Process fusion*

D. Pavlovic
Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304
E-mail: dusko@kestrel.edu

Abstract

The technique of *build fusion*, also known as *deforestation*, removes intermediate results in a composition involving the "build" of an *initial* (inductive, finite) data structure, followed by its consumption. Here we show that it is analogously possible to do *process fusion*, removing intermediate *final* (coinductive, potentially infinite) data passing between a producer and a consumer.

The key observation leading to our results is the fact that the Curry-Howard isomorphism, relating types to propositions, programs to proofs and program composition to cut, extends to the correspondence of *fusion* to *cut elimination*. This simple idea gives us logical interpretations of the basic methods of generic and transformational programming. In the present paper, we provide a logical analysis of the general form of *build fusion* over the inductive data types, regular or nested. The analysis is based on a novel logical interpretation of parametricity in terms of the *paranatural* transformations, introduced in the paper. We extend it to cover *process fusion* on coinductive data types.

The results obtained are truly generic (in the sense of applying to all coinductive (final) data types, including nested ones) and allow a far wider range of optimizations than previously possible. By the standard embedding of initial into final data types, it also applies to arbitrary initial-final mixtures (e.g., infinitely unfolding trees of finite lists). Future work will explore mechanization of the technique and its application to realistic problems.

*This work was supported by the DARPA ITO PCES program.

1 Introduction

1.1 The need for process fusion

An important challenge is to provide automated support for the composition of programs for embedded systems, yielding the efficient high-performance code required for real-time embedded software.

We believe that the best — perhaps only — hope for meeting this objective is to synthesize such systems from high-level, abstract specifications expressing the system requirements while decomposing them into the simplest units possible — expressing the many aspects entering into the total design — and obtain code from further specifications expressing their composition, using both general and domain-specific design theories. This is the philosophy underlying Kestrel's synthesis technology, which has paid off in other application domains.

Here we focus on one small aspect: "optimizing away" the data passing between process components. Simple-minded code generation schemas will tend to translate high-level process composition homomorphically into low-level data passing. While entirely correct as a translation, a successful requirements decomposition might then be penalized by giving rise to relatively inefficient low-level code. We show below how this can be overcome; in fact, the higher the initial abstraction level, the easier it is to analyze the specifications for optimization opportunities and exploit them in synthesis. In other application domains, Kestrel's synthesis technology has obtained efficiency improvements that are beyond the scope of the fiercest attempts at hand-crafted code optimization, simply because the analysis involved is too complicated to be carried out manually for each system generation. The hope is to achieve similar gains here.

The viewpoint of an embedded system as a collection of communicating processes is appropriate. This makes it possible to abstract from implementation details, such as whether processes are sharing a thread or are multi-threaded, or even run on separate processors. These details should be introduced in the synthesis phase, working towards low-level code, and the best data passing paradigm can be chosen given various other aspects of the requirements, such as timing. Below we model data passing as asynchronous

message passing between process components. This leaves an implementation freedom for many low-level schemas from which the most appropriate one can be chosen. The possibilities range from buffered interprocess communication via direct or remote method invocation to shared memory. Here, we are not concerned with these details, because we center on the elimination of data passing.

1.2 Fusion and cut

The Curry-Howard isomorphism is one of the conceptual building blocks of type theory, built deep into the foundation of computer science and functional programming [6, ch. 3]. The fact that it is an *isomorphism* means that the type and the term constructors on one side obey the same laws as the logical connectives, and the logical derivation ruleson the other side. For instance, the products and the sums of types correspond, respectively, to the conjunction and the disjunction, because the respective introduction rules

$$\frac{A \vdash B_0 \quad A \vdash B_1}{A \vdash B_0 \wedge B_1}\wedge I \qquad \frac{A_0 \vdash B \quad A_1 \vdash B}{A_0 \vee A_1 \vdash B}\vee I$$

extended by the labels for proofs, yield the type formation rules

$$\frac{f_0 : A \to B_0 \quad f_1 : A \to B_1}{\langle f_0, f_1\rangle : A \to B_0 \times B_1} \qquad \frac{g_0 : A_0 \to B \quad g_1 : A_1 \to B}{[g_0, g_1] : A_0 + A_1 \to B}$$

In a sense, the pairing constructors $\langle -, -\rangle$ and $[-, -]$ record on the terms the applications of the rules $\wedge I$ and $\vee I$, as the proof constructors.

