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
(*e-mail:* sinya@kurims.kyoto-u.ac.jp)

Susumu Nishimura

Department of Mathematics, Faculty of Science, Kyoto University, Kyoto 606-8502, Japan
(*e-mail:* susumu@math.kyoto-u.ac.jp)

Abstract

This paper develops a new framework for fusion that is designed for eliminating the intermediate data structures involved in the composition of functions that have one accumulating parameter. The new fusion framework comprises two steps: *algebraic fusion* and its subsequent *improvement* process.

The key idea in our development is to regard functions with an accumulating parameter as functions that operate over the monoid of *data context*s. Algebraic fusion composes each such function with a monoid homomorphism that is derived from the definition of the consumer function to obtain a higher-order function that computes over the monoid of endofunctions. The transformation result may be further refined by an improvement process, which replaces the operation over the monoid of endofunctions (i.e., function closures) with another monoid operation over a monoid structure other than function closures.

Using our framework, one can formulate a particular solution to the fusion problem by devising appropriate monoids and monoid homomorphisms. This provides a unified exposition of a variety of fusion methods that have been developed so far in different formalisms. Furthermore, the cleaner formulation makes it possible to argue about some delicate issues on a firm mathematical basis. We demonstrate that algebraic fusion and improvement in the world of CPOs and continuous functions can correctly fuse functions that operate on partial and infinite data structures. We also show that subtle differences in termination behaviors of transformed programs caused by certain different fusion methods can be cleanly explained by corresponding improvement processes that have different underlying monoid structures.

1 Introduction

Modular programming is a widely approved programming discipline and functional programming languages support fine modularity by encouraging us to write a program as a combination of small functions. However, this fine modularity comes with a cost: those small functions must be interfaced with *intermediate data*. For instance, given a composition $c \circ p$ of two functions of type $p, c : \mathsf{list} \to \mathsf{list}$, the *producer* function p generates the intermediate list which is immediately consumed by the *consumer* function c. Though necessary for the composition, the production of the intermediate list is usually inessential for the computation. Thus we would prefer to replace the composed function by an individual function that does not produce any intermediate data.

Fusion is a family of program transformation techniques which transform a given pair (p, c) of producer and consumer functions into an individual function that performs the same computation as the composed function $c \circ p$ does. A substantial amount of research has been done in the last few decades to develop fusion techniques. Wadler (1990) presented a transformation method called *deforestation*, which is an algorithmic instance of Burstall and Darlington's generic unfold-fold transformation strategy (Burstall & Darlington, 1977). Another significant family of solutions consists of *calculational* approaches, where programs are transformed by stepwise application of a set of equational laws on a few combinators (e.g., generic recursion operators such as foldr on lists). Examples of such calculational methods include the promotion theorem (Malcolm, 1989), shortcut fusion (Gill *et al.*, 1993), and its generalizations (Gill, 1996; Takano & Meijer, 1995; Johann, 2002; Ghani *et al.*, 2005).

1.1 Handling accumulating parameters in fusion transformation

These earlier developments of fusion techniques have been very successful, but they do not work for a significant class of functions — functions with *accumulating parameters*. An accumulating parameter of a recursive function is the function's argument on which temporary data is accumulated during recursion. The most typical function with an accumulating parameter is the tail-recursive list reverse function rev defined as follows:

$$\begin{array}{llll} \mathsf{rev} & [] & x & = & x \\ \mathsf{rev} & (a :: l) & x & = & \mathsf{rev}\ l\ (a :: x), \end{array}$$

where [] stands for the empty list and $a :: x$ for the addition of an element a to the head of a list x.

Accumulating parameters are commonly used in functional programming, but a naïve application of the fold-unfold transformation strategy does not handle them successfully. Let us consider the above reverse function composed with a map function and suppose that the input is a non-empty list $a_1 :: l_1$ and the initial value of the accumulating parameter is the empty list []. Then we have an unfolding:

$$\mathsf{map}\ f\ (\mathsf{rev}\ (a_1 :: l_1)\ []) = \mathsf{map}\ f\ (\mathsf{rev}\ l_1\ [a_1])$$

This does not give any chance of folding and thus we can only continue unfolding, like $\mathsf{map}\ f\ (\mathsf{rev}\ l_1\ [a_1])$, $\mathsf{map}\ f\ (\mathsf{rev}\ l_2\ [a_2, a_1])$, $\mathsf{map}\ f\ (\mathsf{rev}\ l_3\ [a_3, a_2, a_1])$, etc. This symptom is well-known as *"not reaching the accumulating parameter,"* where the values accumulated in the parameter have no chance to be consumed by the outer function application (Chin, 1994).

We may instead apply calculational fusion methods such as shortcut fusion to functions with accumulating parameters, if we regard a function with an accumulating parameter as a function returning a function closure. For example, the list reverse function rev can be regarded as a function that takes an input list and returns a function of type list $\to$ list. However, this means that the result of fusion is a higher-order function, whose evaluation produces function closures instead of intermediate data.

Fusion of functions with accumulating parameters has received much attention and several solutions have been proposed in different formalisms: the composition method for

attribute grammars (Ganzinger & Giegerich, 1984), tree transducer composition methods (Engelfriet & Vogler, 1985; Voigtländer & Kühnemann, 2004), and fusion methods for functional programs (Sheard & Fegaras, 1993; Kakehi *et al.*, 2001; Voigtländer, 2004; Nishimura, 2003; Nishimura, 2004). Though built on different formalisms, these developments have had strong influences on each other and thus often employ similar transformation techniques. (See Section 1.3 for a more detailed historical background and the relationship among these developments.)

These precursor methods brought significant advances in dealing with accumulating parameters but their formulations are very *syntactic*. That is, each transformation method is defined by a set of transformation rules that simply operate on the syntax of programs. Fusion of functions with accumulating parameters is generally a complicated task and thus the syntactic formulation often prevents easy access to the significant ideas behind the transformation techniques. For example, transformation methods by Voigtländer (2004) and Nishimura (2003; 2004) make use of a circular let construction, which is the central mechanism in their methods to encode the computation of accumulating parameters in the target program. However, this encoding, at least at first sight, looks quite tricky. It requires a careful reading of every transformation rule and a deep understanding of the behavior of circular let in order to figure out what global effect is intended by the transformation system. Furthermore the syntactic formulation makes it difficult to compare different fusion methods in a formal setting. The above mentioned methods by Voigtländer and Nishimura, for instance, seem to have very close transformation powers, but establishing a formal statement for that would require a considerable amount of rigorous arguments on their syntactic properties.

It seems that the source of complication is the syntax-oriented formulation of the existing fusion methods. A more semantic investigation into the transformation mechanism behind the syntactic formulation would be needed to better understand of the fusion principle, which would enable establishing formal properties, e.g. comparison of transformation powers between different methods. On a more technical side, a more precise semantic analysis into the fusion principle would also contribute to the establishment of the correctness of each fusion method, i.e. preservation of semantics by program transformation. This is a more delicate matter than one might expect. For example, though Sheard and Fegaras (1993) composed the list reverse function with itself into the identity function, this transformation is not necessarily correct, depending on the semantic domain on which the functions operate. The reverse function composed with itself works as the identity function only if lists of finite size are considered. It behaves differently if the input is an infinite list: the identity function returns the infinite list immediately, while the composition of the reverse functions gets stuck, as the first application of the reverse function falls into an infinite search for the last element of the infinite list.

1.2 Algebraic fusion and its improvement

The fusion method proposed in this paper comprises a fusion transformation called *algebraic fusion* and a strategy called *improvement* which is useful for refining and reasoning about the result of algebraic fusion. The inputs of algebraic fusion are i) a producer, which is a recursive function with one accumulating parameter, and ii) a consumer, which is given

as a catamorphism to an arbitrary set. In this paper we concentrate on the case where the number of accumulating parameters is one. We discuss the merit and demerit of this decision at the end of this section.

The key objects in algebraic fusion are *data contexts*, which are formulated as Σ-contexts in Section 2. Data contexts are data structures with holes, and are used to represent the information accumulated by functions with an accumulating parameter. For instance, function rev described in Section 1.1 accumulates the reverse of the first argument on the second argument. We express this behavior by the following unary function $\mathsf{revc} : \mathsf{list} \to \mathsf{listc}$ which returns list contexts:

$$\mathsf{revc}\ [a_1, \cdots, a_n] = a_n :: \cdots :: a_1 :: [-].$$

The set of data contexts together with the hole and the substitution operation forms a *monoid*, which we exploit for the formulation of the concept of functions with an accumulating parameter.

Algebraic fusion alone is not always a perfect solution for the fusion problem of functions with an accumulating parameter. When both producer and consumer have an accumulating parameter, algebraic fusion transforms their composition into a higher-order function. For example, algebraic fusion of $\mathsf{rev} : \mathsf{list} \to \mathsf{list} \to \mathsf{list}$ with itself results in the following higher-order function $\mathsf{revrev} : \mathsf{list} \to (\mathsf{list} \to \mathsf{list}) \to (\mathsf{list} \to \mathsf{list})$.

$$\begin{array}{llcl} \mathsf{revrev} & [] \quad\ f & = & f \\ \mathsf{revrev} & (a :: l)\ f & = & \mathsf{revrev}\ l\ (\lambda w\ .\ f\ (a :: w)), \end{array}$$

which satisfies $\mathsf{revrev}\ t\ (\mathsf{rev}\ u)\ s = \mathsf{rev}\ (\mathsf{rev}\ t\ u)\ s$. Having higher-order function types means that this function operates over function closures.

In the subsequent improvement phase, we shift the operation over function closures to that over first-order objects. Our strategy is to find a data structure M, a recursive function $g : \mathsf{list} \to M$ and a function $h : M \to (\mathsf{list} \to \mathsf{list}) \to (\mathsf{list} \to \mathsf{list})$ so that we can represent the computation of revrev via h; i.e., $\mathsf{revrev} = h \circ g$. This is essentially the same as finding a factorization of the result of algebraic fusion. Improvement provides a convenient method for solving this problem. For example, with the aid of improvement, we can find a decomposition of revrev into the list append function app and a simple function h that takes only constant time (so $g = \mathsf{app}$ and $M = \mathsf{list} \to \mathsf{list}$). By a simple calculation, we can turn revrev into a function that operates over list rather than $\mathsf{list} \to \mathsf{list}$.

We present the above mentioned idea of algebraic fusion and improvement as a general theory for the fusion of functions with one accumulating parameter. The merits of our solution are addressed as follows.

A meta-theory for different fusion techniques. The theory of algebraic fusion and improvement does not represent a single concrete fusion method but it rather serves as a 'meta-theory' of various fusion methods. That is, we can obtain different fusion methods by instantiating appropriate algebraic structures to the theory. This makes it possible to give a uniform account of different fusion methods on a common platform and consequently makes it easier to compare these different methods. More significantly, it also contributes to showing the essence of the intricate business of dealing with accumulating parameters, giving rise to a better understanding of the transformation mechanism behind them. We will demonstrate the strength of algebraic fusion and improvement by

showing that central techniques of certain existing fusion methods are instances of our theory.

Distinction between delicate semantic differences. Our framework provides a theory for the fusion of functions whose semantics is given by a denotational semantics. This allows us to pinpoint the semantic differences that we discussed in Section 1.1.

The development of algebraic fusion is carried out within the elementary universal algebra over the world of sets and functions (i.e. category **Set**). In this setting, the correctness of algebraic fusion and improvement is shown by an equation between two expressions about set-theoretic functions over total and finite data structures. Next, we redevelop algebraic fusion and improvement for functions over partial and infinite data structures. This is carried out by switching the underlying semantic domain to the world of ω-complete pointed partial orders and continuous functions (i.e. category ω**CPPO**). In this setting, the correctness of improvement is characterized by an inequation unless a strictness condition is satisfied. The inequation expresses that the improved program is more likely to terminate, and (informally) corresponds to the fact that some fusion transformations change termination behavior of functions before and after the transformation. More details on this topic will be discussed in Section 3 and 4.

Straightforward equational reasoning. For each particular instance of an intended fusion transformation task our fusion method gives a small set of calculational laws that allow simple equational reasoning (occasionally involving a few inequations when infinite and partial data structures are under consideration).

These (in)equational laws are not only useful for the derivation of a fusion result but are also essential in establishing the correctness of each particular transformation task. We will later show that the central technique behind the circular let construction found in Voigtländer's or Nishimura's method mentioned above can be formulated in our framework so that it is amenable to equational reasoning, using the standard fixpoint semantics. This establishes a simple correctness proof of the technique and also reveals a new semantic perspective of it.

Being a meta-theory, algebraic fusion and improvement give neither immediate indications of automatic strategy for deriving transformation methods nor any guarantee of improved efficiency. As our theory is primarily concerned with semantic properties of transformed program, it is not suitable for estimating the computational cost of the transformed program. This is in contrast to the previous solutions that are based on syntactic formulations, in which the analysis of computational cost is more manageable (see (Voigtländer, 2007), for example). However, we think that this is not necessarily a deficiency of our solution. We claim that our solution has a definite advantage over the syntactically formulated ones in establishing and analyzing semantic properties of transformation. For instance, the standard proof techniques of denotational semantics can be applied to establish the correctness of the circular let construction technique.

We study the case where the number of accumulating parameters is one. This simplifies the task of formulating the concept of functions with an accumulating parameter, and more importantly, the simplification helps introducing a concise formulation of such functions in terms of the theory of universal algebra and monoids. The price to pay is that algebraic fusion can handle fewer producer functions than some existing fusion methods (e.g., Voigt-

lander's and Nishimura's). However, we believe that our semantic formulation is enough to capture the common usage of an accumulating parameter.

1.3 Related work

The earliest attempts for 'fusion' of programs with accumulating parameters are traced back to 1980's. Ganzinger and Giegerich (1984) proposed a composition method for attribute grammars. Engelfriet and Vogler (1985) developed a composition method for tree transducers (corresponding to the case where consumer functions have no accumulating parameters). Later Kühnemann (1998) proposed an improved method that can compose two macro tree transducers (corresponding to functions that have accumulating parameters) subject to certain restrictions. This method was further extended by Voigtländer and Kühnemann to allow more transducers to be composed (Voigtländer & Kühnemann, 2004). Attribute grammars and tree transducers are close cousins and both include mechanisms for computing with accumulating parameters (Fülöp & Vogler, 1998). Both formalisms are based on formal language theory and provide a general platform for fusion transformation, but their formulations look complicated for non-specialists and this makes their core techniques not easily accessible by a wider audience.

