7). It is also easy to check that rev satisfies (C-cons).

We thus proceed to apply algebraic fusion of rev with itself. We first extend rev to a monoid homomorphism $\overline{\text{rev}} : C_{\text{list}} \rightarrow (T_{\text{list}} \rightarrow T_{\text{list}}) \Rightarrow (T_{\text{list}} \rightarrow T_{\text{list}})$.

$$\begin{aligned}\overline{\text{rev}} \ [] \quad f \quad x &= x \\ \overline{\text{rev}} \ (a :: l) \quad f \quad x &= \overline{\text{rev}} \ l \ f \ (a :: x) \\ \overline{\text{rev}} \ [-] \quad f \quad x &= f \ x\end{aligned}$$

We then take the image of the polynomial list-algebra Rev over C_{list} by $\overline{\text{rev}}$, and obtain the following polynomial list-algebra $\overline{\text{rev}}(\text{Rev})$ over $(T_{\text{list}} \rightarrow T_{\text{list}}) \Rightarrow (T_{\text{list}} \rightarrow T_{\text{list}})$:

$$\begin{aligned}\overline{\text{rev}}(\text{Rev}_{[]} &= \text{id} \\ \overline{\text{rev}}(\text{Rev}_{a::-})[X] &= X \circ (\lambda f w.f \ (a :: w)).\end{aligned}$$

This polynomial algebra induces an initial list-algebra homomorphism $\llbracket \overline{\text{rev}}(\text{Rev}) \rrbracket \in T_{\text{list}} \rightarrow (T_{\text{list}} \rightarrow T_{\text{list}}) \rightarrow (T_{\text{list}} \rightarrow T_{\text{list}})$, which is the result of the algebraic fusion of rev and rev . We write this result as revrev for short. The recursive definition of revrev is

$$\begin{aligned}\text{revrev } [] &= \text{id} \\ \text{revrev } (a :: l) &= (\text{revrev } l) \circ (\lambda f w.f \ (a :: w)),\end{aligned}$$

and from Theorem 3.10, revrev satisfies

$$\text{revrev } t \ (\text{rev } u) \ s = \text{rev} \ (\text{rev } t \ u) \ s. \quad (3)$$

3.6 Relationship with Shortcut Fusion

In this section, we informally compare algebraic fusion and shortcut fusion [9]. Here, we consider the case where intermediate data structures passed from producers to consumers are `tree`-terms, and we let τ be a type, $k : \tau \rightarrow \tau \rightarrow \tau$ and $z : \tau$. We write $\text{cata} : \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{tree} \rightarrow \alpha$ for the polymorphic catamorphism constructor for `tree`-terms (here, we identify the signature `tree` and the algebraic data type corresponding to `tree`).

We consider the following minor extension of the shortcut fusion for `tree`-terms using the combinator $\text{build}' : (\forall \alpha. (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \text{tree} \rightarrow \text{tree}$ defined by

$$\text{build}' \ x \ y = x \ \text{tree} \ N \ L \ y.$$

As an application of Reynolds' parametricity principle [14, 21], we have

$$\text{cata} \ \tau \ k \ z \ (\text{build}' \ x \ h) = g \ \tau \ k \ z \ (\text{cata} \ \tau \ k \ z \ h).$$

From this, we obtain $\text{build}'/\text{cata}$ fusion: for any producer $\text{prod} : \rho \rightarrow \text{tree} \rightarrow \text{tree}$ and consumer $\text{cons} : \text{tree} \rightarrow \tau$ satisfying

(B-prod) $\exists g. \text{prod} = \text{build}' \circ g$ and

(B-cons) $\text{cons} = \text{cata} \ \tau \ k \ z$

respectively, we have

$$\text{cons} \ (\text{prod} \ x \ h) = g \ x \ \tau \ k \ z \ (\text{cons} \ h).$$

There is a close correspondence between this extension and algebraic fusion. We observe that the parametricity principle entails that the type $\forall \alpha. (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$, which appears in the type of build' , corresponds to the carrier of the initial `tree`⁺-algebra; in other words, it is the type of `tree`-context. Under this correspondence, build' performs the same computation as `fill`. From this observation, we notice the similarity between $\text{build}'/\text{cata}$ fusion and algebraic fusion.

The difference between $\text{build}'/\text{cata}$ fusion and algebraic fusion is that condition (B-prod) is much weaker than condition (C-prod), i.e., $\text{build}'/\text{cata}$ fusion accepts more producers than algebraic fusion (e.g., see function `rp` in Example 3.7). In algebraic fusion, producers are supposed to perform primitive recursion over Σ -terms and calculate values in the way given by polynomial Σ -algebras. On the other hand, $\text{build}'/\text{cata}$ -fusion has no such constraints on producers; the domain of producers can be of any type.

The major source of this subtle difference stems from the technical foundation on which each fusion transformation is built. Algebraic fusion is formulated in the world of sets and functions using the universal property of initial algebras (Proposition 3.6), while $\text{build}'/\text{cata}$ -fusion is formulated in a second-order logic for a polymorphic programming language with the parametricity principle.

Both $\text{build}'/\text{cata}$ fusion and algebraic fusion are driven by essentially the same fusion law. Therefore, algebraic fusion can be understood as a restriction of $\text{build}'/\text{cata}$ -fusion. However, there is a merit in considering fusion of a restricted class of producers. The program structure of the producer is preserved by algebraic fusion in an explicit form, which makes the subsequent manipulation process easier. In the next section, we propose the concept called *improvement*, which is useful for reasoning and transforming results of algebraic fusion.

4. Improving Algebraic Fusion

The function `revrev` obtained in Example 3.11 calculates the return values using function composition, which is the multiplication of the function space monoid over $T_{\text{list}} \rightarrow T_{\text{list}}$. However, this is an expensive operation in the implementation of functional languages, because it involves the creation of closures. In general, a similar

problem arises for any consumer $\text{cons} : T_{\Sigma} \rightarrow D$ when the multiplication of the function space monoid $D \Rightarrow D$ requires expensive computation.

Our strategy to avoid this problem is to calculate temporary return values in another monoid \mathcal{M} , whose multiplication is efficiently implementable, and then recover the original return values with an efficiently implementable monoid homomorphism. Suppose we find

- a monoid \mathcal{M} ,
- a monoid homomorphism $h : \mathcal{M} \rightarrow (D \Rightarrow D)$, and
- a polynomial Σ -algebra P over \mathcal{M} such that $h(P) = \overline{\text{cons}}(\text{Prod})$.

Then the function $\llbracket P \rrbracket \in T_{\Delta} \rightarrow \mathcal{M}$ is expected to be more efficiently implementable than $\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket$, and from Proposition 3.6, we have

$$h \ (\llbracket P \rrbracket \ t) = \llbracket h(P) \rrbracket \ t = \llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket \ t.$$

We call $\llbracket P \rrbracket$ an *improvement* of the result of algebraic fusion of `prod` and `cons` in \mathcal{M} (via h).