Extending this line of thought a step further, one notices that the term reductions also mirror the proof transformations. E.g., the transformation

$$\frac{\dfrac{A_0 \vdash B \quad A_1 \vdash B}{A_0 \vee A_1 \vdash B} \quad B \vdash C}{A_0 \vee A_1 \vdash C} \;\blacktriangleright\!\blacktriangleright\; \frac{\dfrac{A_0 \vdash B \quad B \vdash C}{A_0 \vdash C} \quad \dfrac{A_1 \vdash B \quad B \vdash C}{A_1 \vdash C}}{A_0 \vee A_1 \vdash C}$$

corresponds to the rewrite

$$h \cdot [f_0, f_1] \;\blacktriangleright\!\blacktriangleright\; [h \cdot f_0 \;,\; h \cdot f_1] \tag{1}$$

where f_0 and f_1 are the labels of the proofs $A_0 \vdash B$ and $A_1 \vdash B$, whereas h is the label of $B \vdash C$. The point of such transformations is that the applications of the cut rule

$$\frac{A \vdash B \quad B \vdash C}{A \vdash C} \tag{2}$$

get pushed up the proof tree, as to be eliminated, by iterating such moves. On the side of terms and programs, the cut, of course, corresponds to the composition

$$\frac{f : A \to B \quad h : B \to C}{h \cdot f : A \to C} \tag{3}$$

Just like the presence of a cut in a proof means that an intermediary proposition has been created, and then cut out, the presence of the composition in a program means that the thread of computation leads through an intermediary type, used to pass data between the components, and then discarded.

While the programs decomposed into simple parts are easier to write and understand, passing the data and control between the components incurs a computational overhead. For instance, running the composite `ssum · zipW` of

```
zipW                : [Nat]×[Nat] -> [Nat×Nat]
zipW (x::xs,y::ys) = (x,y) :: zip xs ys
zipW  (xs,   ys)   = []
```

and

```
ssum            : [Nat×Nat] -> Nat
ssum []         = 0
ssum (x,y)::zs = x + y + sum zs
```

is clearly less efficient than running the fusion

```
sumzip                 : [Nat]×[Nat] -> Nat
sumzip (x::xs,y::ys)  =  x + y + sumzip (xs,ys)
sumzip  (xs,   ys)    =  0
```

where the intermediary lists `[Nat×Nat]` are eliminated. In practice, the data structures passed between the components tend to be very large, and

the gain by eliminating them can be significant. On the other hand, the efficient, monolythic code, obtained by fusion, tends to be more complex, and thus harder to understand and maintain.

To get both efficiency and compositionality, to allow the programmers to write simple, modular code, and optimize it in compilation, the program fusions need to be sufficiently well understood to be automated. Our first point is that the Curry-Howard isomorphism maps this task onto the well ploughed ground of logic.

1.3 Build fusion

The general form of the build fusion that we shall study corresponds, in the inductive case, to the "cut rule"

$$\frac{\begin{array}{cc} FM_F \xrightarrow{F(\!|c|\!)} FC & \\ \mu\downarrow \quad\quad \downarrow c & \\ A \xrightarrow{f} M_F \qquad M_F \xrightarrow{(\!|c|\!)} C & \end{array}}{A \xrightarrow{f'C(\ulcorner c \urcorner)} C} \tag{4}$$

eliminating the inductive data type M_F, which is the initial algebra of the type constructor F. In practice and literature, F is usually a list- or a tree-like constructor, and the type A is is often required to be inductive itself; but we shall see that the above scheme is valid in its full generality. The `sumzip`-example from the preceding section can be obtained as an instance of this scheme, taking $FX = 1 + \mathtt{Nat} \times \mathtt{Nat} \times X$, and thus $M_F = [\mathtt{Nat} \times \mathtt{Nat}]$. The function `ssum` is the catamorphism (fold) of the map $[0, \ddagger] : 1 + \mathtt{Nat} \times \mathtt{Nat} \times \mathtt{Nat} \longrightarrow \mathtt{Nat}$ where $\ddagger$ maps $\langle i, j, k \rangle$ to $i + j + k$.

The dual scheme

$$\frac{\begin{array}{cc} FA \xrightarrow{F(\!|a|\!)} FN_F & \\ a\uparrow \quad\quad \uparrow \nu & \\ A \xrightarrow{(\!|a|\!)} N_F \qquad N_F \xrightarrow{g} C & \end{array}}{A \xrightarrow{g'A(\ulcorner a \urcorner)} C}$$

allows eliminating the coinductive type N_F, the final F-coalgebra.