Kakehi et al. (2001) proposed a fusion law for a list-processing combinator called dmap. This combinator satisfies a quite simple calculational rule, which works elegantly for a less general but important class of functions. Voigtländer (2004) presented a fusion method, called lazy composition, that incorporates the macro tree transducer composition technique presented in (Voigtländer & Kühnemann, 2004) into the functional setting. He makes use of circular let to eliminate multiple traversals of the input data. The second author (Nishimura, 2003; Nishimura, 2004) applied the attribute grammar composition technique to obtain a fusion result as a first-order program from a higher-order intermediate transformation result that is obtained by shortcut fusion. He also makes use of circular let in his higher-order removal technique. However, his higher-order removal method sticks to certain particular syntactic forms of program, which makes it difficult to capture its essence.

The solution proposed in this paper gives a cleaner presentation of the second author's transformation on the higher-order programs representing the intermediate program transformation result. Pushing the intricacies related to circularity construction into a suitable monoid structure, we can derive transformation laws in a strikingly simple way.

Here we point out that the computational structure that is employed in the development of our general fusion law coincides with other formalisms. The computational diagram in Figure 3 of Section 5 looks very similar to those found in the literature on the composition techniques of attribute grammars and tree transducers (say, Figure 11 in (Giegerich, 1988)). Interestingly, similar diagrams also appear in the definition of the composition of morphisms in Abramsky's geometry-of-interaction construction (Abramsky, 1996) and Joyal et al.'s Int-construction (Joyal *et al.*, 1996). Though out of the scope of the present paper, it would be interesting to study a deeper connection between these different formalisms.

Shortcut fusion by Gill et al. (1993) 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 (Wadler, 1989; Ma & Reynolds, 1991). Shortcut fusion has been refined and extended in many directions. Takano et al. generalized it

to arbitrary algebraic data types (Takano & Meijer, 1995). Svenningsson (2002) proposed to make use of a dual of the shortcut fusion rule in order to eliminate accumulating parameters, but his method can only deal with accumulating parameters in consumer functions. Gill (1996) introduced a combinator called augment to accommodate shortcut fusion with the list append function. Johann generalized the augment combinator to arbitrary algebraic data types (Johann, 2002) 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 (Ghani *et al.*, 2005; Ghani *et al.*, 2006).

Algebraic fusion proposed in the present paper looks similar to shortcut fusion but is built on different theoretical foundation, namely, the theory of monoids and universal algebra. Built on different concepts, algebraic fusion and shortcut fusion are thus closely related but have many subtle differences as well. See Section 2.5 for a detailed discussion of this topic.

Ohori and Sasano recently proposed lightweight fusion in (Ohori & Sasano, 2007), for the purpose of providing to a practical compiler a built-in fusion engine that can fuse a wide range of typical recursive function definitions with a low additional compiler overhead. In the paper it is demonstrated that their method can fuse a certain class of functions that have extra parameters. However, as their fusion method is a particular instance of Burstall and Darlington's unfold-fold transformation strategy, it also suffers from the problem of "not reaching the accumulating parameter" in dealing with accumulating parameters, as we have seen in Section 1.1.

Hu et al. (1999) proposed a method to derive a function with an accumulating parameter from an ordinary recursive function over a first-order data type. They took a different formulation of the concept of function with an accumulating parameter from ours; they used higher-order catamorphisms, i.e., initial algebra morphisms induced by algebras whose carrier sets are function spaces. While this formulation makes their methods very flexible, it captures some functions that do not use the second parameter as an accumulating parameter. For example, functions that actually reduce the size of the second parameter are included in their formulation. We exclude such functions from our consideration by employing data contexts as a representation of functions over data structures.

Some fusion techniques (such as (Johann, 2002) and (Nishimura, 2003)) base their correctness argument on the observational equivalence of programs. Our fusion method is built on denotational semantics and hence it does not immediately imply correctness up to observational equivalence. Establishing the connection to the observational equivalence property requires a more precise semantic argument, which we leave for future investigation.

1.4 Outline

The rest of the paper is organized as follows. In Section 2 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 3 we introduce the concept of improvement. We demonstrate that the fusion law of Kakehi et al.'s dmap combinator can be derived by algebraic fusion and improvement. Section 4 exploits a more sophisticated monoid supporting partial and infinite data structures and shows that algebraic fusion and improvement can

achieve the same program transformation as the second author's previous work. Finally, Section 7 concludes the paper.

1.5 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. A Σ-*algebra* is a pair $(D, \{\delta_o\}_{o\in\Sigma})$ of a carrier set D and a family δ of functions indexed by operators in Σ such that $\delta_o \in D^n \to D$ for each $o \in \Sigma^{(n)}$. A Σ-*algebra homomorphism* from a Σ-algebra (D, δ) to another Σ-algebra (E, ϵ) is a function $h \in D \to E$ such that $h(\delta_o(d_1, \cdots, d_n)) = \epsilon_o(h(d_1), \cdots, h(d_n))$ holds for any operator $o \in \Sigma^{(n)}$ and $d_1, \cdots, d_n \in D$. The pair of the set T_Σ of closed Σ-terms and the family ι of functions defined by

$$\iota_o(t_1, \cdots, t_n) = o(t_1, \cdots, t_n) \qquad (o \in \Sigma^{(n)}, t_1, \cdots, t_n \in T_\Sigma)$$

is called *initial* Σ-*algebra*, which satisfies the following universal property: for any Σ-algebra (D, δ) there exists a unique Σ-algebra homomorphism (a.k.a. *catamorphism*) from (T_Σ, ι) to (D, δ). We will write this homomorphism by $[\![\delta]\!] \in T_\Sigma \to D$.

We fix a finite set A (ranged over by a) for the elements of lists. We frequently use the following signatures throughout this paper:

$$\mathsf{nat} = \{Z^0, S^1\} \qquad \mathsf{tree} = \{L^0, N^2\} \qquad \mathsf{list} = \{[]^0\} \cup \{a :: (-)^1 \mid a \in A\}.$$

A *monoid* is a tuple $(M, e \in M, \star \in M^2 \to M)$ of a carrier set M, a *unit* e and *multiplication operator* $\star$ such that they obey the following axioms:

$$e \star x = x, \qquad x \star e = x, \qquad (x \star y) \star z = x \star (y \star z). \tag{1}$$

A *monoid homomorphism* h from $\mathcal{M} = (M, e, \star)$ to $\mathcal{N} = (N, \epsilon, *)$ is a function $h \in M \to N$ obeying the following axioms:

$$h(e) = \epsilon, \qquad h(x \star y) = h(x) * h(y). \tag{2}$$

We write $h : \mathcal{M} \to \mathcal{N}$ to mean that h is a monoid homomorphism from $\mathcal{M}$ to $\mathcal{N}$.

2 Algebraic Fusion

In Section 1 we saw that classical fusion strategies such as fold-unfold transformation do not work well with producer functions with an accumulating parameter. We generalize this problem as follows.

We consider the following two functions:

$$\mathsf{prod} \in T_\Delta \to T_\Sigma \to T_\Sigma \qquad \mathsf{cons} \in T_\Sigma \to D,$$

and assume that prod uses the second parameter as an accumulating parameter, like x in the definition of rev. The fusion problem we tackle is to remove the intermediate Σ-terms passed from prod to cons in the computation of the following expression:

$$\mathsf{cons}\ (\mathsf{prod}\ x\ y).$$

The fusion process highly depends on how the concept of functions with an accumulating

parameter is formulated. Therefore, we first discuss two characteristic features of such functions: *sequential accumulation* and *no inspection of the accumulating parameter*, [1] and adopt them as an assumption that classify functions with an accumulating parameter. We then formulate the assumptions by means of *polynomial algebras* over the monoid of Σ-*contexts*. We believe that our formulation covers most functions with an accumulating parameter in common use.

2.1 Σ*-contexts*

Definition 2.1 *For a signature* Σ*, by* Σ^+ *we mean the signature extended with a nullary operator* $[-]$ *denoting a hole (without loss of generality we assume* $[-] \notin \Sigma^{(0)}$*). A* Σ-context k *is simply a* Σ^+*-term. We write* C_Σ *for the set of* Σ*-contexts instead of* T_{Σ^+}.

Σ-contexts are simply Σ-terms which may contain some holes, like

$$S(S([-])) \in C_{\mathsf{nat}}, \qquad N(L, N([-], [-])) \in C_{\mathsf{tree}}, \qquad a :: a' :: [] \in C_{\mathsf{list}}. \tag{3}$$

We equip C_Σ with a monoid structure given by the hole $[-]$ and the substitution operation $- \cdot -$, which is recursively defined by

$$\begin{aligned} [-] \cdot k &= k \\ (o(k_1, \cdots, k_n)) \cdot k &= o(k_1 \cdot k, \cdots, k_n \cdot k). \end{aligned}$$

It is easy to check that $[-]$ and $- \cdot -$ obey the axioms (1) of monoids, hence the following is a reasonable definition.

Definition 2.2 *The monoid* C_Σ *of* Σ-contexts *is the triple* $(C_\Sigma, [-], - \cdot -)$.

Next, we introduce the monoid of endofunctions over a set D.

Definition 2.3 *The* function space monoid $D \Rightarrow D$ *over a set* D *is the triple* $(D \to D, \mathrm{id}_D, - \circ -)$.

Each Σ-context k gives an endofunction $\lambda t \,.\, k[t] \in T_\Sigma \to T_\Sigma$, where $k[t]$ is the Σ-term obtained by filling all holes in k with t. For instance, Σ-contexts in (3) give the following functions:

$$\lambda x \,.\, S(S(x)) \in T_{\mathsf{nat}} \to T_{\mathsf{nat}}, \qquad \lambda x \,.\, N(L, N(x, x)) \in T_{\mathsf{tree}} \to T_{\mathsf{tree}},$$

$$\lambda x \,.\, a :: a' :: [] \in T_{\mathsf{list}} \to T_{\mathsf{list}}.$$

The above correspondence between Σ-contexts and functions is summarized as a function $\mathsf{fill}_\Sigma \in C_\Sigma \to T_\Sigma \to T_\Sigma$ defined by:

$$\mathsf{fill}_\Sigma \; k = \lambda t \,.\, k[t].$$

[1] The latter restriction corresponds to that in every state of a macro tree transducer (Engelfriet & Vogler, 1985) we can not inspect the arguments of the state except the recursion argument.

The subscript of fill may be omitted when it is clear from context. The function fill is injective, so we can think of C_Σ as a part of the function space $T_\Sigma \to T_\Sigma$. Furthermore, fill_Σ respects the monoid structures of C_Σ and $T_\Sigma \Rightarrow T_\Sigma$, i.e.,

$$\mathsf{fill}_\Sigma([-]) = \mathrm{id}_{T_\Sigma}, \qquad \mathsf{fill}_\Sigma(k \cdot k') = \mathsf{fill}_\Sigma(k) \circ \mathsf{fill}_\Sigma(k').$$

Therefore fill_Σ is a *monoid homomorphism* from C_Σ to $T_\Sigma \Rightarrow T_\Sigma$.

2.2 Assumptions on Functions with an Accumulating Parameter

We discuss the assumption we make on functions with an accumulating parameter.

There should be no objection to the function rev and the following function count using the second argument as an accumulating parameter.

$$\begin{array}{llllll} \mathsf{count} & L & x & = & S\ x \\ \mathsf{count} & (N\ (l, r)) & x & = & S\ (\mathsf{count}\ l\ (\mathsf{count}\ r\ x)). \end{array}$$

This function adds the number of leaves in the first argument to the second argument.

On the other hand, what kinds of functions *do not* use the second argument as an accumulating parameter? The first example is the following function repl:

$$\begin{array}{llllll} \mathsf{repl} & L & x & = & x \\ \mathsf{repl} & (N\ (l, r)) & x & = & N\ (\mathsf{repl}\ l\ x, \mathsf{repl}\ r\ x) \end{array}$$

which replaces all the leaves in the first argument with the second argument. In this paper we exclude repl from consideration, because there is no accumulation of information from the second argument, and the flow of information accumulation is discontinuous, i.e., the result of a recursive call is not passed to another recursive call as an accumulating parameter (unlike rev or count).

The second example is the following function rem:

$$\begin{array}{llllll} \mathsf{rem} & [] & x & = & x \\ \mathsf{rem} & (a :: l) & (b :: l') & = & \mathsf{rem}\ l\ l' \\ \mathsf{rem} & (a :: l) & [] & = & \mathsf{rem}\ l\ [] \end{array}$$

This function removes the first n elements from the list in the second argument, where n is the length of the list in the first argument (when it is longer than the second argument, rem simply returns []). This function contradicts the idea of accumulation, as it removes information on the second argument. In general, pattern-matching on the second argument allows us to write functions that reduce the size of the second argument.

In order to exclude such functions, we introduce two assumptions on the notion of functions with an accumulating parameter.

The first assumption is that functions with an accumulating parameter *sequentially* accumulate information on their second argument. We express this assumption in the following way: such a function $f \in T_\Delta \to T_\Sigma \to T_\Sigma$ computes the value of a Δ-term $o(t_1, \cdots, t_n)$ by

$$f\ (o\ (t_1, \cdots, t_n)) = g_1 \circ (f\ t_{i_1}) \circ g_2 \circ \cdots \circ g_l \circ (f\ t_{i_l}) \circ g_{l+1}, \tag{4}$$

where $i_1, \cdots, i_l \in \{1, \cdots, n\}$ are indices of subterms and $g_1, \cdots, g_{l+1} \in T_\Sigma \to T_\Sigma$ are functions which accumulate information on their argument.

The second assumption is that functions with an accumulating parameter *do not* inspect the contents of the accumulating parameter. This assumption requires that each function g_i in (4) can only accumulate some information on its argument, but not inspect it. We express this requirement by the existence of a Σ-context k_i ($1 \leq i \leq l+1$) such that

$$g_i = \mathsf{fill}\ k_i.$$

We summarize the above two assumptions. We regard $f \in T_\Delta \to T_\Sigma \to T_\Sigma$ as a function with an accumulating parameter if it satisfies the following condition:

(C-prod-s) $f \in T_\Delta \to T_\Sigma \to T_\Sigma$ is a recursive function defined by

$$f\ (o\ (t_1, \cdots, t_n)) = (\mathsf{fill}\ k_1) \circ (f\ t_{i_1}) \circ (\mathsf{fill}\ k_2) \circ \cdots \circ (\mathsf{fill}\ k_l) \circ (f\ t_{i_l}) \circ (\mathsf{fill}\ k_{l+1}) \quad (5)$$

where $o \in \Delta^{(n)}$, $i_1, \cdots, i_l \in \{1, \cdots, n\}$ are indices of subterms and $k_1, \cdots, k_{l+1} \in C_\Sigma$ are Σ-contexts representing accumulation of information.

Functions repl and rem are excluded from consideration as they do not satisfy (C-prod-s).