Example 4.1 (Continued from Example 3.11) We re-introduce the example of an improvement of the result `revrev` of the algebraic fusion of `rev` and `rev` in Section 2. We improve `revrev` with the following parameter:

Monoid We take the opposite monoid $(T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$ of $T_{\text{list}} \Rightarrow T_{\text{list}}$, that is, the monoid $(T_{\text{list}} \rightarrow T_{\text{list}}, \text{id}_{T_{\text{list}}}, \bullet)$ where the multiplication \bullet is defined by $f \bullet g = g \circ f$.

Monoid Homomorphism We take $h = \lambda f. \lambda g. g \circ f$. This is a monoid homomorphism since

$$(h \ f \circ \ h \ f') \ g = g \circ \ f' \circ \ f = g \circ (f \bullet f') = h \ (f \bullet f') \ g.$$

Polynomial Algebra We take the following polynomial list-algebra P over $(T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$:

$$P_{[]} = \text{id}$$

$$P_{a::-}[X] = X \bullet (\lambda w. a :: w).$$

This satisfies $h(P) = \overline{\text{rev}}(\text{Rev})$ since the unique coefficient in P is mapped to the one in $\overline{\text{rev}}(\text{Rev})$, i.e., $h \ (\lambda w. a :: w) = \lambda f w. f \ (a :: w)$.

These data give the improvement $\llbracket P \rrbracket$ of `revrev` in $(T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$, and they satisfy:

$$\text{revrev} = \llbracket h(P) \rrbracket = h \circ \llbracket P \rrbracket. \quad (4)$$

We notice that the recursive definition of $\llbracket P \rrbracket$, with the second argument being explicit, coincides with that of the list concatenation function `app`:

$$\begin{aligned} \llbracket P \rrbracket \ [] &= x = x \\ \llbracket P \rrbracket \ (a :: l) \ x &= ((\llbracket P \rrbracket \ l) \bullet (\lambda x. a :: x)) \ x \\ &= a :: ((\llbracket P \rrbracket \ l) \ x). \end{aligned}$$

Therefore we simply write `app` for $\llbracket P \rrbracket$ below. From Theorem 3.10, we obtain a law about `rev` and `app`:

$$\begin{aligned} \text{rev} \ (\text{rev} \ s \ t) \ u & \\ = \text{revrev} \ s \ (\text{rev} \ t) \ u & \quad \text{by (3)} \\ = h \ (\text{app} \ s) \ (\text{rev} \ t) \ u & \quad \text{by (4)} \\ = \text{rev} \ t \ (\text{app} \ s \ u) & \quad \text{by definition of } h. \end{aligned}$$

4.1 Algebraic Fusion of Takehi et al.'s `Dmap` with Itself

We apply algebraic fusion and its improvement for deriving the fusion law of Takehi et al.'s `dmap` (in this paper we shorten the name to `dm`). This is a generic combinator for representing list-manipulating functions with an accumulating parameter [13]. For

functions $f, g \in A \rightarrow A^1$, dm_f^g is recursively defined by

$$\begin{aligned} \text{dm}_f^g \ [] &= \lambda y. y \\ \text{dm}_f^g (a :: l) &= \lambda y. (f a) :: (\text{dm}_f^g l ((g a) :: y)). \end{aligned}$$

They showed that dm satisfies the following fusion law:

$$\text{dm}_{f'}^{g'} (\text{dm}_f^g l l') = (\text{dm}_{f' \circ f}^{g' \circ g} l) \circ (\text{dm}_{f'}^{g'} l') \circ (\text{dm}_f^{g' \circ f} l).$$

for any $f, g, f', g' \in A \rightarrow A$ by induction. We demonstrate that we can also derive this law using algebraic fusion and improvement. This derivation does not use explicit induction over l or l' .

Algebraic Fusion of dmap and dmap We first apply algebraic fusion to dm_f^g as a producer and $\text{dm}_{f'}^{g'}$ as a consumer. It is easy to check that $\text{dm}_{f'}^{g'}$ satisfies (C-cons). To see that dm_f^g satisfies (C-prod), we define a polynomial list-algebra Dm_f^g over C_{list} :

$$\begin{aligned} (\text{Dm}_f^g)_{[]} &= [-] \\ (\text{Dm}_f^g)_{a::-}[X] &= ((f a) :: [-]) \cdot X \cdot ((g a) :: [-]). \end{aligned}$$

This gives the witness of dm_f^g satisfying (C-prod), i.e.,

$$\text{fill} \circ \llbracket \text{Dm}_f^g \rrbracket = \text{dm}_f^g.$$

We therefore proceed to apply algebraic fusion. We extend $\text{dm}_{f'}^{g'}$ to a monoid homomorphism $\overline{\text{dm}}_{f'}^{g'} : C_{\text{list}} \rightarrow (\text{list} \rightarrow \text{list}) \Rightarrow (\text{list} \rightarrow \text{list})$:

$$\begin{aligned} \overline{\text{dm}}_{f'}^{g'} \ [] & w = \lambda x. x \\ \overline{\text{dm}}_{f'}^{g'} (a :: l) & w = \lambda x. (f' a) :: (\overline{\text{dm}}_{f'}^{g'} l w ((g' a) :: x)) \\ \overline{\text{dm}}_{f'}^{g'} [-] & w = w, \end{aligned}$$

and then calculate the polynomial list-algebra $\overline{\text{dm}}_{f'}^{g'}(\text{Dm}_f^g)$ over $(\text{list} \rightarrow \text{list}) \Rightarrow (\text{list} \rightarrow \text{list})$ as follows:

$$\begin{aligned} \overline{\text{dm}}_{f'}^{g'}((\text{Dm}_f^g)_{[]}) &= \text{id} \\ \overline{\text{dm}}_{f'}^{g'}((\text{Dm}_f^g)_{a::-}[X]) &= \alpha(f', g', f a) \circ X \circ \alpha(f', g', g a) \end{aligned}$$

where coefficients of the form $\alpha(f, g, a)$ is given by

$$\alpha(f, g, a) = \overline{\text{dm}}_f^g(a :: [-]) = \lambda w x. (f a) :: (w ((g a) :: x)).$$

Thus $\llbracket \overline{\text{dm}}_{f'}^{g'}(\text{Dm}_f^g) \rrbracket \in (\text{list} \rightarrow (\text{list} \rightarrow \text{list})) \rightarrow (\text{list} \rightarrow \text{list})$ is the result of the algebraic fusion of dm_f^g and $\text{dm}_{f'}^{g'}$, but we do not display its recursive definition here.

Improvement We improve the above result of the algebraic fusion with the following data.

Monoid We take the product monoid $(\text{list} \Rightarrow \text{list}) \times (\text{list} \Rightarrow \text{list})^{\text{op}}$ whose multiplication will be denoted by \star . Explicitly,

$$(f, g) \star (f', g') = (f \circ f', g \bullet g') = (f \circ f', g' \circ g).$$

We write π_1, π_2 for the first and second projections from this product monoid.

Monoid Homomorphism We take the function $h \in (\text{list} \rightarrow \text{list}) \times (\text{list} \rightarrow \text{list}) \rightarrow (\text{list} \rightarrow \text{list}) \rightarrow (\text{list} \rightarrow \text{list})$ defined by

$$h(p, q) = \lambda w. (p \circ w \circ q).$$

This is indeed a monoid homomorphism from $(\text{list} \Rightarrow \text{list}) \times (\text{list} \Rightarrow \text{list})^{\text{op}}$ to $(\text{list} \rightarrow \text{list}) \rightarrow (\text{list} \rightarrow \text{list})$.