Clearly, the essence of both of the above fusion schemes lies in the terms f' and g'. Where do they come from? The idea is to represent the fixpoints M_F and N_F in their "logical form"

$$M_F \cong \forall X.\ (FX \Rightarrow X) \Rightarrow X \tag{5}$$
$$N_F \cong \exists X.\ X \times (X \Rightarrow FX) \tag{6}$$

The parametric families

$$f'X : (FX \Rightarrow X) \longrightarrow (A \Rightarrow X) \tag{7}$$
$$g'X : (X \Rightarrow FX) \longrightarrow (X \Rightarrow C) \tag{8}$$

are then obtained by extending $f : A \longrightarrow M_F$ and $g : N_F \longrightarrow C$ along isomorphisms (5) and (6), and rearranging the arguments. The equations

$$(\!|c|\!) \cdot f = f'C(\ulcorner c \urcorner) \tag{9}$$
$$g \cdot [\![a]\!] = g'A(\ulcorner a \urcorner) \tag{10}$$

can be proved using logical relations, or their convenient derivative, Wadler's "theorems for free" [8]. This was indeed done already in [5] for (9).

Mapped along the Curry-Howard isomorphism, equations (9–10) become statements about the equivalence of proofs. The fact that all logical relations on all Henkin models must relate the terms involved in these equations does not seem to offer a clue for understanding their equivalence.

In order to acquire some insight into the logical grounds of program fusion, and equivalence, we introduce *paranatural* transformations. As a first application, we characterize the parametricity of families (7) and (8) by an intrinsic commutativity property, with no recourse to models or external structures. The upshot is that we obtain slightly stronger results, suitable for generalizing beyond the scope of the current applications of build fusion.

The paranaturality condition is a variation on the theme of functorial and structural polymorphism [4, 3]. But while the dinatural transformations of [4] allow too many terms, the structor morphisms of [3] precisely correspond to the polymorphic terms, but do not stipulate which of many possible choices of structors should be used to interpret the particular polytypes. We fill this gap, presently just enough to analyze the programs *to* the initial and *from* the final data types as parametric/paranatural families. This is the contents

of proposition 3.1. The results obtained eliminate the extensionality and well-pointedness restrictions of the work based on logical relations. More importantly, the logical insights about fusion and parametricity, gained by chasing diagrams in categorical proof theory [7], allow extending the methods of fusion beyond their current scope. Some evidence of this, severely limited by the available space, is offered in the final section.

2 Paranatural transformations

As it has been well known at least since Freyd's work on recursive types in algebraically compact categories [2], separating the covariant and the contravariant occurrences of X in a polytype $\mathcal{T}(X)$ yields a polynomial functor $T : \mathbb{C}^{op} \times \mathbb{C} \longrightarrow \mathbb{C}$. On the other hand, by simple structural induction, one easily proves that

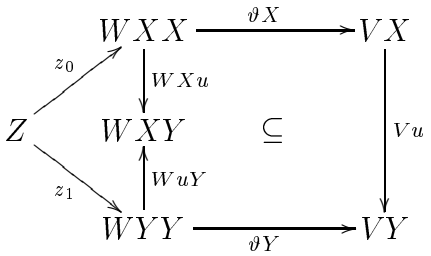
Proposition 2.1 *For every polynomial functor $T : \mathbb{C}^{op} \times \mathbb{C} \longrightarrow \mathbb{C}$ over a cartesian closed category $\mathbb{C}$, there are polynomial functors $W : \mathbb{C}^{op} \times \mathbb{C} \longrightarrow \mathbb{C}$ and $V : \mathbb{C} \longrightarrow \mathbb{C}$, unique up to isomorphism, such that*

$$T \quad \cong \quad W \Rightarrow V$$

This motivates the following

Definition 2.2 *Let $\mathbb{C}$ be a category and $W : \mathbb{C}^{op} \times \mathbb{C} \longrightarrow C$ and $V : \mathbb{C} \longrightarrow \mathbb{C}$ functors on it.*

A paranatural *transformation $\vartheta : W \longrightarrow V$ is a family of $\mathbb{C}$-arrows $\vartheta X : WXX \longrightarrow VX$, such that for every arrow $u : X \longrightarrow Y$ in $\mathbb{C}$, the external pentagon in the following diagram*



commutes whenever the triangle on the left commutes, for all Z, z_0 and z_1 in $\mathbb{C}$.