We note that there are some functions that use the second argument as an accumulating parameter but fail to satisfy (C-prod-s). An example of such function is repl' defined by

$$\begin{array}{llcl} \mathsf{repl}' & L & x & = & x \\ \mathsf{repl}' & (N\ (l, r)) & x & = & N\ (\mathsf{repl}'\ l\ L, \mathsf{repl}'\ r\ (N\ (L, x)). \end{array}$$

Although in the second line each recursive call of repl' does not take the result of the other call as an argument, it is reasonable to recognize repl' as a function with an accumulating parameter.

Therefore there is a room for discussion about whether (C-prod-s) is the definitive formulation of functions with an accumulating parameter. However, the producer functions appearing in typical fusion problems of functions with an accumulating parameter satisfy (C-prod-s), and more importantly, (C-prod-s) has a concise mathematical reformulation in terms of *polynomial algebras over monoids*. Therefore we stop pursuing the concept of accumulating parameters here, and proceed to the development of algebraic fusion. In the next section we give a reformulation of (C-prod-s).

2.3 Polynomial Σ-Algebras over Monoids

First, we observe that the following recursive function $f' \in T_\Delta \to C_\Sigma$:

$$f'\ (o\ (t_1, \cdots, t_n)) = k_1 \cdot (f'\ t_{i_1}) \cdot k_2 \cdot \cdots \cdot k_l \cdot (f'\ t_{i_l}) \cdot k_{l+1} \qquad (o \in \Delta^{(n)}), \quad (6)$$

where $k_1, \cdots, k_{l+1} \in C_\Sigma$ are Σ-contexts taken from (5), satisfies

$$\mathsf{fill} \circ f' = f, \quad (7)$$

since fill is a monoid homomorphism. With (7) in mind, we transform (C-prod-s) to the following equivalent condition (C-prod-s'):

(C-prod-s') $f \in T_\Delta \to T_\Sigma \to T_\Sigma$ is a function such that $f = \mathsf{fill} \circ f'$ for some recursive function $f' \in T_\Delta \to C_\Sigma$ defined by (6).

Next, we introduce *polynomials* over a monoid, which generalize the pattern of the right hand side of (6) for arbitrary monoids.

Definition 2.4 *Let $\mathcal{M} = (M, e, \star)$ be a monoid. An n-variable* polynomial *P over $\mathcal{M}$ is a formal expression*

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

where n, l are natural numbers, $c_1, \ldots, c_{l+1}$ are elements in M called coefficients*, and $1 \leq i_1, \cdots, i_l \leq n$ are indices of the formal parameter variables. For readability, we suppress writing units e in the body of a polynomial.* [2]

For example,

$$\mathsf{Rev}_{a::-}[X] = X \cdot (a :: [-]) \qquad (\text{over } C_{\mathsf{list}})$$
$$\mathsf{Count}_N[X_1, X_2] = (S\ [-]) \cdot X_1 \cdot X_2 \qquad (\text{over } C_{\mathsf{nat}})$$

are polynomials over monoids.

Each polynomial over a monoid denotes a function over the carrier set of the monoid.

Definition 2.5 *Let P be an n-variable polynomial over $\mathcal{M} = (M, e, \star)$. We define a function $\mathsf{fun}(P) \in M^n \to M$ by*

$$\mathsf{fun}(P)\ (x_1, \cdots, x_n) = P[x_1/X_1, \cdots, x_n/X_n].$$

We further transform (C-prod-s') to the following equivalent condition (C-prod-s") by means of polynomials over monoids.

(C-prod-s") $f \in T_\Delta \to T_\Sigma \to T_\Sigma$ is a function such that $f = \mathsf{fill} \circ f'$ for some recursive function $f' \in T_\Delta \to C_\Sigma$ defined by

$$f'\ (o\ (t_1, \cdots, t_n)) = \mathsf{fun}(P_o)(f'\ t_1, \cdots, f'\ t_n)$$

where $o \in \Delta^{(n)}$ and P_o is an n-variable polynomial over C_Σ.

We notice that f' is nothing but the initial Δ-algebra homomorphism determined by the following Δ-algebra:

$$(C_\Sigma, \{\mathsf{fun}(P_o)\}_{o \in \Delta}).$$

The essential information of this algebra is of course the family $\{P_o\}_{o \in \Delta}$ of polynomials indexed by operators in Δ. This observation leads us to the notion of *polynomial algebras* over monoids.

Definition 2.6 *A* polynomial Σ-algebra *over a monoid $\mathcal{M} = (M, e, \star)$ is a family P of polynomials indexed by operators in Σ 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 $(M, \{\mathsf{fun}(P_o)\}_{o \in \Sigma})$, which we also refer to as P.*

An example of a polynomial list-algebra over C_{list} is the family

$$\mathsf{Rev} = \{\mathsf{Rev}_{[]} = [-], \mathsf{Rev}_{a::-}[X] = X \cdot (a :: [-])\}. \tag{8}$$

[2] By defining suitable unit and multiplication operator, one can turn the set of polynomials over $\mathcal{M}$ to the monoid that satisfies a similar universal property owned by polynomial rings — this is the reason for the name "polynomial".

This polynomial algebra induces the initial list-algebra homomorphism $[\![\mathsf{Rev}]\!] \in T_{\mathsf{list}} \to C_{\mathsf{list}}$, which has the following recursive definition:

$$\begin{array}{lll} [\![\mathsf{Rev}]\!] \ [] & = & [-] \\ [\![\mathsf{Rev}]\!] \ (a :: l) & = & [\![\mathsf{Rev}]\!] \ l \cdot (a :: [-]). \end{array}$$

This function satisfies

$$[\![\mathsf{Rev}]\!] \ [a_1, \ldots, a_n] = a_n :: \ldots :: a_1 :: [-]. \tag{9}$$

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

$$\begin{array}{l} \mathsf{Count} = \{\mathsf{Count}_L = S \ [-], \\ \qquad\qquad \mathsf{Count}_N[X_1, X_2] = (S \ [-]) \cdot X_1 \cdot X_2\}. \end{array}$$

The initial tree-algebra homomorphism $[\![\mathsf{Count}]\!] \in T_{\mathsf{tree}} \to C_{\mathsf{nat}}$ has the following recursive definition:

$$\begin{array}{lll} [\![\mathsf{Count}]\!] \ L & = & S \ [-] \\ [\![\mathsf{Count}]\!] \ (N \ (l, r)) & = & (S \ [-]) \cdot [\![\mathsf{Count}]\!] \ l \cdot [\![\mathsf{Count}]\!] \ r. \end{array}$$

By the notion of polynomial algebras over monoids, we finally obtain a concise formulation (which is equivalent to (C-prod-s) and its variants) of the assumption on functions with an accumulating parameter.

(C-prod) $f \in T_\Delta \to T_\Sigma \to T_\Sigma$ is a function such that $f = \mathsf{fill} \circ [\![\mathsf{Prod}]\!]$ for some polynomial Δ-algebra Prod over C_Σ.

It is easy to observe that $\mathsf{rev} = \mathsf{fill} \circ [\![\mathsf{Rev}]\!]$ and $\mathsf{count} = \mathsf{fill} \circ [\![\mathsf{Count}]\!]$. Hence rev and count satisfy (C-prod).

We next introduce the concept of *images* of polynomial algebras along monoid homomorphisms.

Definition 2.7 *Let $\mathcal{M} = (M, e, \star), \mathcal{N} = (N, \epsilon, *)$ be monoids, $h : \mathcal{M} \to \mathcal{N}$ be a monoid homomorphism and P be the following n-variable polynomial over $\mathcal{M}$:*

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

We define an n-variable polynomial $h(P)$ over $\mathcal{N}$ by

$$h(P)[X_1, \cdots, X_n] = h(c_1) * X_{i_1} * h(c_2) * X_{i_2} * \ldots * X_{i_l} * h(c_{l+1}).$$

We call this polynomial the image *of P by h.*

Lemma 2.8 *Let $\mathcal{M}, \mathcal{N}$ be monoids, $h : \mathcal{M} \to \mathcal{N}$ be a monoid homomorphism and P be an n-variable polynomial over $\mathcal{M}$. Then we have*

$$\mathsf{fun}(h(P)) \ (h \ m_1, \ldots, h \ m_n) = h \ (\mathsf{fun}(P) \ (m_1, \ldots, m_n)).$$

Proof
By the axioms (2) of homomorphism. □

This simple lemma implies an important property of polynomial Σ-algebras. In general, given a Σ-algebra (D, δ) and a function $f \in D \to E$, there is no generic way to obtain

a Σ-algebra (E, ϵ) so that f becomes a Σ-algebra homomorphism from (D, δ) to (E, ϵ). However, this is possible if δ is a polynomial algebra and f is a monoid homomorphism; for ϵ, we take the *image* of P by h.

Definition 2.9 *Let $\mathcal{M}, \mathcal{N}$ be monoids, $h : \mathcal{M} \to \mathcal{N}$ be a monoid homomorphism and P be a polynomial Σ-algebra over $\mathcal{M}$. The* image *of P by h (denoted by $h(P)$) is the following polynomial Σ-algebra over $\mathcal{N}$:*

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

The following proposition is a variant of the *promotion theorem* (Malcolm, 1989), and plays an essential role in this paper.

Proposition 2.10 *For any monoid $\mathcal{M}, \mathcal{N}$, monoid homomorphism $h : \mathcal{M} \to \mathcal{N}$ and polynomial Σ-algebra P over $\mathcal{M}$, h is a Σ-algebra homomorphism from P to $h(P)$. Therefore*

$$h \circ [\![P]\!] = [\![h(P)]\!].$$

Proof
Follows from Lemma 2.8. □

2.4 Algebraic Fusion

We now return to the original problem. The aim of algebraic fusion is to fuse a producer function and a consumer function:

$$\mathsf{prod} \in T_\Delta \to T_\Sigma \to T_\Sigma \qquad \mathsf{cons} \in T_\Sigma \to D$$

where prod uses the second argument as an accumulating parameter. As discussed before, we express this assumption by the following condition (C-prod):

(C-prod) There exists a polynomial Δ-algebra Prod over C_Σ such that $\mathsf{prod} = \mathsf{fill} \circ [\![\mathsf{Prod}]\!]$.

We also impose the following condition on the consumer:

(C-cons) $\mathsf{cons} = [\![\delta]\!]$ for some Σ-algebra (D, δ).

This is a common requirement in the study of fusion transformations. In other words, cons is a recursive function defined by

$$\mathsf{cons}\ (o\ (t_1, \cdots, t_n)) = \delta_o\ (\mathsf{cons}\ t_1, \cdots, \mathsf{cons}\ t_n) \qquad (o \in \Sigma^{(n)}).$$

We note that any function satisfying (C-prod) also satisfies (C-cons), since

$$\mathsf{prod} = \mathsf{fill} \circ [\![\mathsf{Prod}]\!] = [\![\mathsf{fill}(\mathsf{Prod})]\!]$$

by Proposition 2.10.

2.4.1 The Idea of Algebraic Fusion

We first explain the idea of algebraic fusion by a simple example. The target expression of the fusion problem is

$$\mathsf{map}_f\ (\mathsf{rev}\ x\ y), \tag{10}$$

where $\mathsf{map}_f \in T_{\mathsf{list}} \to T_{\mathsf{list}}$ is the map function defined by

$$\begin{array}{lll} \mathsf{map}_f & [] & = [] \\ \mathsf{map}_f & (a :: l) & = f\ a :: (\mathsf{map}_f\ l). \end{array}$$

Functions rev and map_f satisfy (C-prod) and (C-cons), respectively.

We analyze the computation of (10) with the case where x is instantiated with a three-element list $x_3 = [a_1, a_2, a_3]$. First, the computation of subexpression $\mathsf{rev}\ x_3\ y$ appends the reverse of x_3 to y. Since rev satisfies (C-prod), this computation can be decomposed into two steps: i) calculation of a list-context representing the accumulation, and ii) the execution of the accumulation by filling the hole of the context with y.

$$\begin{array}{ll} & \mathsf{rev}\ x_3\ y \\ = & \mathsf{fill}\ (\llbracket \mathsf{Rev} \rrbracket\ x_3)\ y & \text{by (C-prod)} \\ = & \mathsf{fill}\ (\underline{a_3 :: a_2 :: a_1 :: [-]})\ y & \text{by (9)} \\ = & a_3 :: a_2 :: a_1 :: y. \end{array}$$

We name the underlined list-context k_3. Next, function map_f consumes the above list and yields the result r_3 of the computation of $\mathsf{map}_f\ (\mathsf{rev}\ x_3\ y)$:

$$r_3 = (f\ a_3) :: (f\ a_2) :: (f\ a_1) :: \mathsf{map}_f\ y.$$

The goal of algebraic fusion is to refine the above computation steps so that we can compute r_3 directly from x_3. We observe that:

1. The part of r_3 which depends on x_3 is the first three elements. We separate this part by introducing a new function ϕ_3:

   $$\phi_3 = \lambda w\ .\ (f\ a_3) :: (f\ a_2) :: (f\ a_1) :: w.$$

   With this, we have $r_3 = \phi_3\ (\mathsf{map}_f\ y)$.
2. If we can compute ϕ_3 directly from x_3, then the goal is achieved because the computation of $\phi_3\ (\mathsf{map}_f\ y)$ directly gives r_3 without creating the intermediate list $a_3 :: a_2 :: a_1 :: y$. So we reduce the fusion problem to the quest of a function computing ϕ_3 from x_3.
   In fact, we can calculate ϕ_3 from k_3 by the following function $\overline{\mathsf{map}}_f \in C_{\mathsf{list}} \to T_{\mathsf{list}} \to T_{\mathsf{list}}$, which extends the domain and codomain of map_f to C_{list} and $T_{\mathsf{list}} \to T_{\mathsf{list}}$ respectively:

   $$\begin{array}{llll} \overline{\mathsf{map}}_f & [-] & w & = w \\ \overline{\mathsf{map}}_f & [] & w & = [] \\ \overline{\mathsf{map}}_f & (a :: l) & w & = f\ a :: (\overline{\mathsf{map}}_f\ l\ w). \end{array}$$

   This function is derived by adding to the recursive definition of map_f i) an extra argument w to each line of map_f, and ii) an extra line that handles the case where the first argument is a hole (the first line). The argument w is simply passed to each recursive call of $\overline{\mathsf{map}}_f$, and is returned when the first argument is a hole.
   Function $\overline{\mathsf{map}}_f$ satisfies

   $$\phi_3 = \overline{\mathsf{map}}_f\ k_3 = \overline{\mathsf{map}}_f\ (\llbracket \mathsf{Rev} \rrbracket\ x_3). \tag{11}$$

3. The extension $\overline{\mathsf{map}}_f$ is not merely a function, but also a monoid homomorphism from C_{list} to $T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}}$. The equality

$$\overline{\mathsf{map}}_f\ [-] = \mathrm{id}$$

is obvious. We can also show the following equality by induction on the structure of k_1:

$$\overline{\mathsf{map}}_f\ (k_1 \cdot k_2) = (\overline{\mathsf{map}}_f\ k_1) \circ (\overline{\mathsf{map}}_f\ k_2).$$

Hence we can refine the computation of the right hand side of (11) by Proposition 2.10:

$$\phi_3 = \overline{\mathsf{map}}_f\ (\llbracket \mathsf{Rev} \rrbracket\ x_3) = \llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket\ x_3.$$

From this, $\llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket \in T_{\mathsf{list}} \to T_{\mathsf{list}} \to T_{\mathsf{list}}$ is a good candidate for the function which calculates ϕ_3 directly from x_3. The recursive definition of $\llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket$ is

$$\begin{array}{llcl} \llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket & [] \quad\ \ x & = & x \\ \llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket & (a :: l) \quad x & = & \llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket\ l\ (f\ a :: x), \end{array}$$

and we see that this function performs the list reversal and mapping of f at the same time. Furthermore, it is easy to show that for any $x, y \in T_{\mathsf{list}}$,

$$\llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket\ x\ (\mathsf{map}_f\ y) = \mathsf{map}_f\ (\mathsf{rev}\ x\ y). \tag{12}$$

So we take $\llbracket \overline{\mathsf{map}}_f(\mathsf{Rev}) \rrbracket$ as the answer of the fusion problem.