Polynomial Algebra The coefficients of the form $a(f, g, a)$ in Dm_f^g can be given by h and the following element $A(f, g, a) \in (\text{list} \rightarrow \text{list}) \times (\text{list} \rightarrow \text{list})$ in the product monoid:

$$A(f, g, a) = (\lambda x. (f a) :: x, \lambda x. (g a) :: x),$$

i.e., $h(A(f, g, a)) = \alpha(f, g, a)$. Therefore the following polynomial list-algebra DM over $(\text{list} \Rightarrow \text{list}) \times (\text{list} \Rightarrow \text{list})^{\text{op}}$:

$$\text{DM}_{[]} = (\text{id}, \text{id})$$

$$\text{DM}_{a::-}[X] = A(f', g', f a) \star X \star A(f', g', g a)$$

satisfies

$$\overline{\text{dm}}_{f'}^{g'}(\text{Dm}_f^g) = h(\text{DM}).$$

From this, we obtain an improvement $\llbracket \text{DM} \rrbracket \in (\text{list} \rightarrow (\text{list} \rightarrow \text{list})) \times (\text{list} \rightarrow \text{list})$ of the result of algebraic fusion of dm_f^g and $\text{dm}_{f'}^{g'}$, and it satisfies

$$\llbracket \overline{\text{dm}}_{f'}^{g'}(\text{Dm}_f^g) \rrbracket = \llbracket h(\text{DM}) \rrbracket = h \circ \llbracket \text{DM} \rrbracket. \quad (5)$$

Decomposition of the Improvement The images of DM by π_1 and π_2 induce initial algebra homomorphisms $\llbracket \pi_1(\text{DM}) \rrbracket$ and $\llbracket \pi_2(\text{DM}) \rrbracket$, which have the following recursive definitions:

$$\begin{aligned} \llbracket \pi_1(\text{DM}) \rrbracket \ [] & w = w \\ \llbracket \pi_1(\text{DM}) \rrbracket (a :: l) & w = (f' (f a)) :: (\llbracket \pi_1(\text{DM}) \rrbracket l ((f' (g a)) :: w)) \\ \llbracket \pi_2(\text{DM}) \rrbracket \ [] & w = w \\ \llbracket \pi_2(\text{DM}) \rrbracket (a :: l) & w = (g' (g a)) :: (\llbracket \pi_2(\text{DM}) \rrbracket l ((g' (f a)) :: w)). \end{aligned}$$

By comparing the above definitions with that of dm , we notice that

$$\llbracket \pi_1(\text{DM}) \rrbracket = \text{dm}_{f' \circ f}^{f' \circ g} \quad \llbracket \pi_2(\text{DM}) \rrbracket = \text{dm}_{g' \circ g}^{g' \circ f}.$$

Furthermore, for any $l \in T_{\text{list}}$, we have

$$\begin{aligned} (\text{dm}_{f' \circ f}^{f' \circ g} l, \text{dm}_{g' \circ g}^{g' \circ f} l) & \\ = (\llbracket \pi_1(\text{DM}) \rrbracket l, \llbracket \pi_2(\text{DM}) \rrbracket l) & \\ = (\pi_1 \circ \llbracket (\text{DM}) \rrbracket l, \pi_2 \circ \llbracket (\text{DM}) \rrbracket l) & \text{ by Proposition 3.9} \\ = \llbracket \text{DM} \rrbracket l. & \end{aligned} \quad (6)$$

From this, we derive the law of dmap :

$$\begin{aligned} \text{dm}_{f'}^{g'} (\text{dm}_f^g l l') & \\ = \llbracket \overline{\text{dm}}_{f'}^{g'}(\text{Dm}_f^g) \rrbracket l (\text{dm}_{f'}^{g'} l') & \text{ by Theorem 3.10} \\ = h(\llbracket \text{DM} \rrbracket l) (\text{dm}_{f'}^{g'} l') & \text{ by (5)} \\ = h(\text{dm}_{f' \circ f}^{f' \circ g} l, \text{dm}_{g' \circ g}^{g' \circ f} l) (\text{dm}_{f'}^{g'} l') & \text{ by (6)} \\ = (\text{dm}_{f' \circ f}^{f' \circ g} l) \circ (\text{dm}_{f'}^{g'} l') \circ (\text{dm}_{g' \circ g}^{g' \circ f} l) & \text{ by definition of } h. \end{aligned}$$

4.2 Canonical Improvement of Algebraic Fusion

We must find three parameters for an improvement of a results of algebraic fusion: a monoid, a monoid homomorphism, and a polynomial algebra structure. It is not known if a (non-trivial) improvement exists for an arbitrary combination of producers and consumers. However, we show below that if the consumer is an initial algebra homomorphism induced by a polynomial algebra over an algebraic object, then we can always find an improvement in a canonical way.

Let \mathcal{D} be an algebraic object admitting the concept of polynomials, such as monoid, group, ring, etc., and D denote the carrier set of \mathcal{D} . We consider an algebraic fusion of a producer $\text{prod} \in T_{\Delta} \rightarrow T_{\Sigma} \rightarrow T_{\Sigma}$ satisfying (C-prod) and a consumer $\text{cons} \in T_{\Sigma} \rightarrow D$ satisfying the following condition that is stronger than (C-cons):

(C-cons-p) There exists a polynomial Σ -algebra CONS over \mathcal{D} such that $\text{cons} = \llbracket \text{CONS} \rrbracket$.

¹ In this article, the range of f, g is fixed to A because we only consider the list of elements in A . In general, f, g can be any function in $A \rightarrow B$, and the discussion in this section is not affected by this general situation.

Lemma 4.2 Any function $f : T_\Delta \rightarrow T_\Sigma \rightarrow T_\Sigma$ satisfying (C-prod) also satisfies (C-cons-p).

Theorem 4.3 If prod satisfies (C-prod) and cons satisfies (C-cons-p), then the result of the algebraic fusion of prod and cons can be improved with the following data:

Monoid We take the monoid $\mathcal{D}^H = (D[H], H, *)$, where $D[H]$ is the set of one-variable polynomials over \mathcal{D} , and the multiplication $P * Q$ is defined by the substitution of a polynomial Q to H in P :

$$P * Q = P[Q/H].$$

Monoid Homomorphism We take the function $\text{fn} : D[H] \rightarrow (D \rightarrow D)$, which casts polynomials to functions from D to D . It is easy to see that this is a monoid homomorphism.