The class of the paranatural transformations from W to V is written $\mathsf{Para}(W, V)$.

Remark. When $\mathbb{C}$ supports calculus of relations, the triangle and the quantifier over Z, z_0 and z_1 can be omitted: the condition just means that the rest if the diagram commutes up to $\subseteq$.

Proposition 2.3 *Let $\mathcal{L}$ be a polymorphic λ-calculus, and $\mathbb{C}_{\mathcal{L}}$ the cartesian closed category generated by its closed types and terms. For every type constructor $\mathcal{T}$, definable in $\mathcal{L}$, there is a bijective correspondence*

$$\mathbb{C}_{\mathcal{L}}\,(A,\ \forall X.\mathcal{T}(X)) \quad \cong \quad \mathsf{Para}(A \times W, V)$$

natural in A.

3 Characterizing fixpoints

Proposition 3.1 *Let $\mathbb{C}$ be a cartesian closed category, and F a strong endofunctor on it. Whenever the initial F-algebra M_F, resp. the final F-coalgebra N_F exist, then the following correspondences*

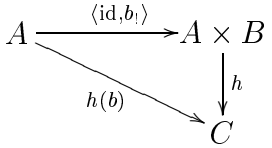
$$\mathbb{C}(A, M_F) \quad \cong \quad \mathsf{Para}\,(A \times (FX \Rightarrow X),\ X) \tag{11}$$

$$\mathbb{C}(N_F, B) \quad \cong \quad \mathsf{Para}\,(X \times (X \Rightarrow FX),\ B) \tag{12}$$

hold naturally in A, resp. B.

In well-pointed categories and strongly extensional λ-calculi, this propositoion boils down to the following "Yoneda" lemmas.

Notation. Given $h : A \times B \longrightarrow C$ and $b : 1 \longrightarrow B$, we write $h(b)$ for the result of partially evaluating h on b



where $b_!$ denotes the composite $A \xrightarrow{!} 1 \xrightarrow{b} B$.

Lemma 3.2 *For paranatural transformations*

$$\begin{aligned} \varphi_X &: A \times (FX \Rightarrow X) \longrightarrow X \\ \psi_Y &: Y \times (Y \Rightarrow FY) \longrightarrow B \end{aligned}$$

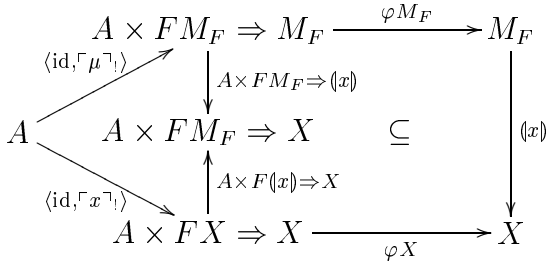
hold the equations

$$\varphi_X(\ulcorner x \urcorner) = (\!|x|\!) \cdot \varphi_{M_F}(\mu) \tag{13}$$

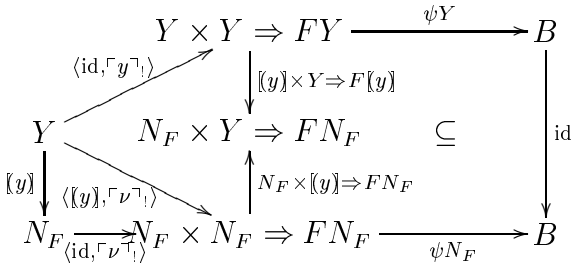
$$\psi_Y(\ulcorner y \urcorner) = \psi_{N_F}(\nu) \cdot [\![y]\!] \tag{14}$$

for all $x : FX \longrightarrow X$ *and* $y : Y \longrightarrow FY$.

While (13) follows from



(14) is obtained by chasing



In well-pointed categories, $\varphi_X : A \times (FX \Rightarrow X) \longrightarrow X$ is completely determined by its values $\varphi_X(\ulcorner x \urcorner) : A \longrightarrow C$ on all $x : FX \longrightarrow X$. Similarly, $\psi_Y : Y \times (Y \Rightarrow FY) \longrightarrow B$ is completely determined by its values on $y : Y \longrightarrow FY$.