2.4.2 Algebraic Fusion

Algebraic fusion is a straightforward generalization of the fusion steps described above. Let $\mathsf{prod} \in T_\Delta \to T_\Sigma \to T_\Sigma$ and $\mathsf{cons} \in T_\Sigma \to D$ be functions satisfying (C-prod) and (C-cons) respectively.

The first step of algebraic fusion is to extend the domain and codomain of cons from T_Σ and D to C_Σ and $D \to D$, respectively. This is done by adding two things to the recursive definition of cons: (i) an extra argument w, and (ii) a line which handles the case where the argument is a hole. This extension yields the following recursive function $\overline{\mathsf{cons}} \in C_\Sigma \to D \to D$:

$$\begin{array}{llcll} \overline{\mathsf{cons}} & [-] & w & = & w \\ \overline{\mathsf{cons}} & (o\ (k_1, \cdots, k_n)) & w & = & \delta_o\ (\overline{\mathsf{cons}}\ k_1\ w, \cdots, \overline{\mathsf{cons}}\ k_n\ w). \qquad (o \in \Sigma^{(n)}) \end{array}$$

The extra argument w is distributed to each recursive call of $\overline{\mathsf{cons}}$ (for $o \in \Sigma^{(0)}$, w is simply discarded), and is returned only when the argument is the hole.

This extension can also be described as follows: from the Σ-algebra (D, δ) determining the consumer, we construct a Σ^+-algebra $(D \to D, \overline{\delta})$; its algebra structure $\overline{\delta}_o$ is defined by

$$\begin{array}{rcl} \overline{\delta}_{[-]} & = & \mathrm{id}_D \\ \overline{\delta}_o(f_1, \cdots, f_n) & = & \lambda x \in D\ .\ \delta_o(f_1(x), \cdots, f_n(x)). \qquad (o \in \Sigma^{(n)}) \end{array}$$

Then $\overline{\mathsf{cons}}$ is exactly the initial Σ^+-algebra homomorphism $\llbracket \overline{\delta} \rrbracket : C_\Sigma \to D \to D$.

The extension $\overline{\mathsf{cons}}$ satisfies the following two properties that are essential to algebraic fusion.

Proposition 2.11 *For any* $\mathsf{cons} \in T_\Sigma \to D$ *satisfying (C-cons),*

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

Proof
1, 2) Easy induction on the structure of Σ-contexts. □

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

$$[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!] \in T_\Delta \to D \to D.$$

The following theorem, which generalizes (12), shows the correctness of algebraic fusion.

Theorem 2.12 *For any* $t \in T_\Delta$ *and* $u \in T_\Sigma$*, we have*

$$[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!]\ t\ (\mathsf{cons}\ u) = \mathsf{cons}\ (\mathsf{prod}\ t\ u).$$

Proof
Let $t \in T_\Delta$ and $u \in T_\Sigma$. Then,

$$\begin{aligned}
&[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!]\ t\ (\mathsf{cons}\ u) \\
&= \overline{\mathsf{cons}}\ ([\![\mathsf{Prod}]\!]\ t)\ (\mathsf{cons}\ u) && \text{by Proposition 2.10} \\
&= \mathsf{cons}\ ((\mathsf{fill} \circ [\![\mathsf{Prod}]\!])\ t\ u) && \text{by Proposition 2.11-2} \\
&= \mathsf{cons}\ (\mathsf{prod}\ t\ u) && \text{by (C-prod).}
\end{aligned}$$

□

2.5 Relationship with Shortcut Fusion

In this section we informally compare algebraic fusion and a variant of shortcut fusion (Gill *et al.*, 1993) through a fusion problem where intermediate data structures passed from producers to consumers are tree-terms.

We discuss the variant of shortcut fusion in a call-by-name functional language with parametric polymorphism (such as Haskell), and allow ourselves to use Reynolds' parametricity principle. We fix $k : \tau \to \tau \to \tau$ and $z : \tau$ for some type τ, and write $\mathsf{cata} : \forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to \mathsf{tree} \to \alpha$ for the polymorphic catamorphism constructor for tree-terms (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 $\mathsf{build}' : (\forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha) \to \mathsf{tree} \to \mathsf{tree}$ defined by

$$\mathsf{build}'\ g\ y = g\ (N)\ L\ y,$$

where (N) is the curried version of the data constructor N in tree. This combinator allows us to write functions with an extra argument, including those with an accumulating parameter.

For instance, function repl' in Section 2.2 can be expressed as $\mathsf{repl}' = \mathsf{build}' \circ f$ where f is the following recursive function:

$$\begin{array}{llllll} f & L & p & q & x & = & x \\ f & (N\ (l, r)) & p & q & x & = & p\ (f\ l\ p\ q\ q)\ (f\ r\ p\ q\ (p\ q\ x)). \end{array}$$

By Reynolds' parametricity principle (Wadler, 1989; Ma & Reynolds, 1991), for every $g : \forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha$, the following equality holds:

$$g\ k\ z\ (\mathsf{cata}\ k\ z\ y) = \mathsf{cata}\ k\ z\ (\mathsf{build}'\ g\ y).$$

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

(B-prod) $\exists f.\mathsf{prod} = \mathsf{build}' \circ f$ and
(B-cons) $\mathsf{cons} = \mathsf{cata}\ k\ z$

respectively, we have

$$f\ x\ k\ z\ (\mathsf{cons}\ h) = \mathsf{cons}\ (\mathsf{prod}\ x\ h). \tag{13}$$

The key observation that relates $\mathsf{build}'/\mathsf{cata}$ fusion and algebraic fusion is that the type

$$\forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha$$

of the first argument of build' can be identified with the set of tree-contexts, because this type has a canonical initial tree^+-algebra structure by the parametricity principle (see (Plotkin & Abadi, 1993) for a general exposition). Furthermore, the computation of build' coincides with that of $\mathsf{fill}_{\mathsf{tree}}$. This observation leads us to the following relationship between $\mathsf{build}'/\mathsf{cata}$ fusion and algebraic fusion:

$\mathsf{build}'/\mathsf{cata}$ fusion		Algebraic fusion
(B-prod)	$\cdots$	(C-prod)
(B-cons)	$\cdots$	(C-cons)
Fusion law (13)	$\cdots$	Theorem 2.12.

The difference between $\mathsf{build}'/\mathsf{cata}$ fusion and algebraic fusion is that condition (B-prod) is much weaker than condition (C-prod), i.e., $\mathsf{build}'/\mathsf{cata}$ fusion accepts more producers than algebraic fusion (e.g., function repl and repl' in Section 2.2). In algebraic fusion, producers are supposed to perform primitive recursion over tree-terms and calculate values in the way given by polynomial tree-algebras. On the other hand, $\mathsf{build}'/\mathsf{cata}$ fusion has no such constraints on producers, and the domain of producers can be of any type. The major source of this 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 2.10), while $\mathsf{build}'/\mathsf{cata}$-fusion is formulated in a second-order logic for a polymorphic programming language with the parametricity principle.

Both $\mathsf{build}'/\mathsf{cata}$ fusion and algebraic fusion are driven by essentially the same fusion law; fusion law (13) corresponds to the equation in Theorem 2.12. Therefore, algebraic fusion performs the same fusion transformation as $\mathsf{build}'/\mathsf{cata}$-fusion, but accepts fewer producers than $\mathsf{build}'/\mathsf{cata}$-fusion. However, there is a merit in considering fusion of a restricted class of producers. The program structure of producers is preserved by algebraic

fusion in an explicit form, which makes the subsequent manipulation process easier. In the next section we propose the concept of *improvement*, which is useful for reasoning about and transforming results of algebraic fusion.

3 Improving Algebraic Fusion

Algebraic fusion does not impose any restriction on the codomain of consumers, so it can be a function space $D \to D'$. When such consumers are supplied to algebraic fusion, it results in *higher-order functions* of type $T_\Delta \to (D \to D') \to (D \to D')$. Below we see an example of this situation.

Example 3.1 Function rev satisfies both (C-prod) and (C-cons). Hence we can apply algebraic fusion to fuse rev with itself. We first extend rev to a monoid homomorphism $\overline{\mathsf{rev}} : C_{\mathsf{list}} \to (T_{\mathsf{list}} \to T_{\mathsf{list}}) \Rightarrow (T_{\mathsf{list}} \to T_{\mathsf{list}})$.

$$\begin{array}{llllll} \overline{\mathsf{rev}} & [-] & w & x & = & w\ x \\ \overline{\mathsf{rev}} & (a :: l) & w & x & = & \overline{\mathsf{rev}}\ l\ w\ (a :: x) \\ \overline{\mathsf{rev}} & [] & w & x & = & x \end{array}$$

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

$$\begin{aligned} \overline{\mathsf{rev}}(\mathsf{Rev}) = \{ & \overline{\mathsf{rev}}(\mathsf{Rev}_{[]}) = \mathrm{id}_{T_{\mathsf{list}} \to T_{\mathsf{list}}} \\ & \overline{\mathsf{rev}}(\mathsf{Rev}_{a::\_})[X] = X \circ (\lambda w x\ .\ w(a :: x)) \}. \end{aligned}$$

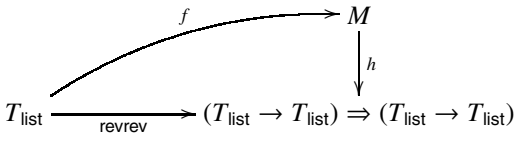
Hence the result of the algebraic fusion of rev with itself is $[\![\overline{\mathsf{rev}}(\mathsf{Rev})]\!] \in T_{\mathsf{list}} \to (T_{\mathsf{list}} \to T_{\mathsf{list}}) \to (T_{\mathsf{list}} \to T_{\mathsf{list}})$. Below we write this function revrev for short. The recursive definition of revrev, with the second argument being explicit, is

$$\begin{array}{llll} \mathsf{revrev} & [] & w & = & w \\ \mathsf{revrev} & (a :: l) & w & = & \mathsf{revrev}\ l\ (\lambda x\ .\ w\ (a :: x)), \end{array}$$

and from Theorem 2.12, revrev satisfies

$$\mathsf{revrev}\ t\ (\mathsf{rev}\ u)\ s = \mathsf{rev}\ (\mathsf{rev}\ t\ u)\ s. \tag{14}$$

Although revrev does not create intermediate lists passed between rev and itself, it is not a satisfactory result because revrev is a higher-order function that creates a closure for each recursive call of itself. Can we represent the computation of revrev by means of some other data structures, say M? We capture this question as a *decomposition (factorization) problem* of revrev into two functions f, h such that $h \circ f = \mathsf{revrev}$.



(15)

In general finding a nontrivial decomposition is difficult, particularly if we do not use any structure of revrev and its (co)domain. We propose a decomposition strategy, called *improvement*, that exploits the underlying structures of algebraic fusion. Suppose that we

find a monoid $\mathcal{M} = (M, e, \star)$ and a monoid homomorphism $h : \mathcal{M} \to (T_{\mathsf{list}} \to T_{\mathsf{list}}) \Rightarrow (T_{\mathsf{list}} \to T_{\mathsf{list}})$ such that the single coefficient $\lambda wx\,.\,w\,(a :: x)$ in $\overline{\mathsf{rev}}(\mathsf{Rev})$ can be given by some $k_c \in M$ via h, i.e.,

$$h(k_c) = \lambda wx\,.\,w\,(a :: x).$$

Then, by replacing the coefficient and multiplication symbol $\circ$ in $\overline{\mathsf{rev}}(\mathsf{Rev})$ with k_c and $\star$ respectively, we obtain the following polynomial algebra P over $\mathcal{M}$:

$$P = \{P_{[]} = e,\ P_{a::-}[X] = X \star k_c\}.$$

This clearly satisfies

$$h(P) = \overline{\mathsf{rev}}(\mathsf{Rev});$$

hence from Proposition 2.10 we obtain a decomposition:

$$[\![h(P)]\!] = h \circ [\![P]\!] = \mathsf{revrev}.$$

Generalizing this pattern, we introduce the concept of improvement.

Definition 3.2 *An* improvement *of the result of algebraic fusion of* $\mathsf{prod} \in T_\Delta \to T_\Sigma \to T_\Sigma$ *satisfying (C-prod) and* $\mathsf{cons} \in T_\Sigma \to D$ *satisfying (C-cons) is a triple of:*

- *a monoid* $\mathcal{M} = (M, e, \star)$*,*
- *a monoid homomorphism* $h : \mathcal{M} \to D \Rightarrow D$ *and*
- *a polynomial* Δ*-algebra* P *over* $\mathcal{M}$ *such that* $h(P) = \overline{\mathsf{cons}}(\mathsf{Prod})$*,*

where $\overline{\mathsf{cons}}$ *and* Prod *are the monoid homomorphism and polynomial algebra defined in Section 2.4.*

We note that there always exists two trivial improvements: i) $\mathcal{M} = C_\Sigma, h = \overline{cons}, P = \mathsf{Prod}$ and ii) $\mathcal{M} = D \Rightarrow D, h = \mathrm{id}_D, P = \overline{\mathsf{cons}}(\mathsf{Prod})$.

Finding an improvement takes the following steps.