Polynomial Algebra We first define a monoid homomorphism $\text{CONS} : C_\Sigma \rightarrow \mathcal{D}^H$:

$$\begin{aligned} \overline{\text{CONS}}([-]) &= H \\ \overline{\text{CONS}}(o(k_1, \dots, k_n)) &= \text{CONS}_o[\overline{\text{CONS}}(k_1)/X_1, \dots, \overline{\text{CONS}}(k_n)/X_n]. \end{aligned}$$

This function satisfies $\overline{\text{CONS}} = \text{fn} \circ \overline{\text{CONS}}$. We then take the polynomial Δ -algebra $\overline{\text{CONS}}(\text{Prod})$ over \mathcal{D}^H for improvement, where Prod is the polynomial Δ -algebra mentioned in (C-prod). This polynomial algebra satisfies

$$\text{fn}(\overline{\text{CONS}}(\text{Prod})) = \overline{\text{CONS}}(\text{Prod}).$$

This improvement should be understood as a theoretical existence rather than a practical improvement of efficiency, because the implementation of $- * -$ is not efficient in general. However, in some cases, the polynomial Δ -algebra $\overline{\text{CONS}}(\text{Prod})$ can be restricted to a submonoid of \mathcal{D}^H , which may have an efficient implementation. In fact, the parameters chosen in the previous examples of improvements are all equivalent to the canonical improvements restricted to appropriate submonoids.

Example 4.4 (continued from Example 3.11) From Lemma 4.2, rev satisfies both (C-prod) and (C-cons-p). Thus from Theorem 4.3, revrev can be improved in $(\text{list} \Rightarrow \text{list})^H$ via fn . We see the detail of this canonical improvement, and relate this to the improvement in Example 4.1.

We first define a polynomial list-algebra REV over $\text{list} \Rightarrow \text{list}$ as follows:

$$\begin{aligned} \text{REV}_{[]} &= \text{id} \\ \text{REV}_{a:-}[X] &= X \circ (\lambda x.a :: x). \end{aligned}$$

The monoid homomorphism $\overline{\text{REV}}$ sends Rev to the following polynomial list-algebra over $(\text{list} \Rightarrow \text{list})^H$:

$$\begin{aligned} \overline{\text{REV}}(\text{REV}_{[]}) &= H \\ \overline{\text{REV}}(\text{REV}_{a:-}[X]) &= X * (H \circ (\lambda x.a :: x)) \end{aligned}$$

which satisfies $\text{fn}(\overline{\text{REV}}(\text{Rev})) = \overline{\text{rev}}(\text{Rev})$.

Now we consider the following subset of one-variable monoid polynomials over $T_{\text{list}} \Rightarrow T_{\text{list}}$:

$$\{H \circ f \mid f \in T_{\text{list}} \rightarrow T_{\text{list}}\} \subseteq (T_{\text{list}} \rightarrow T_{\text{list}})[H].$$

This subset includes the unit $H = H \circ \text{id}$ of $(T_{\text{list}} \Rightarrow T_{\text{list}})^H$, and is closed under $- * -$ since

$$(H \circ f) * (H \circ g) = (H \circ g) \circ f = H \circ (g \circ f).$$

Therefore, this subset specifies a submonoid $(T_{\text{list}} \Rightarrow T_{\text{list}})_1^H$ of $(T_{\text{list}} \Rightarrow T_{\text{list}})^H$.

We notice that this submonoid is isomorphic to $(T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$; the following function $i \in (T_{\text{list}} \rightarrow T_{\text{list}})[H] \rightarrow T_{\text{list}} \rightarrow T_{\text{list}}$ is a bijective monoid homomorphism between $(T_{\text{list}} \Rightarrow T_{\text{list}})_1^H$ and $(T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$.

$$i(H \circ f) = f \quad (i^{-1} f = H \circ f)$$

Furthermore, the polynomial list-algebra P used for improving revrev in Example 4.1 is the image of $\overline{\text{REV}}(\text{Rev})$ by i :

$$i(\overline{\text{REV}}(\text{Rev})) = P.$$

Example 4.5 From Lemma 4.2, dm_f^g satisfies both (C-prod) and (C-cons-p) for any $f, g \in A \rightarrow A$. Therefore from Theorem 4.3, the result of algebraic fusion of dm_f^g and $\text{dm}_{f'}^{g'}$ can always be improved in $(T_{\text{list}} \Rightarrow T_{\text{list}})^H$ via fn . The polynomial list-algebra Q giving the improvement is canonically determined as follows:

$$\begin{aligned} Q_{[]} &= H \\ Q_{a:-} &= \mathcal{A}(f', g', f a) * X * \mathcal{A}(f', g', g a) \end{aligned}$$

where $\mathcal{A}(f, g, a) = ((\lambda x.f a :: x) \circ H \circ (\lambda x.g a :: x))$.

The polynomial algebra Q can be restricted to the submonoid of $(T_{\text{list}} \Rightarrow T_{\text{list}})^H$ specified by the following subset:

$$\{f \circ H \circ g \mid f, g \in T_{\text{list}} \rightarrow T_{\text{list}}\} \subseteq (T_{\text{list}} \rightarrow T_{\text{list}})[H].$$

This submonoid is isomorphic to $(T_{\text{list}} \Rightarrow T_{\text{list}}) \times (T_{\text{list}} \Rightarrow T_{\text{list}})^{\text{op}}$, which is exactly the monoid used for improving algebraic fusion of dm and itself in Section 4.1. Furthermore, Q and the polynomial list-algebra Dm_f^g in Section 4.1 are equal via this isomorphism.

5. Algebraic Fusion for Partial and Infinite Data Structures

To accommodate the development in the previous sections with partial (data structures which may contain divergent computation) and infinite data structures, we replace sets and functions with ω -complete pointed partial orders (CPO for short) and continuous functions. For CPOs D, E , by $[D \rightarrow E]$ (resp. $[D \rightarrow_{\perp} E]$) we mean the CPO of (resp. strict) continuous functions. The concept of continuous Σ -algebras is fairly standard; see for example [10].

Definition 5.1 A continuous Σ -algebra \mathcal{D} is a pair (D, d) of a CPO D and an operator-indexed family of continuous functions $\{d_o\}_{o \in \Sigma}$ such that $d_o \in [D^n \rightarrow D]$ for each $o \in \Sigma^{(n)}$. A (resp. strict) continuous Σ -algebra homomorphism $f : (D, d) \rightarrow (E, e)$ is a (resp. strict) continuous function $f \in [D \rightarrow E]$ satisfying $f \circ d_o = e_o \circ f^n$ for each $o \in \Sigma^{(n)}$.

It is well-known that we can construct an initial object $\mathcal{T}_{\Sigma}^{\infty} = (T_{\Sigma}^{\infty}, \text{in}^{\infty})$ in the category of continuous Σ -algebras and strict continuous Σ -algebra homomorphisms (see for example [10]). This construction yields a CPO T_{Σ}^{∞} consisting of partial and infinite Σ -terms (including total ones). For example, T_{nat}^{∞} is the CPO of lazy natural numbers:

$$\{\perp, Z, S \perp, S Z, S(S \perp), S(S Z), \dots, \infty\}$$

whose partial order expresses the amount of information. Below we assume $T_{\Sigma} \subseteq T_{\Sigma}^{\infty}$ without loss of generality. In this section, we identify the operators in Σ with the continuous term constructors over T_{Σ}^{∞} . For each continuous Σ -algebra $\mathcal{D} = (D, d)$, we write $\llbracket d \rrbracket : \mathcal{T}_{\Sigma}^{\infty} \rightarrow \mathcal{D}$ for the unique strict continuous Σ -algebra homomorphism.