However, in order to show that $\varphi_{M_F}(\mu)$ is generic for φ and $\psi_{N_F}(\nu)$ for ψ without the well-pointedness assumption, one needs to set up slightly different constructions.

Proof of 3.1. (11) We define maps

$$\begin{aligned}(-)' &: \mathbb{C}(A, M_F) \longrightarrow \mathsf{Para}\,(A \times (FX \Rightarrow X),\ X)\\ \texttt{build} &: \mathsf{Para}\,(A \times (FX \Rightarrow X),\ X) \longrightarrow \mathbb{C}(A, M_F)\end{aligned}$$

and show that they are inverse to each other.

Given $f : A \longrightarrow M_F$, the X-th component of f' will be

$$\begin{aligned}f'_X \ :\ A \times (FX \Rightarrow X) &\xrightarrow{f \times k} M_F \times (M_F \Rightarrow X)\\ &\xrightarrow{\varepsilon} X\end{aligned}$$

where $k : (FX \Rightarrow X) \longrightarrow (M_F \Rightarrow X)$ maps the algebra structures $x : FX \to X$ to the catamorphisms $(\!|x|\!) : M_F \to X$. Formally, k is obtained by transposing the catamorphism $(\!|\kappa|\!) : M_F \longrightarrow (FX \Rightarrow X) \Rightarrow X$ for the F-algebra κ on $(FX \Rightarrow X) \Rightarrow X$, obtained by transposing the composite

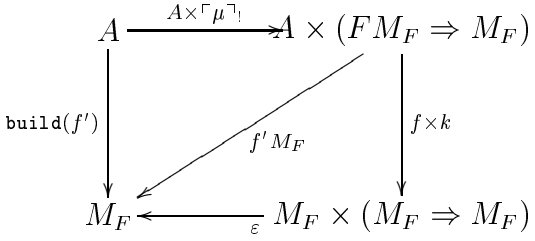
$$\begin{aligned}&(FX \Rightarrow X) \times F\,((FX \Rightarrow X) \Rightarrow X) \longrightarrow\\ &\xrightarrow{\text{(i)}} (FX \Rightarrow X) \times (FX \Rightarrow X) \times F\,((FX \Rightarrow X) \Rightarrow X)\\ &\xrightarrow{\text{(ii)}} (FX \Rightarrow X) \times F\,((FX \Rightarrow X) \times (FX \Rightarrow X) \Rightarrow X)\\ &\xrightarrow{\text{(iii)}} (FX \Rightarrow X) \times FX\\ &\xrightarrow{\text{(iv)}} X\end{aligned}$$

where arrow (i) is derived from the diagonal on $FX \Rightarrow X$, (ii) from the strength, while (iii) and (iv) are just evaluations.

Towards the definition of `build`, for a paranatural $\varphi \ :\ A \times (FX \Rightarrow X) \longrightarrow X$ take

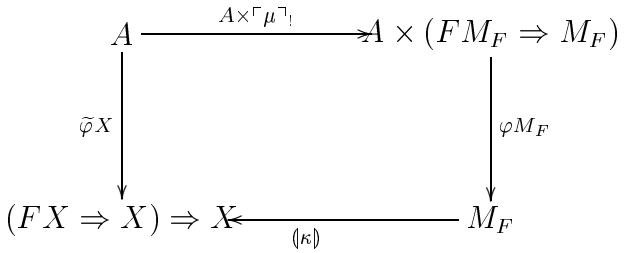
$$\begin{aligned}\texttt{build}(\varphi)\ :\ A &\xrightarrow{A \times \ulcorner \mu \urcorner} A \times (FM_F \Rightarrow M_F)\\ &\xrightarrow{\varphi M_F} M_F\end{aligned}$$

Composing the above two definitions, one gets the commutative square



Since $k \cdot \ulcorner \mu \urcorner = \ulcorner \mathrm{id}_M \urcorner$, the path around the square reduces to f, and yields $\mathtt{build}(f') = f$.

The converse $\mathtt{build}(\varphi)' = \varphi$ is the point-free version of lemma 3.2. It amounts to proving that the paranaturality of φ implies (indeed, it is equivalent) to the commutativity of



where $\widetilde{\varphi}X$ is the transpose of φX. Showing this is an exercise in cartesian closed structure. On the other hand, the path around the square is easily seen to be $\mathtt{build}(\varphi)'_X$.