1. We first guess a monoid $\mathcal{M}$ and a monoid homomorphism $h : \mathcal{M} \to D \Rightarrow D$ that seem suitable for improvement. This choice requires some heuristics, and depends on the specific fusion problems.
2. For every coefficient $c \in D \to D$ in the polynomial algebra $\overline{\mathsf{cons}}(\mathsf{Prod})$, we find an element $\hat{c} \in M$ such that $h(\hat{c}) = c$. If we cannot find such an element in M for some coefficient c, then we go back to step 1 and try another monoid.
3. Suppose that the component of the polynomial algebra $\overline{\mathsf{cons}}(\mathsf{Prod})$ at $o \in \Sigma^{(n)}$ is the following:

   $$\overline{\mathsf{cons}}(\mathsf{Prod}_o)[X_1, \cdots, X_n] = c_1 \circ X_{i_1} \circ c_2 \circ X_{i_2} \circ \cdots \circ c_l \circ X_{i_l} \circ c_{l+1}$$

   where $c_1, \cdots, c_{l+1} \in D \to D$ and $1 \le i_1, \cdots, i_l \le n$ (c.f. Definition 2.4). We then define the following n-variable polynomial P_o over $\mathcal{M}$:

   $$P_o[X_1, \cdots, X_n] = \hat{c}_1 \star X_{i_1} \star \hat{c}_2 \star X_{i_2} \star \cdots \star \hat{c}_l \star X_{i_l} \star \hat{c}_{l+1}.$$

   By gathering P_o we obtain a polynomial Δ-algebra P over $\mathcal{M}$ such that

   $$h(P) = \overline{\mathsf{cons}}(\mathsf{Prod}).$$

To summarize, when we restrict the decomposition problem (15) to the case where M is a monoid and h is a monoid homomorphism, the problem is reduced to finding appropriate elements in M that give coefficients in $\overline{\mathsf{cons}}(\mathsf{Prod})$ via h.

We note that there seems to be no universal method to find a monoid and monoid homomorphism that guarantee the improvement of efficiency or readability of any result of algebraic fusion.

We devote the rest of this paper for demonstrating the strength and flexibility of improvement. We begin with a small example of improvement, then gradually increase the size and complexity, using sophisticated monoids and monoid homomorphisms. We cover several examples of improvement that give alternative accounts of existing (post-) fusion transformations.

The first example is an improvement of revrev.

Example 3.3 (Continued from Example 3.1) We improve revrev with the following parameters:

Monoid We take the opposite monoid $(T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}})^{\mathsf{op}}$ of $T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}}$, that is, the monoid $(T_{\mathsf{list}} \to T_{\mathsf{list}}, \mathrm{id}_{T_{\mathsf{list}}}, \bullet)$ where the multiplication $\bullet$ is defined by

$$f \bullet g = g \circ f.$$

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

$$h\ \mathrm{id}_{T_{\mathsf{list}}} = \lambda g \,.\, g = \mathrm{id}_{T_{\mathsf{list}} \to T_{\mathsf{list}}}$$

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

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

$$\begin{aligned} P = \{&P_{[]} = \mathrm{id}_{T_{\mathsf{list}}} \\ &P_{a::-}[X] = X \bullet (\lambda x \,.\, a :: x)\}. \end{aligned}$$

This satisfies $h(P) = \overline{\mathsf{rev}}(\mathsf{Rev})$ since the single coefficient in P is mapped to the one in $\overline{\mathsf{rev}}(\mathsf{Rev})$, i.e.,

$$h\ (\lambda x \,.\, a :: x) = \lambda wx \,.\, w\ (a :: x).$$

These data give an improvement $[\![P]\!] \in T_{\mathsf{list}} \to T_{\mathsf{list}} \to T_{\mathsf{list}}$ of revrev via h, and it satisfies

$$\mathsf{revrev} = [\![h(P)]\!] = h \circ [\![P]\!]. \tag{16}$$

We notice that the recursive definition of $[\![P]\!]$, with the second argument being explicit, coincides with that of the list concatenation function app:

$$\begin{array}{llrcl} [\![P]\!] & [] & x & = & x \\ [\![P]\!] & (a :: l) & x & = & ((([\![P]\!]\ l) \bullet (\lambda x \,.\, a :: x))\ x \\ & & & = & a :: ([\![P]\!]\ l\ x). \end{array}$$

Therefore we simply write app for $[\![P]\!]$ below. From Theorem 2.12, we obtain a law about

rev and app:

$$\begin{aligned}
&\mathsf{rev}\ (\mathsf{rev}\ s\ t)\ u \\
&= \mathsf{revrev}\ s\ (\mathsf{rev}\ t)\ u && \text{by (14)} \\
&= h\ (\mathsf{app}\ s)\ (\mathsf{rev}\ t)\ u && \text{by (16)} \\
&= \mathsf{rev}\ t\ (\mathsf{app}\ s\ u) && \text{by definition of } h.
\end{aligned}$$

Taking the opposite monoid is crucial in the improvement of revrev. If we take the ordinary function space monoid $T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}}$ and the monoid homomorphism $h' : (T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}}) \to (T_{\mathsf{list}} \to T_{\mathsf{list}}) \Rightarrow (T_{\mathsf{list}} \to T_{\mathsf{list}})$ defined by $h' = \lambda fg\ .\ f \circ g$, we cannot find a polynomial algebra P' that gives an improvement. This is because the single coefficient $\lambda wx\ .\ w\ (a :: x)$ can not be represented by h', i.e. there is no $f \in T_{\mathsf{list}} \to T_{\mathsf{list}}$ such that

$$\lambda gx\ .\ f\ (g\ x) = \lambda wx\ .\ w\ (a :: x).$$

3.1 Algebraic Fusion of Kakehi et al.'s Dmap with Itself

We next see a bigger example of algebraic fusion and improvement. In (Kakehi *et al.*, 2001), Kakehi et al. studied a combinator called dmap (dm for short), which abstracts a common pattern shared by list-manipulating functions with an accumulating parameter. For functions $f, g \in A \to A$[3], $\mathsf{dm}^g_f \in T_{\mathsf{list}} \to T_{\mathsf{list}} \to T_{\mathsf{list}}$ is recursively defined by

$$\begin{array}{lllll}
\mathsf{dm}^g_f & [] & = & \lambda x\ .\ x \\
\mathsf{dm}^g_f & (a :: l) & = & \lambda x\ .\ (f\ a) :: (\mathsf{dm}^g_f\ l\ ((g\ a) :: x)).
\end{array}$$

Kakehi showed that the following fusion law holds:

$$\mathsf{dm}^{g'}_{f'}\ (\mathsf{dm}^g_f\ l\ l') = (\mathsf{dm}^{f' \circ g}_{f' \circ f}\ l) \circ (\mathsf{dm}^{g'}_{f'}\ l') \circ (\mathsf{dm}^{g' \circ f}_{g' \circ g}\ l) \tag{17}$$

where $f, g, f', g' \in A \to A$ are functions.

In this section we demonstrate that the above law can also be derived using algebraic fusion and improvement. By carefully choosing a monoid in improvement process, we derive (17) without using explicit induction over l or l'.

Algebraic Fusion of dmap and dmap We first apply algebraic fusion to dm^g_f as a producer and $\mathsf{dm}^{g'}_{f'}$ as a consumer. It is easy to check that $\mathsf{dm}^{g'}_{f'}$ satisfies (C-cons). To see that dm^g_f satisfies (C-prod) we first transform the definition of dm^g_f using function composition and fill:

$$\begin{array}{lllll}
\mathsf{dm}^g_f & [] & = & \lambda x\ .\ x \\
 & & = & \mathsf{fill}\ [-] \\
\mathsf{dm}^g_f & (a :: l) & = & \lambda x\ .\ (f\ a) :: (\mathsf{dm}^g_f\ l\ ((g\ a) :: x)) \\
 & & = & (\lambda x\ .\ (f\ a) :: x) \circ (\mathsf{dm}^g_f\ l) \circ (\lambda x\ .\ (g\ a) :: x) \\
 & & = & (\mathsf{fill}\ ((f\ a) :: [-])) \circ (\mathsf{dm}^g_f\ l) \circ (\mathsf{fill}\ ((g\ a) :: [-])).
\end{array}$$

[3] In this article the domain and range of f, g are fixed to A because we only consider the list of elements in A (see Section 1.5). In general, f, g can be any function in $A \to B$, and the discussion in this section is not affected by this generalization.

This definition matches with the condition (C-prod-s), and the following polynomial list-algebra Dm^g_f over C_{list} gives dm^g_f:

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

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

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

then calculate the image of Dm^g_f by $\overline{\mathsf{dm}}^{g'}_{f'}$:

$$\begin{aligned}\overline{\mathsf{dm}}^{g'}_{f'}(\mathsf{Dm}^g_f) = \{&\overline{\mathsf{dm}}^{g'}_{f'}((\mathsf{Dm}^g_f)_{[]}) = \mathrm{id}_{T_{\mathsf{list}} \to T_{\mathsf{list}}}\\ &\overline{\mathsf{dm}}^{g'}_{f'}((\mathsf{Dm}^g_f)_{a::-})[X] = \alpha(f', g', f\ a) \circ X \circ \alpha(f', g', g\ a)\}\end{aligned}$$

where $\alpha(f, g, a)$ is the coefficient defined by

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

The result of the algebraic fusion of dm^g_f and $\mathsf{dm}^{g'}_{f'}$ is thus the initial list-algebra homomorphism $\left[\!\left[\overline{\mathsf{dm}}^{g'}_{f'}(\mathsf{Dm}^g_f)\right]\!\right] \in T_{\mathsf{list}} \to (T_{\mathsf{list}} \to T_{\mathsf{list}}) \to (T_{\mathsf{list}} \to T_{\mathsf{list}})$, but we do not display its recursive definition here.

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

Monoid We take the product monoid $(T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}}) \times (T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}})^{\mathsf{op}}$ whose multiplication will be denoted by $\star$. Explicitly, $\star$ is defined as follows:

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

The first and second projection functions π_1, π_2 from this product monoid are monoid homomorphisms.

Monoid Homomorphism We take the function $h \in (T_{\mathsf{list}} \to T_{\mathsf{list}}) \times (T_{\mathsf{list}} \to T_{\mathsf{list}}) \to (T_{\mathsf{list}} \to T_{\mathsf{list}}) \to (T_{\mathsf{list}} \to T_{\mathsf{list}})$ defined by

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

One can easily verify that this is a monoid homomorphism from $(T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}}) \times (T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}})^{\mathsf{op}}$ to $(T_{\mathsf{list}} \to T_{\mathsf{list}}) \Rightarrow (T_{\mathsf{list}} \to T_{\mathsf{list}})$.

Polynomial Algebra Any coefficient of the form $\alpha(f, g, a)$ can be given by h and the following element $A(f, g, a) \in (T_{\mathsf{list}} \to T_{\mathsf{list}}) \times (T_{\mathsf{list}} \to T_{\mathsf{list}})$ 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

$(T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}}) \times (T_{\mathsf{list}} \Rightarrow T_{\mathsf{list}})^{\mathsf{op}}$:

$$\begin{aligned}\mathsf{DM} = \{&\mathsf{DM}_{[]} = (\mathrm{id}_{T_{\mathsf{list}}}, \mathrm{id}_{T_{\mathsf{list}}})\\ &\mathsf{DM}_{a::\_}[X] = A(f', g', f\ a) \star X \star A(f', g', g\ a)\}\end{aligned}$$

satisfies

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

From this, we obtain an improvement $[\![\mathsf{DM}]\!] \in T_{\mathsf{list}} \to (T_{\mathsf{list}} \to T_{\mathsf{list}}) \times (T_{\mathsf{list}} \to T_{\mathsf{list}})$ of the result of algebraic fusion of dm^{g}_{f} and $\mathsf{dm}^{g'}_{f'}$, and it satisfies

$$\left[\!\!\left[\overline{\mathsf{dm}}^{g'}_{f'}(\mathsf{Dm}^{g}_{f})\right]\!\!\right] = [\![h(\mathsf{DM})]\!] = h \circ [\![\mathsf{DM}]\!]. \tag{18}$$

Decomposition of the Improvement We calculate the images of DM by π_1 and π_2.

$$\begin{aligned}\pi_1(\mathsf{DM}_{[]}) &= \mathrm{id}_{T_{\mathsf{list}}}\\ \pi_1(\mathsf{DM}_{a::\_})[X] &= (\lambda x\ .\ (f'\ (f\ a)) :: x) \circ X \circ (\lambda x\ .\ (f'\ (g\ a)) :: x)\\ \pi_2(\mathsf{DM}_{[]}) &= \mathrm{id}_{T_{\mathsf{list}}}\\ \pi_2(\mathsf{DM}_{a::\_})[X] &= (\lambda x\ .\ (g'\ (f\ a)) :: x) \bullet X \bullet (\lambda x\ .\ (g'\ (g\ a)) :: x)\end{aligned}$$

By expanding the recursive definition of $[\![\pi_1(\mathsf{DM})]\!]$ and $[\![\pi_2(\mathsf{DM})]\!]$, we obtain

$$[\![\pi_1(\mathsf{DM})]\!] = \mathsf{dm}^{f' \circ g}_{f' \circ f} \qquad [\![\pi_2(\mathsf{DM})]\!] = \mathsf{dm}^{g' \circ f}_{g' \circ g}.$$

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

$$\begin{aligned}&(\mathsf{dm}^{f' \circ g}_{f' \circ f}\ l, \mathsf{dm}^{g' \circ f}_{g' \circ g}\ l)\\ &= ([\![\pi_1(\mathsf{DM})]\!]\ l, [\![\pi_2(\mathsf{DM})]\!]\ l)\\ &= (\pi_1([\![\mathsf{DM}]\!]\ l), \pi_2([\![\mathsf{DM}]\!]\ l)) \qquad \text{by Proposition 2.10}\\ &= [\![\mathsf{DM}]\!]\ l.\end{aligned} \tag{19}$$

From this, we derive the law of dmap:

$$\begin{aligned}&\mathsf{dm}^{g'}_{f'}\ (\mathsf{dm}^{g}_{f}\ l\ l')\\ &= \left[\!\!\left[\overline{\mathsf{dm}}^{g'}_{f'}(\mathsf{Dm}^{g}_{f})\right]\!\!\right]\ l\ (\mathsf{dm}^{g'}_{f'}\ l') && \text{by Theorem 2.12}\\ &= h\ ([\![\mathsf{DM}]\!]\ l)\ (\mathsf{dm}^{g'}_{f'}\ l') && \text{by (18)}\\ &= h\ (\mathsf{dm}^{f' \circ g}_{f' \circ f}\ l, \mathsf{dm}^{g' \circ f}_{g' \circ g}\ l)\ (\mathsf{dm}^{g'}_{f'}\ l') && \text{by (19)}\\ &= (\mathsf{dm}^{f' \circ g}_{f' \circ f}\ l) \circ (\mathsf{dm}^{g'}_{f'}\ l') \circ (\mathsf{dm}^{g' \circ f}_{g' \circ g}\ l) && \text{by definition of } h.\end{aligned}$$

4 Algebraic Fusion and Improvement for Partial and Infinite Data Structures

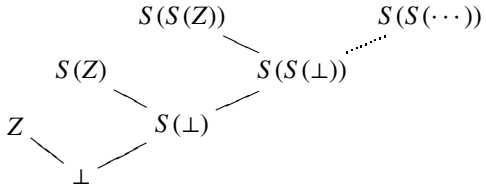
To extend the development in the previous sections to partial and infinite data structures, we replace sets and functions with ω-complete pointed partial orders (CPO for short) and continuous functions. For CPOs D, E, by $[D \to E]$ ($[D \to_{\perp} E]$) we mean the CPO of (strict) continuous functions. The concept of continuous Σ-algebras is fairly standard; see for example (Goguen *et al.*, 1977).