The universal property of the initial object asserts that for each strict continuous Σ -algebra homomorphism $h : (D, d) \rightarrow (E, e)$, we have $h \circ \llbracket d \rrbracket = \llbracket e \rrbracket$. This equality, often referred to as the *promotion theorem* [15], is the heart of program transformation. However, it

should be weakened to an inequality when we allow h to be a (not necessarily strict) continuous Σ -algebra homomorphism.

Proposition 5.2 *Let $h : (D, d) \rightarrow (E, e)$ be a continuous Σ -algebra homomorphism. Then,*

1. $\llbracket e \rrbracket \sqsubseteq h \circ \llbracket d \rrbracket$,
2. for any $t \in T_\Sigma$, we have $\llbracket e \rrbracket t = h \circ \llbracket d \rrbracket t$, and
3. $\llbracket e \rrbracket = h \circ \llbracket d \rrbracket$ if and only if h is strict.

Next, we introduce the concept of continuous monoids, which are simply monoid objects in the category of CPOs and continuous functions.

Definition 5.3 *A continuous monoid is a monoid $\mathcal{D} = (D, e, \star)$ where D is a CPO and the multiplication is a continuous function $-\star- \in [D \times D \rightarrow D]$. A (resp. strict) continuous monoid homomorphism $h : (D, e, \star) \rightarrow (E, \epsilon, *)$ is a (resp. strict) continuous function $h \in [D \rightarrow E]$ satisfying the laws of monoid homomorphisms.*

For a CPO D , by $[D \Rightarrow D]$ we mean the continuous monoid $([D \rightarrow D], \text{id}_D, - \circ -)$ of the continuous endofunctions over D .

The definition of monoid polynomials and polynomial Σ -algebras are the same as in Section 3. A polynomial Σ -algebra Q over a continuous monoid \mathcal{D} now determines a continuous Σ -algebra, since each n -variable polynomial over \mathcal{D} determines an n -ary continuous function. A (resp. strict) continuous monoid homomorphism $h : \mathcal{D} \rightarrow \mathcal{E}$ is then a (resp. strict) continuous Σ -algebra homomorphism from Q to $h(Q)$.

We write C_Σ^∞ for the carrier CPO of the continuous Σ^+ -algebra $\mathcal{T}_{\Sigma^+}^\infty$. The context-filling operator $-\cdot-\in [(C_\Sigma^\infty)^2 \rightarrow C_\Sigma^\infty]$ is continuous, and the triple $(C_\Sigma^\infty, [-], -\cdot-)$ forms the continuous monoid C_Σ^∞ of Σ -contexts. The action of filling a Σ -context with a Σ -term is a strict continuous monoid homomorphism $\text{fill}_\Sigma^\infty : C_\Sigma^\infty \rightarrow [T_\Sigma^\infty \Rightarrow T_\Sigma^\infty]$. The subscript of $\text{fill}_\Sigma^\infty$ may be omitted when it is clear from the context.

We introduce the algebraic fusion for partial and infinite data structures. Let $\text{prod} \in [T_\Delta^\infty \rightarrow [T_\Sigma^\infty \rightarrow T_\Sigma^\infty]]$ and $\text{cons} \in [T_\Sigma^\infty \rightarrow D]$ be continuous functions satisfying:

(C-prod') There exists a polynomial Δ -algebra Prod over C_Σ^∞ such that $\text{prod} = \text{fill}_\Sigma^\infty \circ \llbracket \text{Prod} \rrbracket$ (hence, prod should be strict).

(C-cons') There exists a continuous Σ -algebra (D, δ) such that $\text{cons} = \llbracket \delta \rrbracket$ (hence, cons should be strict).

Likewise in the algebraic fusion for total and finite data structures, we first extend the domain of the consumer function to C_Σ^∞ . This extension yields a strict continuous monoid homomorphism $\overline{\text{cons}} : C_\Sigma^\infty \rightarrow [D \Rightarrow D]$ satisfying $\overline{\text{cons}} k (\text{cons } t) = \text{cons} (\text{fill } k t)$ for any $k \in C_\Sigma^\infty$ and $t \in T_\Delta^\infty$ (c.f. Proposition 3.9).

We then take the image of Prod with $\overline{\text{cons}}$ and obtain a polynomial Δ -algebra $\overline{\text{cons}}(\text{Prod})$ over $[D \Rightarrow D]$. We call $\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket \in [T_\Delta^\infty \rightarrow_\perp [D \rightarrow D]]$ the result of algebraic fusion of prod and cons .

Theorem 5.4 *For any $x \in T_\Delta^\infty$ and $y \in T_\Sigma^\infty$, we have*

$$\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket x (\text{cons } y) = \text{cons} (\text{prod } x y).$$

Proof Since $\overline{\text{cons}}$ is strict, we have $\overline{\text{cons}} \circ \llbracket \text{Prod} \rrbracket = \llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket$ by Proposition 5.2-3. Therefore, we can prove this theorem in the same way as the equational reasoning in the proof of Theorem 3.10. \square

5.1 Improvement for Infinite Data Structures

The concept of improvement is affected by the transition from the world of sets and functions to that of CPOs and continuous functions. Suppose we find a continuous monoid \mathcal{M} , a monoid polynomial P over \mathcal{M} and a continuous monoid homomorphism $h : \mathcal{M} \rightarrow [D \Rightarrow D]$ such that $h(P) = \overline{\text{cons}}(\text{Prod})$. From Proposition 5.2, we have

1. $\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket = \llbracket h(P) \rrbracket \sqsubseteq h \circ \llbracket P \rrbracket$,
2. for any $t \in T_\Sigma$, we have $\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket t = h \circ \llbracket P \rrbracket t$, and
3. $\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket = h \circ \llbracket P \rrbracket$ if and only if h is strict.

Unlike the improvement in Section 4, we have the equality $\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket = h \circ \llbracket P \rrbracket$ if and only if h is strict; in general, we merely have the inequality $\llbracket \overline{\text{cons}}(\text{Prod}) \rrbracket \sqsubseteq h \circ \llbracket P \rrbracket$, which means that $h \circ \llbracket P \rrbracket$ is more likely to terminate than the result of the algebraic fusion. To examine this phenomenon in detail, we revisit the example of algebraic fusion of rev and rev .

Example 5.5 In the continuous setting, the iterative reverse function rev in Section 2 is interpreted as a continuous function $\text{rev}^\infty \in [T_{\text{list}}^\infty \rightarrow_\perp [T_{\text{list}}^\infty \rightarrow T_{\text{list}}^\infty]]$, which satisfies (C-prod') and (C-cons'). By applying algebraic fusion to rev^∞ and rev^∞ in the same way as Example 3.11, we obtain $\text{revrev}^\infty \in [T_{\text{list}}^\infty \rightarrow_\perp [[T_{\text{list}}^\infty \rightarrow T_{\text{list}}^\infty] \rightarrow [T_{\text{list}}^\infty \rightarrow T_{\text{list}}^\infty]]$, which satisfies

$$\text{revrev}^\infty t (\text{rev}^\infty u) = \text{rev}^\infty (\text{rev}^\infty t u)$$

by Theorem 5.4.