Towards a proof of (12), we internalize (14) similarly like we did (13) above. The natural correspondences

$$\begin{aligned} (-)' \quad &: \quad \mathbb{C}(N_F, B) \longrightarrow \mathsf{Para}\,(X \times (X \Rightarrow FX),\ B) \\ \mathtt{process} \quad &: \quad \mathsf{Para}\,(X \times (X \Rightarrow FX),\ B) \longrightarrow \mathbb{C}(N_F, B) \end{aligned}$$

are defined

$$\begin{aligned} g'_X \ : \ X \times (X \Rightarrow FX) \quad &\xrightarrow{X \times \ell} \quad X \times (X \Rightarrow N_F) \\ &\xrightarrow{\varepsilon} \quad N_F \\ &\xrightarrow{g} \quad B \end{aligned}$$

and

$$\begin{aligned} \mathtt{process}(\psi) \ : \ N_F \quad &\xrightarrow{N_F \times \ulcorner \nu \urcorner_!} \quad N_F \times (N_F \Rightarrow FN_F) \\ &\xrightarrow{\psi N_F} \quad B \end{aligned}$$

for $g : N_F \longrightarrow B$ and $\psi : X \times (X \Rightarrow FX) \longrightarrow B$. The arrow $\ell : (X \Rightarrow FX) \longrightarrow (X \Rightarrow FX)$ maps the coalgebra structures $x : X \to FX$ to the anamorphisms $(\!|x|\!) : X \to N_F$. $\square$

4 Applications

4.1 Zip

Using correspondence (11), i.e. the maps realizing it, we can now, first of all, provide the rational reconstruction of the simple fusion from the introduction. The abstract form of the function `zipW`, leaving the type parameter X implicit, will be

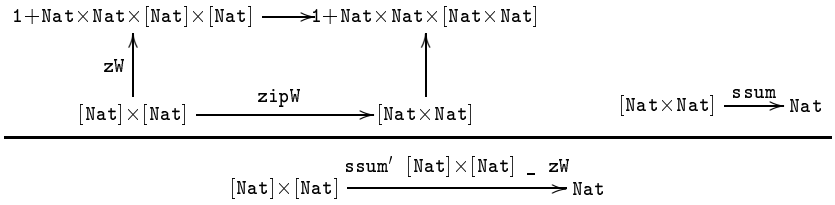
```
zipW' : ((1+Nat×Nat×X)->X) -> ([Nat]×[Nat]->X)
zipW' [m,c] (x::xs,y::ys) = c(x, y, zipW' [m,c] (xs,ys))
zipW' [m,c] (xs, ys) = m
```

While `zipW` can be recovered as the instance `zipW' [[],(::)]`, i.e. `zipW = build(zipW')`, the fusion is obtained as

```
sumzip  =  zipW' [0,‡]
```

But what is `zipW`, if it is not a catamorphism? How come that it still has a recursive definition?

It is in fact an *ana*morphism, and `ssum` · `zipW` can be simplified by process fusion as well. The scheme is this time



where

```
zW (x::xs,y::ys) = (x,y,xs,ys)
zW (xs,ys)       = One  (the element of 1)
```

induces `zipW` = ⟦`zW`⟧, whereas (leaving again the type parameter implicit)

```
ssum'     : X × (X -> 1+Nat×Nat×X ) -> Nat
ssum' x d = case d x of
              One     -> 0
              (n,m,y) -> n + m + ssum' y d
```

Calculating the conclusion this time yields

```
sumzip = ssum' _ zW
```

Finally, lifting proposition 3.1 to the category $\mathbb{C}^{\mathbb{C}}$ of endofunctors, we can derive the process fusion rule for nested data types [1]. Consider, e.g., the type constructor `Nest`, that can be defined as a fixpoint of the functor $\Psi : \mathbb{C}^{\mathbb{C}} \longrightarrow \mathbb{C}^{\mathbb{C}}$, mapping $\Psi(F) = \lambda X.1 + X \times F(X \times X)$.