Definition 4.1 *A* continuous Σ-algebra *is a pair* (D, δ) *of a CPO* D *and a family* δ *of continuous functions indexed by operators in* Σ *such that* $\delta_o \in [D^n \to D]$ *for each* $o \in \Sigma^{(n)}$. *A* (strict) continuous Σ-algebra homomorphism $f \in (D, \delta) \to (D', \delta')$ *is a (strict) continuous function* $f \in [D \to D']$ *satisfying*

$$f(\delta_o(x_1, \cdots, x_n)) = \delta'_o(f(x_1), \cdots, f(x_n)) \qquad (x_1, \cdots, x_n \in D)$$

for each $o \in \Sigma^{(n)}$.

It is well-known that we can construct an initial object $\mathcal{T}^\infty_\Sigma = (T^\infty_\Sigma, in^\infty)$ in the category of continuous Σ-algebras and strict continuous Σ-algebra homomorphisms (see e.g. (Goguen *et al.*, 1977)). 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* whose Hasse diagram is illustrated as follows:



Below we assume $T_\Sigma \subseteq T^\infty_\Sigma$ without loss of generality. We identify each operator in Σ and the corresponding continuous term constructor over T^∞_Σ. For a continuous Σ-algebra (D, δ), we write $[\![\delta]\!]$ for the unique strict continuous Σ-algebra homomorphism from $\mathcal{T}^\infty_\Sigma$ to (D, δ).

The universal property of the initial object asserts that for each strict continuous Σ-algebra homomorphism $h \in (D, \delta) \to (D', \delta')$, we have $h \circ [\![\delta]\!] = [\![\delta']\!]$. However, it should be weakened to an inequality if h is a (not necessarily strict) continuous Σ-algebra homomorphism.

Proposition 4.2 *Let* $h \in (D, \delta) \to (D', \delta')$ *be a continuous* Σ*-algebra homomorphism. Then,*

1. $[\![\delta']\!] \sqsubseteq h \circ [\![\delta]\!]$,
2. *for any* $t \in T_\Sigma$, *we have* $[\![\delta']\!]\ t = h\ ([\![\delta]\!]\ t)$, *and*
3. $[\![\delta']\!] = h \circ [\![\delta]\!]$ *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 4.3 *A* continuous monoid *is a monoid* $\mathcal{D} = (D, e, \star)$ *where* D *is a CPO and the multiplication is a continuous function* $- \star - \in [D \times D \to D]$. *A (*strict*)* continuous monoid homomorphism $h : (D, e, \star) \to (E, \epsilon, *)$ *is a (strict) continuous function* $h \in [D \to E]$ *satisfying the laws of monoid homomorphisms.*

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

The definition of monoid polynomials, polynomial Σ-algebras and images of polynomial algebras remains the same. 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 (strict) continuous monoid homomorphism $h : \mathcal{D} \to \mathcal{E}$ is then a (strict) continuous Σ-algebra homomorphism from Q to $h(Q)$ (c.f. Proposition 2.10).

We write C_Σ^∞ for the carrier CPO of the continuous Σ^+-algebra $\mathcal{T}_{\Sigma^+}^\infty$. The substitution operator $- \cdot - \in [(C_\Sigma^\infty)^2 \to C_\Sigma^\infty]$ 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 $\mathsf{fill}_\Sigma^\infty : C_\Sigma^\infty \to [T_\Sigma^\infty \Rightarrow T_\Sigma^\infty]$. The subscript of $\mathsf{fill}_\Sigma^\infty$ may be omitted when it is clear from context.

We introduce the algebraic fusion for partial and infinite data structures. Let

$$\mathsf{prod} \in [T_\Delta^\infty \to_\perp [T_\Sigma^\infty \to T_\Sigma^\infty]], \qquad \mathsf{cons} \in [T_\Sigma^\infty \to_\perp D]$$

be continuous functions satisfying the following conditions:

(C-prod') There exists a polynomial Δ-algebra Prod over C_Σ^∞ such that $\mathsf{prod} = \mathsf{fill}^\infty \circ [\![\mathsf{Prod}]\!]$ (hence, prod should be strict with respect to the first argument).

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

Similarly to the algebraic fusion for total and finite data structures, we first extend the domain and codomain of the consumer function to C_Σ^∞ and $[D \to D]$, respectively. This extension yields a strict continuous monoid homomorphism $\overline{\mathsf{cons}} : C_\Sigma^\infty \to [D \Rightarrow D]$ satisfying

$$\overline{\mathsf{cons}}\ k\ (\mathsf{cons}\ t) = \mathsf{cons}\ (\mathsf{fill}^\infty\ k\ t)$$

for any $k \in C_\Sigma^\infty$ and $t \in T_\Delta^\infty$ (c.f. Proposition 2.11).

We then calculate the image of Prod by $\overline{\mathsf{cons}}$ and obtain the following strict continuous Δ-algebra homomorphism:

$$[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!] \in [T_\Delta^\infty \to_\perp [D \to D]].$$

We call this the result of algebraic fusion of prod and cons.

Theorem 4.4 *For any $x \in T_\Delta^\infty$ and $y \in T_\Sigma^\infty$, we have*

$$[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!]\ x\ (\mathsf{cons}\ y) = \mathsf{cons}\ (\mathsf{prod}\ x\ y).$$

Proof
Since $\overline{\mathsf{cons}}$ is strict, we have $\overline{\mathsf{cons}} \circ [\![\mathsf{Prod}]\!] = [\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!]$ by Proposition 4.2-3. Therefore, we can prove this theorem in the same way as the equational reasoning in the proof of Theorem 2.12. □

The concept of improvement is affected by the transition from the world of sets and functions to that of CPOs and continuous functions. Suppose that we find a continuous monoid $\mathcal{M}$, a polynomial Δ-algebra P over $\mathcal{M}$ and a continuous monoid homomorphism $h : \mathcal{M} \to [D \Rightarrow D]$ such that $h(P) = \overline{\mathsf{cons}}(\mathsf{Prod})$. From Proposition 4.2, we have

1. $[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!] = [\![h(P)]\!] \sqsubseteq h \circ [\![P]\!]$,
2. for any $t \in T_\Delta$, we have $[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!]\ t = h\ ([\![P]\!]\ t)$, and
3. $[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!] = h \circ [\![P]\!]$ if and only if h is strict.

Unlike the improvement in Section 3, we now have the equality $[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!] = h \circ [\![P]\!]$ if and only if h is strict; in general, we merely have the inequality $[\![\overline{\mathsf{cons}}(\mathsf{Prod})]\!] \sqsubseteq h \circ [\![P]\!]$, which means that $h \circ [\![P]\!]$ 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 with itself.

Example 4.5 In the continuous setting, the reverse function rev is interpreted as a continuous function $\mathsf{rev}^\infty \in [T^\infty_{\mathsf{list}} \to_\perp [T^\infty_{\mathsf{list}} \to T^\infty_{\mathsf{list}}]]$ that satisfies (C-prod') and (C-cons'). By applying algebraic fusion to rev^∞ and rev^∞ in the same way as in Example 3.1, we obtain $\mathsf{revrev}^\infty \in [T^\infty_{\mathsf{list}} \to_\perp [[T^\infty_{\mathsf{list}} \to T^\infty_{\mathsf{list}}] \to [T^\infty_{\mathsf{list}} \to T^\infty_{\mathsf{list}}]]]$, which satisfies

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

by Theorem 4.4.

Similarly to Example 3.3, we can improve revrev^∞ by

- the continuous monoid $[T^\infty_{\mathsf{list}} \Rightarrow T^\infty_{\mathsf{list}}]^{\mathsf{op}}$,
- the continuous monoid homomorphism $h^\infty : [T^\infty_{\mathsf{list}} \Rightarrow T^\infty_{\mathsf{list}}]^{\mathsf{op}} \to [[T^\infty_{\mathsf{list}} \to T^\infty_{\mathsf{list}}] \Rightarrow [T^\infty_{\mathsf{list}} \to T^\infty_{\mathsf{list}}]]$ defined by $h^\infty = \lambda f g\ .\ g \circ f$ and
- the polynomial list-algebra P^∞ over $[T^\infty_{\mathsf{list}} \Rightarrow T^\infty_{\mathsf{list}}]^{\mathsf{op}}$, which is the same as P in Example 3.3 except that P's single coefficient $\lambda x\ .\ a :: x \in T_{\mathsf{list}} \to T_{\mathsf{list}}$ is replaced with the continuous function $\lambda x\ .\ a :: x \in [T_{\mathsf{list}} \to T_{\mathsf{list}}]$.

However, here a subtlety about termination behavior slips in; the continuous monoid homomorphism h^∞ is not strict, because

$$h^\infty(\perp) = \lambda g\ .\ g \circ \perp \neq \perp.$$

Hence, from Proposition 4.2 we merely have the following inequality:

$$\mathsf{revrev}^\infty\ t \sqsubseteq h^\infty\ ([\![P^\infty]\!]\ t)$$

and the inequality becomes an equality only for total and finite lists $t \in T_{\mathsf{list}}$. As we have seen in Example 3.3, $[\![P^\infty]\!]$ coincides with the continuous list-concatenation function app^∞. From Theorem 4.4, we obtain:

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

and the inequality becomes an equality for $t \in T_{\mathsf{list}}$. 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 $\mathsf{rev}\ (\mathsf{rev}\ t\ s)\ u$ and $\mathsf{rev}\ s\ (\mathsf{app}\ t\ u)$ in call-by-name languages with lazy lists.

Example 4.6 In the continuous setting, the recursive definition of dm^g_f in Section 3.1 gives a continuous function $\mathsf{dm}^g_f \in [T^\infty_{\mathsf{list}} \to_\perp [T^\infty_{\mathsf{list}} \to T^\infty_{\mathsf{list}}]]$. We can safely replace monoids and monoid homomorphisms used in the derivation process of (17) with continuous counterparts, because monoid homomorphisms h, π_1, π_2 used in the process are all strict. Hence the fusion law (17) holds for partial and infinite lists as well.

5 A Semantic Higher-Order Removal

The motivation for introducing the concept of improvement in Section 3 was that when both a producer and a consumer have an accumulating parameter, their algebraic fusion (hence, shortcut fusion) yields a higher-order function. This problem has already been recognized as a folklore problem, and the second author tackled it in (Nishimura, 2003; Nishimura, 2004). He introduced a program transformation, called *higher-order removal*, that reduces the order of computation. His transformation is designed for a call-by-name functional language, and transforms a result of shortcut fusion:

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

satisfying certain syntactic conditions to a function of the following type:

$$f' : \tau \to (\sigma \times \sigma) \to (\sigma \times \sigma),$$

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. Let D be a CPO. We introduce two continuous monoids $D^{\triangleright}, D^{\infty}$ and two strict continuous monoid homomorphisms $\alpha_D : D^{\triangleright} \to D^{\infty}, \beta_D : D^{\infty} \to [[D \to D] \Rightarrow [D \to D]]$ (the subscript of α, β may be omitted) that play a central role in the semantic representation of higher-order removal given below.

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

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

   To give an intuitive understanding of this monoid, we give a graphical presentation of elements of $D^{\triangleright}$ by circuits processing bidirectional (say, inbound and outbound) information flow. We draw the circuit corresponding to $(f, g) \in D^{\triangleright}$ as shown in Figure 1. It processes inbound and outbound information by f and g respectively. This circuit has an asymmetry in that g can refer to the input of f. Under this graphical presentation, the input-output relation of $(f, g) @ (f', g')$ can be captured by the following series circuit:

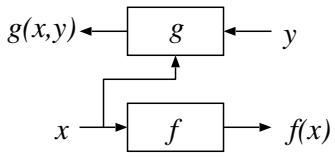


Fig. 1. Diagram for $(f, g) \in D^{\triangleright}$

2. The second continuous monoid is $D^{\infty} = ([D^2 \to D^2], \mathrm{id}_{D^2}, \infty)$ whose multiplication $f \infty g$ is defined by:

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

   where $Y \in [[(D^2)^2 \to (D^2)^2] \to (D^2)^2]$ is the least fixed point operator. If we are

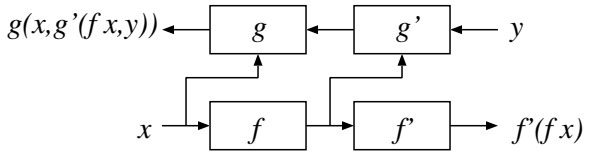


Fig. 2. Diagram for (f, g) @ (f', g')

allowed to use recursive let expressions (see Appendix A for the formal definition), an alternative definition of $f \infty g$ is

$$f \infty g(x, y) = \textbf{let } (p, q) = f\ (x, s)\ ;\ (r, s) = g\ (p, y) \textbf{ in } (r, q).$$

A graphical presentation of this monoid is the following: an element of this monoid is 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 corresponds to the wiring of two processing boxes as follows:

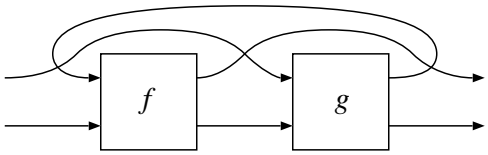


Fig. 3. Diagram for $f \infty g$

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 feed-backed to the upper input of the left box.

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

$$\alpha\ (f, g) = \lambda(x, y)\ .\ (f\ x, g\ (x, y))$$
$$\beta\ f = \lambda w x\ .\ \textbf{let } (p, q) = f\ (x, w\ p) \textbf{ in } q$$

(see Appendix A for the verification of α and β being monoid homomorphisms). Below we give the behavior of α and β in terms of the action on circuit diagrams.