Similar to Example 4.1, we can improve revrev^∞ with a continuous monoid $[T_{\text{list}}^\infty \Rightarrow T_{\text{list}}^{\text{op}}]$, a continuous monoid homomorphism $h^\infty : [T_{\text{list}}^\infty \Rightarrow T_{\text{list}}^{\text{op}}] \rightarrow [[T_{\text{list}}^\infty \rightarrow T_{\text{list}}^\infty] \Rightarrow [T_{\text{list}}^\infty \rightarrow T_{\text{list}}^\infty]]$ mapping f to $\lambda g.g \circ f$, and a polynomial list-algebra P^∞ over $[T_{\text{list}}^\infty \Rightarrow T_{\text{list}}^{\text{op}}]$, which is simply the continuous version of P in Example 4.1.

However, here, a subtlety about termination behavior slips in: the continuous monoid homomorphism h^∞ is not strict. Thus, in general, we have the following inequality:

$$\text{revrev}^\infty t \sqsubseteq h^\infty \circ \llbracket P^\infty \rrbracket t$$

and both sides coincide only for $t \in T_{\text{list}}$. As we have seen in Example 4.1, $\llbracket P^\infty \rrbracket$ coincides with the continuous list-concatenation function app^∞ . From Theorem 5.4, we obtain:

$$\begin{aligned} \text{rev}^\infty (\text{rev}^\infty t s) u &= \text{revrev}^\infty t (\text{rev}^\infty s) u \\ &\sqsubseteq h^\infty (\text{app}^\infty t) (\text{rev}^\infty s) u \\ &= \text{rev}^\infty s (\text{app}^\infty t u), \end{aligned}$$

and the inequality becomes an equality for any $t \in T_{\text{list}}$. This indicates that the improvement does not have the same termination behavior as revrev^∞ for partial and infinite lists. This correctly captures the actual differences between $\text{rev} (\text{rev } t s) u$ and $\text{rev } s (\text{app } t u)$ in call-by-name languages with lazy lists.

5.2 A Semantic Higher-Order Removal

As we pointed out, when both a producer and a consumer have an accumulating parameter, their algebraic fusion (hence, shortcut fusion) yields a higher-order function. This is the motivation for introducing the concept of improvement in Section 4. This problem has already been recognized by Nishimura in [16, 17], and he introduced a program transformation technique called *higher-order removal* to reduce the order of computation. His transformation process is designed for a call-by-name language, and it takes a result of shortcut fusion:

$$f : \tau \rightarrow (\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma)$$

satisfying certain syntactic conditions. f is then transformed to the following function:

$$f' : \tau \rightarrow (\sigma \times \sigma) \rightarrow (\sigma \times \sigma)$$

which essentially performs the same computation as f .

Here, we give a similar program transformation in a simple and clean way using the concept of improvement with appropriate monoids and monoid homomorphisms. The following continuous monoids $D^\triangleright, D^\infty$ and two strict continuous monoid homomorphisms α, β play a crucial role in the semantic representation of higher-order removal:

$$D^\triangleright \xrightarrow{\alpha} D^\infty \xrightarrow{\beta} [[D \rightarrow D] \Rightarrow [D \rightarrow D]].$$

1. The first continuous monoid is $D^\triangleright = ([D \rightarrow D] \times [D^2 \rightarrow D], (\text{id}, \pi_2), @)$, where the multiplication $(f, g)@(f', g')$ is defined by

$$(f, g)@(f', g') = (f' \circ f, \lambda(x, y).g(x, g'((f x), y))).$$

We can draw an element of this monoid as a circuit diagram consisting of two processing boxes (Figure 1). The lower box has an input terminal on the left and an output terminal on the right, while the upper box has an output terminal on the left and two input terminals, one on the right and the other connected to the input of the lower box.

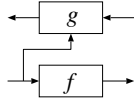


Figure 1. Diagram for $(f, g) \in D^\triangleright$

The multiplication of such circuits is simply done by juxtaposition (Figure 2).

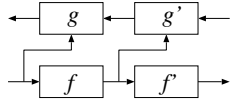


Figure 2. Diagram for $(f, g)@(f', g')$

2. The second continuous monoid is $D^\infty = ([D^2 \rightarrow D^2], \text{id}, \circ)$ whose multiplication $f \circ g$ is defined to be the following function:

$$\lambda(x, y).\text{let } ((-, q), (r, -)) = Y(\lambda((p, -), (-, s)).(f(x, s), g(p, y))) \text{ in } (r, q)$$

where $Y \in [[(D^2)^2 \rightarrow (D^2)^2] \rightarrow (D^2)^2]$ is the least fixed point operator. If we are allowed to use recursive let-expressions, the above function has a more compact definition:

$$\lambda(x, y).\text{letrec } (p, q) = f(x, s) \text{ and } (r, s) = g(p, y) \text{ in } (r, q).$$

We can draw an element of this monoid as a processing box with two input terminals on the left and two output terminals on the right. The multiplication of two elements in this monoid can be drawn as a circuit connecting two boxes in the following way:

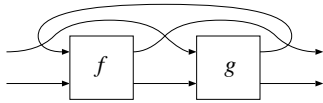


Figure 3. Diagram for $f \circ g$

The lower output of the left box is connected to the lower input of the right box, while the upper output of the right box is feedbacked to the upper input of the left box.²

3. Two strict continuous monoid homomorphisms $\alpha : D^\triangleright \rightarrow D^\infty$ and $\beta : D^\infty \rightarrow [[D \rightarrow D] \Rightarrow [D \rightarrow D]]$ are defined by

$$\alpha(f, g) = \lambda(x, y).(f x, g(x, y))$$

$$\beta f = \lambda w x.\text{letrec } (p, q) = f(x, w p) \text{ in } q.$$

The behavior of α is simply to flip over g in Figure 1 then regard the entire circuit as a processing box with two inputs on the left and two outputs on the right.

The behavior of β is to construct from $f \in [D^2 \rightarrow D^2]$ an action which maps one-input one-output processing box w drawn on the left in Figure 4 to a one-input one-output circuit described on the right in Figure 4.

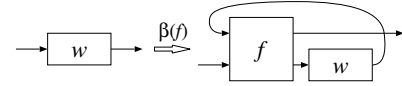


Figure 4. Behavior of $\beta(f)$

We note that the composition $\beta \circ \alpha$ behaves as follows:

$$\beta \circ \alpha(f, g) = \lambda w x.g(x, w(f x)).$$

Definition 5.6 Let $f \in [T_\Sigma^\infty \rightarrow_\perp [[D \rightarrow D] \rightarrow [D \rightarrow D]]]$ be a result of algebraic fusion such that f can be improved with the monoid D^\triangleright , the strict monoid homomorphism $\beta \circ \alpha$, and a polynomial Σ -algebra P over D^\triangleright . Then, we call $[[\alpha(P)]] \in [T_\Sigma^\infty \rightarrow_\perp [D^2 \rightarrow D^2]]$ the higher-order removal of f .