The elements of the data type `Nest Nat` are the lists where the i-th entry is an element of $\mathtt{Nat}^{2^i}$. Abbreviating `Nest Nat` to $\{\mathtt{Nat}\}$, we can now define

```
zWN (x::xs,y::ys) = (x,y,fst xs,fst ys,
                           snd xs,snd ys)
zWN (xs,ys)       = One
```

where `fst` and `snd` are the obvious projections $\{\mathtt{X} \times \mathtt{X}\} \longrightarrow \{\mathtt{X}\}$, and and derive $\mathtt{zipWN} : \{\mathtt{Nat}\} \times \{\mathtt{Nat}\} \longrightarrow \{\mathtt{Nat} \times \mathtt{Nat}\}$ as $[\![\mathtt{zWN}]\!]$ again. On the other hand, working out the paranaturality condition in $\mathbb{C}^{\mathbb{C}}$ allows lifting

```
ssumN           : {Nat×Nat} -> Nat
ssumN []        = 0
ssumN (x,y)::zs = x + y + ssumN (fst zs)
                        + ssumN (snd zs)
```

to

```
ssumN'         : F(Nat) ×
                 F(X) -> 1+X×X×F(X×X) -> Nat
ssumN' F X f d = case d Nat f of
       One     -> 0
       (n,m,g) -> m + n + ssumN' FF X g dd
```

where `FF` and `dd` are the instances with `X`×`X` instead of `X`. (Here we made the type parameters explicit, to show how the functor is transformed in the recursion.) The fusion

```
sumzipN = ssumN' Nest×Nest Nat _ zWN
```

is this time

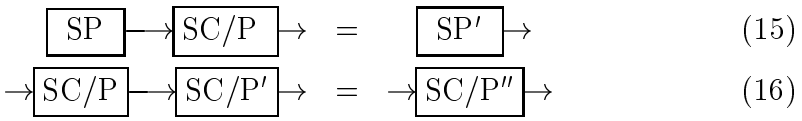
```
sumzipN                 : {Nat}×{Nat} -> Nat
sumzipN (x::xs,y::ys)   =  x + y + sumzipN (fst xs,fst ys) +
                                   sumzipN (snd xs,snd ys)
sumzipN  (xs,   ys)     =  0
```

4.2 Process fusion on streams

Whenever we have two processes, a Stream Producer (SP) and a Stream Consumer (SC), their composition can be fused into a single process, doing away with the intermediate stream.

To apply this, there is basically one requirement: the SP has to be expressed in the form of an anamorphism. This is not so much a restriction as a task to massage the expression denoting the SP into a suitable form.

If the SC is itself an SP (SC/P), we see the following two special patterns of process fusion:

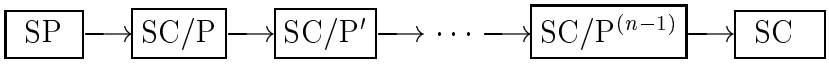


(15)

(16)

(The equality sign implies that, viewed as transformations, these steps are reversible, and in program derivation temporarily going the "wrong" way may be crucial to getting to the desired result.)

In principle, this can be cascaded, and in a pipeline of processes like



any subsegment could be fused. (The result of fusion for patterns (15) and (16) may not produce its result immediately in the required anamorphic form; to what extent this is the case, and if so whether there is an automatic technique for getting it there, requires further study.) Because of the genericity, the method is not restricted to single streams, but applies equally to multiple (parallel) streams.

It should be clear how this applies to, for example, some issues in the event-channel architecture. In particular, it makes precise how client filtering can be moved to the (proxy) server side. Depending on the requirements of the application, further forms of client processing could be moved to the server, such as data smoothing or interpolation. Likewise, queuing high-priority events can be by-passed if they would next immediately get dequeued.

Given the richness of the event channel model, practical application will require a substantial amount of work. But note that the transformations can be done already at the level of specifications; it is not necessary to have executable source code.

We give a concrete example. We apologize for how trivial it is, but we need a really simple example to avoid the exposition of the technique getting drowned out by the details. We want to fuse a producer process

```
loop
  get x
  if x > 0 then put x
end-loop
```

(which happens to be a stream filter) with a consumer process

```
loop
  get x
  put x-1
end-loop
```

(which happens to be a stream map).

We do not bother to introduce the language, as it is meant to be intuitively obvious and introduced for exposition purposes only, but it can easily be

extended with guards and local state. As a side remark, it is easy to see how the data passing by the `put` statement of the producer may be modeled at a low level as an invocation of a consumer method.