- Monoid homomorphism α flips over g in Figure 1 and regards the entire circuit as a processing box with two input terminals on the left and two output terminals on the right.
- Monoid homomorphism β constructs from $f \in [D^2 \to D^2]$ an action which maps the one-input one-output processing box w drawn on the left in Figure 4 to the one-input one-output circuit described on the right in Figure 4.

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

$$\beta\ (\alpha\ (f, g)) = \lambda w x\ .\ g\ (x, w\ (f\ x)). \tag{20}$$

Definition 5.1 *Let* $f \in [T_{\Sigma}^{\infty} \to_{\perp} [[D \to D] \to [D \to D]]]$ *be a result of algebraic fusion such that* f *can be improved with the monoid* $D^{\triangleright}$*, the strict continuous monoid*

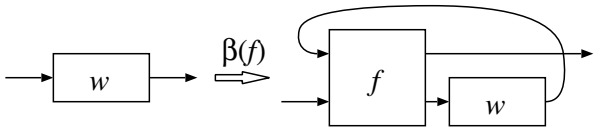


Fig. 4. Action of $\beta(f)$

homomorphism $\beta \circ \alpha$ and a polynomial Σ-algebra P over $D^{\triangleright}$. Then we call $[\![\alpha(P)]\!] \in [T_\Sigma^\infty \to_\perp [D^2 \to D^2]]$ the higher-order removal *of f.*

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

$$\begin{aligned} f(o(t_1, \cdots, t_n)) \quad = \quad & (\lambda w x \,.\, g_1\ (x, w\ (f_1\ x))) \circ f\ t_{i_1} \circ (\lambda w x \,.\, g_2\ (x, w\ (f_2\ x))) \circ \\ & \cdots \circ f\ t_{i_l} \circ (\lambda w x \,.\, g_{l+1}\ (x, w\ (f_{l+1}\ x))), \end{aligned}$$

where l is a natural number, $1 \le i_1, \cdots, i_l \le n$ are indices of subterms and $f_1, \cdots, f_{l+1} \in [D \to D]$ and $g_1, \cdots, g_{l+1} \in [D^2 \to D]$ are continuous functions. That f can be written in this form is equivalent to the improvability of f with some polynomial algebra P over $D^{\triangleright}$. Then the higher-order removal of f is a strict continuous function $f' \in [T_\Sigma^\infty \to_\perp [D^2 \to D^2]]$ defined by

$$\begin{aligned} f'(o(t_1, \cdots, t_n))(x, y) = & \textbf{let}\ (q_1, s_1) = (f_1\ x, g_1\ (x, r_1))\ ; \\ & (p_2, r_1) = f'\ t_{i_1}\ (q_1, s_2)\ ; \\ & \cdots \\ & (q_j, s_j) = (f_j\ p_j, g_j\ (p_j, r_j))\ ; \\ & (p_{j+1}, r_j) = f'\ t_{i_j}\ (q_j, s_{j+1})\ ; \\ & \cdots \\ & (\text{repeat the above pattern till } j = l) \\ & (q_{l+1}, s_{l+1}) = (f_{l+1}\ p_{l+1}, g_{l+1}\ (p_{l+1}, y))\ \textbf{in} \\ & (q_{l+1}, s_1). \end{aligned} \tag{21}$$

This f' is exactly the recursive definition of $[\![\alpha(P)]\!]$ in Definition 5.1.

We show that higher-order removal retains the computational content of the original program, and it can be recovered via β. Since β is a strict monoid homomorphism, the following theorem is an immediate consequence of Proposition 4.2.

Theorem 5.2 *Let $f \in [T_\Sigma^\infty \to_\perp [[D \to D] \to [D \to D]]]$ 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 $[\![\alpha(P)]\!] \in [T_\Sigma^\infty \to_\perp [D^2 \to D^2]]$ of f satisfies

$$f = \beta \circ [\![\alpha(P)]\!].$$

As we discussed in Section 1.2, algebraic fusion can handle fewer producer functions than the modern syntactic fusion transformations using recursive let bindings, such as lazy composition and Nishimura's higher-order removal. However, within the restricted class of

producers, algebraic fusion plus the semantic higher-order removal can achieve the same transformation as these precursors. Below we see an example of such transformation.

We instantiate the set A of elements of lists with $\{0, \cdots, 9, +\}$. We also assume the existence of a continuous function $shows_k \in [T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}]$ which adds the numeric representation of a natural number k in front of a given list. Furthermore, we introduce a new signature term:

$$\mathsf{term} = \{Num^0_k \mid k \in \mathbf{N}\} \cup \{Add^2\}.$$

We consider the fusion of the following functions $\mathsf{asc} \in [T^{\infty}_{\mathsf{term}} \to_{\perp} [T^{\infty}_{\mathsf{term}} \to T^{\infty}_{\mathsf{term}}]]$ and $\mathsf{unp} \in [T^{\infty}_{\mathsf{term}} \to_{\perp} [T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}]]$, which were introduced as a running example of lazy composition by Voigtländer (2004).

$$\begin{array}{llllll}
\mathsf{asc} & Num_k & y & = & Add\ (y, Num_k) \\
\mathsf{asc} & (Add\ (t, u)) & y & = & \mathsf{asc}\ t\ (\mathsf{asc}\ u\ y)
\end{array}$$

$$\begin{array}{llllll}
\mathsf{unp} & Num_k & y & = & shows_k\ y \\
\mathsf{unp} & (Add\ (t, u)) & y & = & \mathsf{unp}\ t\ ('+' :: \mathsf{unp}\ u\ y)
\end{array}$$

Voigtländer applied his lazy composition method to asc (as a producer) and unp (as a consumer) and obtained the following function ascunp (function $\overline{ascunp}''$ in p. 131, (Voigtländer, 2004); the order of outputs is swapped):

$$\begin{array}{llllll}
\mathsf{ascunp} & Num_k & (x, y) & = & ('+' :: shows_k\ x, y) \\
\mathsf{ascunp} & (Add\ (t, u)) & (x, y) & = & \textbf{let}\ (p, q) = \mathsf{ascunp}\ t\ (x, s)\ ; \\
 & & & & \quad (r, s) = \mathsf{ascunp}\ u\ (p, y) \\
 & & & & \textbf{in}\ (r, q).
\end{array}$$

He then applied a post-fusion transformation, called tuple elimination, and derived the following simpler function ascunp' (function $\overline{ascunp}''''$ in p.132, (Voigtländer, 2004)):

$$\begin{array}{llllll}
\mathsf{ascunp}' & Num_k & x & = & '+' :: shows_k\ x \\
\mathsf{ascunp}' & (Add\ (t, u)) & x & = & \mathsf{ascunp}'\ u\ (\mathsf{ascunp}'\ t\ x)
\end{array}$$

In Section 5.2 of (Voigtländer, 2004), he also pointed out that ascunp' shows different termination behavior from ascunp when partial or infinite lists are supplied.

We aim to derive the same results using algebraic fusion and improvement. We first apply algebraic fusion to asc and unp. To see that the producer asc satisfies (C-prod'), we transform the definition of asc using the function composition and fill^{∞}.

$$\begin{array}{llll}
\mathsf{asc}\ Num_k & = & \lambda y\ .\ Add\ (y, Num_k) \\
 & = & \mathsf{fill}^{\infty}(Add\ ([-], Num_k)) \\
\mathsf{asc}\ (Add\ (t, u)) & = & \lambda y\ .\ \mathsf{asc}\ t\ (\mathsf{asc}\ u\ y) \\
 & = & (\mathsf{asc}\ t) \circ (\mathsf{asc}\ u)
\end{array}$$

From this, the following polynomial term-algebra Asc over $C^{\infty}_{\mathsf{term}}$ satisfies $\mathsf{asc} = [\![\mathsf{Asc}]\!]$ (c.f. (C-prod-s)):

$$\begin{array}{c}
\mathsf{Asc} = \{\mathsf{Asc}_{Num_k} = Add\ ([-], Num_k), \\
\mathsf{Asc}_{Add}[X, Y] = X \cdot Y\}.
\end{array}$$

The consumer unp satisfies (C-cons') since it is a recursive function over $T^{\infty}_{\mathsf{term}}$. So we

extend unp to a strict continuous monoid homomorphism $\overline{\mathsf{unp}} : C^{\infty}_{term} \to [[T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}] \Rightarrow [T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}]]$ as follows:

$$\begin{array}{llllll} \overline{\mathsf{unp}} & [-] & w & x & = & w\ x \\ \overline{\mathsf{unp}} & Num_k & w & x & = & shows_k\ x \\ \overline{\mathsf{unp}} & (Add\ (t,u)) & w & x & = & \overline{\mathsf{unp}}\ t\ w\ (\text{'+'} :: \overline{\mathsf{unp}}\ u\ w\ x). \end{array}$$

The image of Asc by $\overline{\mathsf{unp}}$ is

$$\begin{aligned} \overline{\mathsf{unp}}(\mathsf{Asc}) = \{&\overline{\mathsf{unp}}(\mathsf{Asc}_{Num_k}) = \lambda wx\ .\ w\ (\text{'+'} :: shows_k\ x) \\ &\overline{\mathsf{unp}}(\mathsf{Asc}_{Add})[X,Y] = X \circ Y\}, \end{aligned}$$

and the result of the algebraic fusion is $[\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!] \in [T^{\infty}_{\mathsf{term}} \to_{\perp} [[T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}] \to [T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}]]]$. We display its recursive definition below:

$$\begin{array}{lll} [\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]\ Num_k & = & \lambda wx\ .\ w\ (\text{'+'} :: shows_k\ x) \\ [\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]\ (Add\ (t,u)) & = & [\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]\ t \circ [\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]\ u. \end{array}$$

We next derive ascunp and ascunp' by improvement using appropriate monoids. For deriving ascunp, we apply the semantic higher-order removal (Definition 5.1). We observe that the above result can be improved by

- the continuous monoid $(T^{\infty}_{\mathsf{list}})^{\triangleright}$,
- the strict continuous monoid homomorphism

$$\beta_{T^{\infty}_{\mathsf{list}}} \circ \alpha_{T^{\infty}_{\mathsf{list}}} : (T^{\infty}_{\mathsf{list}})^{\triangleright} \to [[T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}] \Rightarrow [T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}]]$$

  and
- the following polynomial term-algebra Q over $(T^{\infty}_{\mathsf{list}})^{\triangleright}$:

$$\begin{aligned} Q = \{&Q_{Num_k} = (\lambda x\ .\ \text{'+'} :: shows_k\ x, \pi_2) \\ &Q_{Add}[X,Y] = X\ @\ Y\}. \end{aligned}$$

  The single coefficient in Q_{Num_k} is obtained by finding continuous functions f, g satisfying

$$\beta \circ \alpha(f,g) = \lambda wx\ .\ g\ (x, w\ (f\ x)) = \lambda wx\ .\ w\ (\text{'+'} :: shows_k\ x).$$

  A solution is $f = \lambda x\ .\ \text{'+'} :: shows_k\ x$ and $g = \pi_2$.

Hence we can give the higher-order removal of $[\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]$. By Definition 5.1, it is $[\![\alpha(Q)]\!] \in [T^{\infty}_{\mathsf{term}} \to_{\perp} [(T^{\infty}_{\mathsf{list}})^2 \to (T^{\infty}_{\mathsf{list}})^2]]$ where the image $\alpha(Q)$ is

$$\begin{aligned} \alpha(Q) = \{&\alpha(Q_{Num_k}) = \lambda(x,y)\ .\ (\text{'+'} :: shows_k\ x, y) \\ &\alpha(Q_{Add})[X,Y] = X \infty Y\}. \end{aligned}$$

By expanding the recursive definition of $[\![\alpha(Q)]\!]$, we conclude $[\![\alpha(Q)]\!] = \mathsf{ascunp}$. We can also directly derive ascunp from the syntactic formulation of higher-order removal

(eq. (21) in page 30). From Theorem 5.2, ascunp satisfies

$$\begin{aligned}
&\mathsf{unp}\ (\mathsf{asc}\ x\ y)\\
&= [\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]\ x\ (\mathsf{unp}\ y) && \text{by Theorem 4.4}\\
&= \beta\ (\mathsf{ascunp}\ x)\ (\mathsf{unp}\ y) && \text{by Theorem 5.2}\\
&= \lambda z\ .\ \mathbf{let}\ (p, q) = (\mathsf{ascunp}\ x)\ (z, \mathsf{unp}\ y\ p)\ \mathbf{in}\ q && \text{by Definition of } \beta.
\end{aligned}$$

Next, we derive ascunp′. We observe that the algebraic fusion result $[\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]$ of asc and unp can be improved by

- the continuous monoid $[T^{\infty}_{\mathsf{list}} \Rightarrow T^{\infty}_{\mathsf{list}}]^{\mathsf{op}}$,
- the non-strict continuous monoid homomorphism $h^{\infty} : [T^{\infty}_{\mathsf{list}} \Rightarrow T^{\infty}_{\mathsf{list}}]^{\mathsf{op}} \to [[T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}] \Rightarrow [T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}]]$ defined by $h^{\infty} = \lambda fg\ .\ g \circ f$ (c.f. Example 4.5) and
- the following polynomial term-algebra R over $[T^{\infty}_{\mathsf{list}} \Rightarrow T^{\infty}_{\mathsf{list}}]^{\mathsf{op}}$:

$$\begin{aligned}
R = \{&R_{Num_k} = \lambda x\ .\ '+' :: shows_k\ x\\
&R_{Add}[X, Y] = X \bullet Y\}.
\end{aligned}$$

  Again, the single coefficient at R_{Num_k} is obtained by finding a continuous function f satisfying

$$h^{\infty}(f) = \lambda g\ .\ g \circ f = \lambda wx\ .\ w\ ('+' :: shows_k\ x).$$

  A solution is $f = \lambda x\ .\ '+' :: shows_k\ x$.

From these data, we obtain an improvement $[\![R]\!] \in [T^{\infty}_{\mathsf{term}} \to_{\perp} [T^{\infty}_{\mathsf{list}} \to T^{\infty}_{\mathsf{list}}]]$ of $[\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]$. The recursive definition of $[\![R]\!]$ coincides with that of ascunp′. Since the monoid homomorphism h^{∞} is non-strict, we have the following inequation:

$$\begin{aligned}
&\mathsf{unp}\ (\mathsf{asc}\ x\ y)\\
&= [\![\overline{\mathsf{unp}}(\mathsf{Asc})]\!]\ x\ (\mathsf{unp}\ y) && \text{by Theorem 4.4}\\
&\sqsubseteq h^{\infty}\ (\mathsf{ascunp}'\ x)\ (\mathsf{unp}\ y) && \text{by Proposition 4.2-1}\\
&= (\mathsf{unp}\ y) \circ (\mathsf{ascunp}'\ x) && \text{by Definition of } h^{\infty}.
\end{aligned}$$

This inequation correctly captures the change of the termination behavior, which is discussed in Section 5.2 of (Voigtländer, 2004).