Below, we show an equivalent, but more syntactic description of the definition of higher-order removal. Suppose that a result f of algebraic fusion has the following recursive definition for each $o \in \Delta^{(n)}$:

$$f(o(t_1, \dots, t_n)) = (\lambda w x.g_1(x, w(f_1 x))) \circ f t_{i_1} \circ \dots \circ f t_{i_{l+1}} \circ (\lambda w x.g_{l+1}(x, w(f_{l+1} x))),$$

where l is a natural number, $1 \leq i_1, \dots, i_l \leq n$ are indexes for subterms, and $f_1, \dots, f_{l+1} \in [D \rightarrow D]$ and $g_1, \dots, g_{l+1} \in [D^2 \rightarrow D]$ are continuous functions. Then, the higher-order removal of f is a strict continuous function $f' \in [T_\Sigma^\infty \rightarrow_\perp [D^2 \rightarrow D^2]]$, which has the following recursive definition for each $o \in \Delta^{(n)}$:

$$f'(o(t_1, \dots, t_n))(x, y) = \text{letrec } (q_1, s_1) = (f_1 x, g_1(x, r_1)) \text{ and } (p_2, r_1) = f' t_{i_1}(q_1, s_2)$$

...

$$\text{and } (q_j, s_j) = (f_j p_j, g_j(p_j, r_j))$$

$$\text{and } (p_{j+1}, r_j) = f' t_{i_j}(q_j, s_{j+1})$$

...

$$\text{and } (q_l, s_l) = (f_l p_l, g_l(p_l, r_l))$$

$$\text{and } (p_{l+1}, r_l) = f' t_{i_l}(q_l, s_{l+1})$$

$$\text{and } (q_{l+1}, s_{l+1}) = (f_{l+1} p_{l+1}, g_{l+1}(p_{l+1}, y))$$

$$\text{in } (q_{l+1}, s_1).$$

We show that the higher-order removal retains the computational content of the original program, and it can be recovered via

²It is interesting to point out that the connection of boxes in Figure 3 is exactly the pattern which appeared in the definition of the composition of morphisms in Abramsky's geometry-of-interaction construction [1] and Joyal et al.'s Int-construction [12].

β . Since β is a strict monoid homomorphism, the following theorem is an immediate consequence of Proposition 5.2.

Theorem 5.7 *Let $f \in [T_\Sigma^\infty \rightarrow_\perp [[D \rightarrow D] \rightarrow [D \rightarrow D]]]$ be a result of an algebraic fusion such that f can be improved with the monoid D^\triangleright , the strict monoid homomorphism $\beta \circ \alpha$, and a polynomial Σ -algebra P over D^\triangleright .*

Then the higher-order removal $\llbracket \alpha(P) \rrbracket \in [T_\Sigma^\infty \rightarrow_\perp [D^2 \rightarrow D^2]]$ of f satisfies

$$f = \beta \circ \llbracket \alpha(P) \rrbracket.$$

Example 5.8 We demonstrate the higher-order removal with the result of the algebraic fusion of $\text{ex} \in [T_{\text{nat}}^\infty \rightarrow_\perp [T_{\text{tree}}^\infty \rightarrow T_{\text{tree}}^\infty]]$ as a producer and $\text{bl} \in [T_{\text{tree}}^\infty \rightarrow_\perp [T_{\text{tree}}^\infty \rightarrow T_{\text{tree}}^\infty]]$ as a consumer:

$$\begin{aligned} \text{ex } Z &= \lambda x. N(L, x) \\ \text{ex } (S n) &= \lambda x. \text{ex } n (\text{ex } n x) \end{aligned}$$

$$\begin{aligned} \text{bl } L &= \lambda x. x \\ \text{bl } (N(l, r)) &= \lambda x. N(\text{bl } l (N(x, x)), \text{bl } r (N(x, x))). \end{aligned}$$

The producer function ex , given L as the initial accumulator argument, translates a natural number representation $S^n(Z)$ into a binary tree of depth $2^n + 1$, where the left branch of every binary node is a leaf node. The consumer function bl , given L as the initial accumulator argument, replaces every leaf node at level k (with the root node's level being 0) with a complete binary tree of depth $k + 1$.

First, ex satisfies (C-prod^{*}), because the following polynomial nat-algebra Ex over C_{tree}^∞ :

$$\begin{aligned} \text{Ex}_Z &= N(L, [-]) \\ \text{Ex}_S[X] &= X \cdot X \end{aligned}$$

satisfies $\text{fill}^\infty \circ \llbracket \text{Ex} \rrbracket = \text{ex}$. It is easy to see that bl satisfies (C-cons^{*}). Therefore we proceed to algebraic fusion. We lift bl to a strict continuous monoid homomorphism $\bar{\text{bl}} : C_{\text{tree}}^\infty \rightarrow [[\text{tree} \rightarrow \text{tree}]]$ as follows:

$$\begin{aligned} \bar{\text{bl}} L &w = \lambda x. x \\ \bar{\text{bl}} (N(l, r)) &w = \lambda x. N(\bar{\text{bl}} l w (N(x, x)), \bar{\text{bl}} r w (N(x, x))) \\ \bar{\text{bl}} [-] &w = w. \end{aligned}$$

We then calculate the polynomial nat-algebra $\bar{\text{bl}}(\text{Ex})$ over $[[T_{\text{tree}}^\infty \rightarrow T_{\text{tree}}^\infty]] \Rightarrow [T_{\text{tree}}^\infty \rightarrow T_{\text{tree}}^\infty]$:

$$\begin{aligned} \bar{\text{bl}}(\text{Ex}_Z) &= \bar{\text{bl}}(N(L, [-])) \\ &= \lambda wx. N(N(x, x), w(N(x, x))) \\ \bar{\text{bl}}(\text{Ex}_S)[X] &= X \circ X. \end{aligned}$$

Thus, the result of algebraic fusion is $\llbracket \bar{\text{bl}}(\text{Ex}) \rrbracket \in [\text{nat} \rightarrow_\perp [[\text{tree} \rightarrow \text{tree}] \rightarrow [\text{tree} \rightarrow \text{tree}]]]$. We simply call this result blex .

We apply the higher-order removal to blex . We need to find a polynomial nat-algebra P over $(T_{\text{tree}}^\infty)^\triangleright$ such that $\beta \circ \alpha(P) = \bar{\text{bl}}(\text{Ex})$. An answer is

$$\begin{aligned} P_Z &= (\lambda x. N(x, x), \lambda(x, y). N(N(x, x), y)) \\ P_S[X] &= X @ X, \end{aligned}$$

and the following polynomial nat-algebra $\alpha(P)$ over $(T_{\text{tree}}^\infty)^\infty$

$$\begin{aligned} \alpha(P_Z) &= \lambda(x, y). (N(x, x), N(N(x, x), y)) \\ \alpha(P_S)[X] &= X \infty X \end{aligned}$$

gives the higher-order removal $\llbracket \alpha(P) \rrbracket \in [T_{\text{nat}}^\infty \rightarrow_\perp [(T_{\text{tree}}^\infty)^\triangleright]^2 \rightarrow (T_{\text{tree}}^\infty)^\triangleright]$ of blex . Below, we simply write blex' for $\llbracket \alpha(P) \rrbracket$. The

recursive definition of blex' is

$$\begin{aligned} \text{blex}' Z &(x, y) = (N(x, x), N(N(x, x), y)) \\ \text{blex}' (S n) &(x, y) = (\text{blex}' n) \infty (\text{blex}' n) (x, y) \\ &= \text{letrec } (p, q) = \text{blex}' n (x, s) \\ &\quad \text{and } (r, s) = \text{blex}' n (p, y) \text{ in } (r, q), \end{aligned}$$

and from Theorem 5.7, blex' satisfies

$$\begin{aligned} \text{blex } t &= \beta (\text{blex}' t) \\ &= \lambda wx. \text{letrec } (p, q) = \text{blex}' t (x, w p) \text{ in } q. \end{aligned}$$

6. Related Work

In an early study of fusion laws, Sheard and Fegaras presented the second-order promotion theorem [18]. They demonstrated that the identity function can be obtained from the composition of the list reverse function with itself by applying their fusion law and what they call “inverse” transformation. The present paper gives a more formal and general exposition for this transformation process.