First we express the producer process as a stream function:

```
f (x::rest) = (x::f rest) if x > 0
            = f xs        otherwise
```

Introducing the auxilary function `ff` by

```
ff (x::rest) = (x, rest) if x > 0
             = ff rest   otherwise
```

we can transform the producer process into the coinductive pattern of an anamorphism for the stream data type (the coinductive data type N_F corresponding to the functor $FX = \texttt{Message} \times X$):

```
f xs = (y::f ys)
  where (y, ys) = ff xs
```

or, using the anamorphism combinator:

$$\texttt{f} \;=\; [\!(\texttt{ff})\!]$$

The consumer process, expressed as a stream function, is:

```
g (x::rest) = (x-1::g rest)
```

To make the consumption explicit, we use the function `nu` (i.e., the final coalgebra morphism $\nu : N_F \to FN_F$), functionally defined by:

```
nu (x::rest) = (x, rest)
```

Using this, we rewrite the definition of `g` into:

```
g z = (x-1::g rest)
  where (x, rest) = nu z
```

Generalizing this with an embedding transformation abstracting from `nu` into a `g'` that is parametric as in (8) (the type parameter X is left implicit as before),

```
g = g' nu
  where g' n z = (x-1::g' n rest)
          where (x, rest) = n z
```

Process fusion (10) tells us now that

$$\texttt{g} \cdot [\![\texttt{ff}]\!] \;=\; \texttt{g' ff}$$

Giving the fusion result a name, say `h`, we have

```
h = g' ff
  where g' n z = (x-1::g' n rest)
          where (x, rest) = n z
```

By specialization we remove the use of `g'` – the converse of the generalization step above:

```
h z = (x-1::h rest)
  where (x, rest) = ff z
```

We now remove the use of `ff` – the converse of the step that introduced it:

```
h (x::rest) = (x-1::h rest) if x > 0
            = h xs          otherwise
```

Compiling this into our simple process language results in:

```
loop
  get x
  if x > 0 then put x-1
end-loop
```

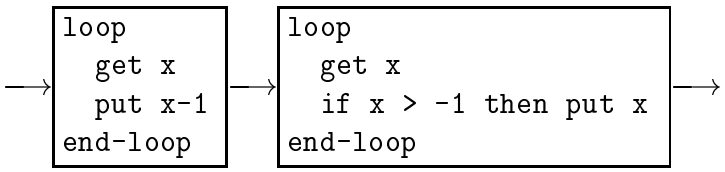
Analogously, we can fuse the producer process

```
loop
  get x
  put x-1
end-loop
```

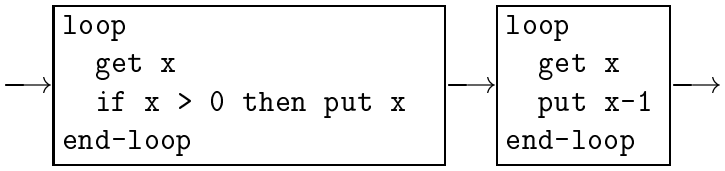
with the consumer process

```
loop
  get x
  if x > -1 then put x
end-loop
```

For brevity the details are omitted, but this happens to give the identical result as before. Since the transformations are "reversible" (forms are replaced by equivalent forms), we have also shown shown that the filter in the process composition



may leap-frog to the left position in modified form:



5 Future work

Further work will explore mechanization of the technique and its application to realistic problems. As the simple examples above have shown, manual application will be quite laborious for non-trivial cases. Particular questions to be investigated are the automation of the transformations leading to anamorphic forms.

Acknowledgements

I am indebted to Lambert Meertens for helping me carry out the detailed transformations of Section 4.2.

References

[1] R. Bird and L. Meertens. Nested datatypes. In *Proceedings MFPS '98*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

[2] P. J. Freyd. Algebraically complete categories. In A. Carboni, editor, *Proceedings of the 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer, 1991.

[3] P. J. Freyd. Structural polymorphism. *Theoretical Computer Science*, 115(1):107–129, 1993.

[4] P. J. Freyd, J.-Y. Girard, A. Scedrov, and P. J. Scott. Semantic parametricity in polymorphic lambda calculus. In *Proceedings Third Annual Symposium on Logic in Computer Science*, pages 274–279. IEEE Computer Society Press, July 1988.

[5] A. Gill, J. Launchbury, and S. Peyton-Jones. A short cut to deforestation. In *Proceedings of FPCA '93*. ACM, 1993.

[6] J. Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[7] D. Pavlovic. Maps II: Chasing diagrams in categorical proof theory. *Journal of the IGPL*, 4(2):1–36, 1996.

[8] P. Wadler. Theorems for free! In *Proceedings of FPCA '89*. ACM, 1989.