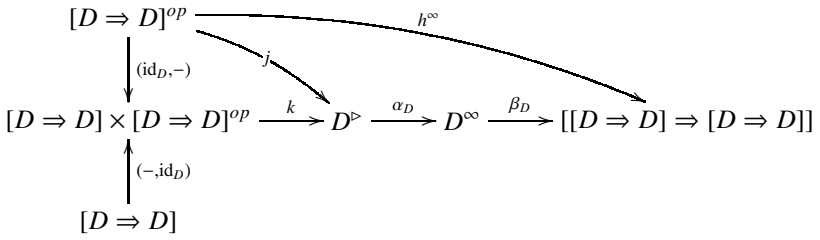
Finally, we point out a relationship between the above two improvements. There is a non-strict continuous monoid homomorphism $j : [T^{\infty}_{\mathsf{list}} \Rightarrow T^{\infty}_{\mathsf{list}}]^{\mathsf{op}} \to (T^{\infty}_{\mathsf{list}})^{\triangleright}$ defined by

$$j(f) = (f, \pi_2),$$

and we have $Q = j(R)$.

6 Summary of Continuous Monoids Used in Improvement

We present a diagram which summarizes continuous monoids and continuous monoid homomorphisms that appeared in this paper.



- In the above diagram, k is a strict continuous monoid homomorphism defined by $k(f, g) = (g, \lambda(x, y) \,.\, f(y))$. The monoid homomorphism $j : [D \Rightarrow D]^{\mathsf{op}} \to D^\triangleright$ in the previous section is actually the composite $k \circ (\mathrm{id}_D, -)$.
- The non-strict continuous monoid homomorphism h^∞ is used to improve the result of algebraic fusion of rev^∞ with itself in Example 4.5. We have $h^\infty = \beta_D \circ \alpha_D \circ j$.
- The continuous monoid homomorphisms α_D and β_D are used in the semantic higher-order removal in Section 5.
- The composite $\beta_D \circ \alpha_D \circ k$ is equal to (the continuous version of) the monoid homomorphism h used in the improvement of the result of algebraic fusion of dmap with itself in Section 3.1 (see also Example 4.6).
- There is a non-strict continuous monoid homomorphism $(-, \mathrm{id}_D) : [D \Rightarrow D] \to [D \Rightarrow D] \times [D \Rightarrow D]^{\mathsf{op}}$, and the composite $\beta_D \circ \alpha_D \circ k \circ (-, \mathrm{id}_D)$ is equal to the continuous monoid homomorphism $f \mapsto \lambda g \,.\, f \circ g$. This is strict, and can be used to improve the result of algebraic fusion of the list append function app^∞ (see Example 4.5) with itself. The improvement tells us the associativity of app^∞.

7 Conclusion

We have developed a new fusion method called algebraic fusion, and its subsequent improvement process, as a general solution to the problem of fusing a producer function and a consumer function which have one accumulating parameter. Built on top of the elementary theory of universal algebra and monoids, our solution provides a simple but flexible framework that gives a clean account of existing fusion methods and also establishes semantic justification for those methods that previously relied on delicate arguments.

As handling accumulating parameters in fusion transformation is a fairly complicated task, we believe that semantic abstractions of the fusion process, such as the algebraic presentation given in this paper, are a quite effective tool for analyzing the existing fusion methods and even devising new ones. The precursor fusion techniques, which we have summarized in the Introduction, provide powerful solutions but are not easily adapted for further extensions and improvements, as they are so densely formulated in their own syntactic realm. We hope that the algebraic exposition in this paper contributes to a deeper understanding of different fusion techniques and that it leads to the development of new techniques.

Acknowledgment

The first author thanks Zhenjiang Hu, who kindly offered an opportunity to present the early version of this work in Tokyo Programming Seminar. Janis Voigtländer's detailed comments and constructive criticism significantly contributed for revising this paper. We are grateful to Samuel Lindley for proofreading the draft of this paper, to Masahito Hasegawa for his comment on the connection between Figure 3 and Int-construction, and to Takeshi Abe for fruitful discussions. Last but not least, the authors thank Julia Lawall and anonymous referees for their valuable comments.

References

Abramsky, Samson. (1996). Retracting some paths in process algebra. *Pages 1–17 of: Proc. of CONCUR '96, concurrency theory, 7th international conference*. Lecture Notes in Computer Science, vol. 1119. Springer.

Burstall, Rod M., & Darlington, John. (1977). A transformation system for developing recursive programs. *Journal of ACM*, **24**(1), 44–67.

Chin, Wei-Ngan. (1994). Safe fusion of functional expressions II: Further improvements. *Journal of functional programming*, **4**(4), 515–555.

Engelfriet, Joost, & Vogler, Heiko. (1985). Macro tree transducers. *Journal of computer and system sciences*, **31**, 71–146.

Fülöp, Zoltán, & Vogler, Heiko. (1998). *Syntax-directed semantics: Formal models based on tree transducers*. Monographs in Theoretical Computer Science. Springer Verlag.

Ganzinger, Harald, & Giegerich, Robert. 1984 (June). Attribute coupled grammars. *Pages 157–170 of: Proceedings of the ACM SIGPLAN '84 symposium on compiler construction*. SIGPLAN Notices, vol. 19(6).

Ghani, Neil, Johann, Patricia, Uustalu, Tarmo, & Vene, Varmo. (2005). Monadic augment and generalised short cut fusion. *Pages 294–305 of: International conference on functional programming (ICFP '05)*. ACM Press.

Ghani, Neil, Uustalu, Tarmo, & Vene, Varmo. (2006). Generalizing the augment combinator. *Pages 65–78 of: Trends in functional programming 5*. Intellect.

Giegerich, Robert. (1988). Composition and evaluation of attribute coupled grammars. *Acta informatica*, **25**(4), 355–423.

Gill, Andrew. (1996). *Cheap deforestation for non-strict functional languages*. Ph.D. thesis, University of Glasgow.

Gill, Andrew, Launchbury, John, & Peyton Jones, Simon. (1993). A short cut to deforestation. *Pages 223–232 of: Proc. of the conference on functional programming languages and computer architecture*. ACM Press.

Goguen, Joseph A., Thatcher, James W., Wagner, Eric G., & Wright, Jesse B. (1977). Initial algebra semantics and continuous algebras. *Journal of ACM*, **24**(1), 68–95.

Hu, Zhenjiang, Iwasaki, Hideya, & Takeichi, Masato. (1999). Calculating accumulations. *New generation comput.*, **17**(2), 153–173.

Johann, Patricia. (2002). A generalization of short-cut fusion and its correctness proof. *Higher-order and symbolic computation*, **15**(4), 273–300.

Joyal, A., Street, R., & Verity, D. (1996). Traced monoidal categories. *Mathematical proceedings of the cambridge philosophical society*, **119**(3), 447–468.

Kakehi, Kazuhiko, Glück, Robert, & Futamura, Yoshihiko. (2001). On deforesting parameters of accumulating maps. *Pages 46–56 of: Logic based program synthesis and transformation, 11th*

international workshop, LOPSTR 2001. Lecture Notes in Computer Science, vol. 2372. Springer Verlag.

Kühnemann, Armin. (1998). Benefits of tree transducers for optimizing functional programs. *Pages 146–157 of: Foundations of software technology and theoretical computer science 18th conference*. Lecture Notes in Computer Science, vol. 1530. Springer Verlag.

Ma, QingMing, & Reynolds, John C. (1991). Types, abstractions, and parametric polymorphism, part 2. *Pages 1–40 of: Proc. of mathematical foundations of programming semantics (MFPS 1991)*. Lecture Notes in Computer Science, vol. 598. Springer Verlag.

Malcolm, G. (1989). Homomorphisms and promotability. *Pages 335–347 of: Mathematics of program construction*. Lecture Notes in Computer Science, vol. 375. Springer Verlag.

Nishimura, Susumu. (2003). Correctness of a higher-order removal transformation through a relational reasoning. *Pages 358–375 of: Programming language and systems, first Asian symposium, APLAS 2003 proceedings*. Lecture Notes in Computer Science, vol. 2895. Springer Verlag.

Nishimura, Susumu. (2004). Fusion with stacks and accumulating parameters. *Pages 101–112 of: Proc. of the 2004 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*. ACM Press.

Ohori, Atsushi, & Sasano, Isao. (2007). Lightweight fusion by fixed point promotion. *Pages 143–154 of: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. ACM press.

Plotkin, G., & Abadi, M. (1993). A logic for parametric polymorphism. *Pages 361–375 of: International conference on typed lambda calculi and applications, TLCA '93*. Lecture Notes in Computer Science, vol. 664. Springer.

Sheard, Tim, & Fegaras, Leonidas. (1993). A fold for all seasons. *Pages 233–242 of: International conference on functional programming languages and computer architecture (FPCA'93)*. ACM Press.

Svenningsson, Josef. (2002). Shortcut fusion for accumulating parameters & zip-like functions. *Pages 124–132 of: Proc. of the 2002 international conference on functional programming*.

Takano, Akihiko, & Meijer, Erik. (1995). Shortcut deforestation in calculational form. *Pages 306–313 of: Proc. of international conference on functional programming languages and computer architecture (FPCA'95)*. ACM Press.

Voigtländer, Janis. (2004). Using circular programs to deforest in accumulating parameters. *Higher-order and symbolic computation*, **17**(1), 129–163.

Voigtländer, Janis. (2007). Formal efficiency analysis for tree transducer composition. *Theory of computing systems*, **41**(4), 619–689.

Voigtländer, Janis, & Kühnemann, Armin. (2004). Composition of functions with accumulating parameters. *Journal of functional programming*, **14**(3), 317–363.

Wadler, Philip. (1989). Theorems for free! *Pages 347–359 of: International conference on functional programming and computer architecture (FPCA'89)*. Addison-Wesley.

Wadler, Philip. (1990). Deforestation: transforming programs to eliminate trees. *Theoretical computer science*, **73**(2), 231–248.

A Proof of α and β being monoid homomorphisms

Definition A.1 *Let $D_1, \cdots, D_n, D$ be CPOs and $f_i \in [D_1 \times \cdots \times D_n \to D_i]$ $(1 \leq i \leq n)$ and $g \in [D_1 \times \cdots \times D_n \to D]$ be continuous functions. We also assume that values $f_i(x_1, \cdots, x_n)$ and $g(x_1, \cdots, x_n)$ can be expressed by mathematical expressions M_i and N using $x_i \in D_i$ $(1 \leq i \leq n)$. By recursive let expression*

$$\textbf{let } x_1 = M_1 \ ; \ \ldots \ ; \ x_n = M_n \textbf{ in } N,$$

we mean the following element in D:

$$g\left(Y\left(\lambda v \in \prod_{i=1}^{n} D_i \,.\, (f_1(v), \cdots, f_n(v))\right)\right)$$

where $Y \in [[\prod_{i=1}^n D_i \to \prod_{i=1}^n D_i] \to \prod_{i=1}^n D_i]$ *is the least fixpoint operator.*

We also allow tuple patterns to appear in the binding, like

$$\textbf{let } (x, y) = M \ ; \ z = L \textbf{ in } N.$$

We regard this as an abbreviation of

$$\textbf{let } v = M[\pi_1(v)/x, \pi_2(v)/y] \ ; \ z = L[\pi_1(v)/x, \pi_2(v)/y] \textbf{ in } N[\pi_1(v)/x, \pi_2(v)/y]$$

where π_1, π_2 are projections from a product CPO.

There are some equations that hold for recursive let expressions:

$$\begin{aligned}
(\textbf{let } V \textbf{ in } L) &= (\textbf{let } \pi(V) \textbf{ in } L) \\
(\textbf{let } (x, y) = (M_1, M_2) \ ; \ V \textbf{ in } L) &= (\textbf{let } x = M_1 \ ; \ y = M_2 \ ; \ V \textbf{ in } L) \\
(\textbf{let } x = f(\cdots, \textbf{let } V \textbf{ in } M, \cdots) \ ; \ W \textbf{ in } L) &= (\textbf{let } x = f(\cdots, M, \cdots) \ ; \ V \ ; \ W \textbf{ in } L) \\
(\textbf{let } x = M \ ; \ V \textbf{ in } L) &= (\textbf{let } V[M/x] \textbf{ in } L[M/x]) \quad (x \notin FV(M))
\end{aligned}$$

where V and W are meta-variables denoting (possibly empty) sequences of variable bindings $x_1 = M_1 \ ; \ \cdots \ ; \ x_n = M_n$. In the first equation $\pi(V)$ is a permutation of the variable bindings in V. In the last equation $V[M/x]$ is the binding obtained by substituting M for every occurrence of x in V.

Below we prove that the continuous functions α_D and β_D defined in Section 5 are monoid homomorphisms. It is obvious that $\alpha_D(\mathrm{id}_D, \pi_2) = \mathrm{id}_{D^2}$ and $\beta_D(\mathrm{id}_{D^2}) = \mathrm{id}_{[D \to D]}$.

$$\begin{aligned}
& \alpha_D(f, g) \mathbin{\infty} \alpha_D(f', g')(x, y) \\
= \ & \textbf{let } (p, q) = (f\ x, g(x, s)) \ ; \ (r, s) = (f'\ p, g'(p, y)) \textbf{ in } (r, q) \\
= \ & \textbf{let } p = f\ x \ ; \ q = g(x, s) \ ; \ r = f'\ p \ ; \ s = g'(p, y) \textbf{ in } (r, q) \\
= \ & (f'\ (f\ x), g(x, g'(f\ x, y))) \\
= \ & \alpha_D((f, g) \mathbin{@} (f', g'))(x, y).
\end{aligned}$$

$$\begin{aligned}
& \beta_D(f \mathbin{\infty} g)\ h\ x \\
= \ & \textbf{let } (p, q) = (\textbf{let } (p', q') = f(x, s) \ ; \ (r, s) = g(p', h\ p) \textbf{ in } (r, q')) \textbf{ in } q \\
= \ & \textbf{let } (p', q') = f(x, s) \ ; \ (r, s) = g(p', h\ p) \ ; \ (p, q) = (r, q') \textbf{ in } q \\
= \ & \textbf{let } (p', q') = f(x, s) \ ; \ (r, s) = g(p', h\ r) \textbf{ in } q' \\
= \ & \textbf{let } (p', q') = f(x, \beta_D\ g\ h\ p') \textbf{ in } q' \\
= \ & ((\beta_D\ f) \circ (\beta_D\ g))\ h\ x.
\end{aligned}$$