Shortcut fusion by Gill et al. [9] is one of the most successful fusion methods in practice, because of its conceptual simplicity: a single fusion law for program calculation is derived from the parametricity principle [14, 21]. Shortcut fusion has been refined and extended in many directions. Takano et al. generalized it to arbitrary algebraic data types [19]. In [8], Gill introduced a combinator called *augment* to accommodate his shortcut fusion with the list append function. Johann generalized his *augment* combinator to arbitrary algebraic data types [11] and proved its correctness. Ghani et al. analyzed the underlying mathematical structure of the *augment* combinator, and proposed a more general scheme called *monadic augment* [7, 6]. The present paper offers an alternative solution in an algebraic setting, where the elementary theory of monoids and universal algebra give a principled way of devising fusion laws. Built on different concepts, these fusion methods are closely related but have many subtle differences as well, as discussed in Section 3.6.

Takehi et al. [13] formulated *dmap* as a combinator and provided a fusion laws for it. Their fusion law works quite elegantly for some simple but important functions, but it cannot handle cases that are more general. As demonstrated herein, their fusion law can be explained in terms of a rather simple monoid homomorphism.

The second author, Nishimura, proposed a powerful fusion method [16, 17], whose transformation principle was originally drawn from a technique of attribute grammar [5]. His fusion method, however, crucially relies on a transformation called higher-order removal, whose transformation rules have a broken locality. This adds an extra complexity and makes it difficult to recognize the essence of his fusion technique. The present paper gives a strikingly clear account of the intricate task in removing higher-order functions. The intricacy is pushed into the corresponding monoids and monoid homomorphisms and the transformation result naturally follows. This simple but universal principle for the derivation of transformation rules, as we have seen in Section 5, has allowed us to show that the transformation rules work as well for eliminating partial and infinite data structures. To the authors' knowledge, this is the first work that presents a formal account of this issue.

There is another stream of developments of fusion techniques, based on Burstall and Darlington's unfold-fold transformation [2]. This line of research was initiated by Wadler, who presented *deforestation* [22] as a fusion transformation for a particular class of list processing programs. It has been recognized that it is difficult to control the deforestation process, when it is applied to functions with accumulating parameters, because some obstructing function calls may incur infinite unfoldings [3].

Voigtländer has proposed the *lazy composition* technique [20] that solves the issue by an elegant use of circular *let*. His technique

was derived from the corresponding technique developed for tree transducers [4], which are a close cousin of the attribute grammar formalism. However, he has only informally discussed the correctness of his method resorting to an intuitive account of the laziness.

It may be valuable to point out that the visual diagrams in Section 5.2 are very similar to those appearing in the literature on the composition techniques of attribute grammars and tree transducers. This reinforces our claim on the universality of our method; that is, ours is quite successful for capturing the common principle embodied in varying transformation mechanisms that have been independently developed in different formalisms.

7. Conclusion and Future Work

This paper has demonstrated that the task of fusing the producer function and consumer function which have one accumulating parameter can be cleanly described in terms of monoids and monoid homomorphisms. We have defined algebraic fusion, a fusion method based on the algebraic setting, and further developed the notion of improvement that allows us to algebraically refine the results of fusion. We have examined that algebraic fusion and improvement explain varying existing fusion methods gracefully. We hope that the present paper provides a fresh look at fusion transformations.

We believe that the general concept of our work can be applied to a wider range of fusion problems. We are now working on extending the underlying algebraic domain (monoids) to richer ones, so that the function rp in Section 3.4, for instance, can be fused.

Acknowledgment

The authors thank anonymous reviewers for their constructive criticisms and encouraging comments. We are also grateful to Masahito Hasegawa, who pointed out the connection between Figure 3 and Int-construction, and to Takeshi Abe for fruitful discussions.

References

- [1] S. Abramsky. Retracting some paths in process algebra. In *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *LNCS*, pages 1–17. Springer, 1996.
- [2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.
- [3] W.-N. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.
- [4] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics, Formal Models Based on Tree Transducers*. Springer Verlag, 1998.
- [5] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19(6) of *SIGPLAN Notices*, pages 157–170, June 1984.
- [6] N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic augment and generalised short cut fusion. In *International Conference on Functional Programming (ICFP '05)*, pages 294–305. ACM Press, 2005.
- [7] N. Ghani, T. Uustalu, and V. Vene. Generalizing the augment combinator. In *Trends in Functional Programming 5*, pages 65–78. Intellect, 2006.
- [8] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1996.
- [9] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM Press, June 1993.
- [10] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, 1977.
- [11] P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.
- [12] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. In *Math. Proc. Camb. Phil. Soc.*, pages 447–468, 1996.
- [13] K. Takeuchi, R. Glück, and Y. Futamura. On deforesting parameters of accumulating maps. In *Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001*, volume 2372 of *LNCS*, pages 46–56. Springer Verlag, 2001.
- [14] Q. Ma and J. C. Reynolds. Types, abstractions, and parametric polymorphism, part 2. In *Proc. of Mathematical Foundations of Programming Semantics (MFPS 1991)*, volume 598 of *LNCS*, pages 1–40. Springer Verlag, 1991.
- [15] G. Malcolm. Homomorphisms and promotability. In *Mathematics of Program Construction*, volume 375 of *LNCS*, pages 335–347. Springer Verlag, 1989.
- [16] S. Nishimura. Correctness of a higher-order removal transformation through a relational reasoning. In *Programming Language and Systems, First Asian Symposium, APLAS 2003 Proceedings*, volume 2895 of *LNCS*, pages 358–375. Springer Verlag, 2003.
- [17] S. Nishimura. Fusion with stacks and accumulating parameters. In *Proc. of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 101–112. ACM Press, 2004.
- [18] T. Sheard and L. Fegaras. A fold for all seasons. In *International Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, pages 233–242. ACM Press, 1993.
- [19] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. of International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 306–313. ACM Press, 1995.
- [20] J. Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17(1), 2004.
- [21] P. Wadler. Theorems for free! In *International Conference on Functional Programming and Computer Architecture (FPCA '89)*, pages 347–359. Addison-Wesley, 1989.
- [22] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.