Periodic-drop-take calculus for stream transformers

Mak, R.H.

Published: 01/01/2005

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):
Abstract

Stream transformers are a formalism to specify and reason about stream processing systems. Many application specific circuits, e.g. in the area of signal processing, classify as such systems. This paper presents a two-operator calculus to reason about a specific class of stream operators, viz. the periodic stream samplers. The calculus is sound and complete and an algorithm using only a few rules is given to bring each operator sequence in canonical form. The calculus can be used to show functional correctness of any permutation circuit. It is shown how the basic building blocks of these circuits are specified using the operators, and as an example of the calculus the correctness of a non-trivial buffer is proven. In addition it is argued that this calculus can be a valuable ingredient of any larger calculus that deals with circuit components that have more complicated functionality.

1 Introduction

Stream transformers are a formalism to specify and reason about stream processing systems. There exist many types of stream processing systems, each with its own language and semantic. An excellent survey can be found in [6].

The stream transformer calculus presented in this paper is motivated by the desire to show the functional correctness of application specific circuits, defined in some high-level hardware description language like Tangram [2] or Balsa [1]. In particular we are interested in large circuits built from a small set of simple building blocks in a LEGO®-like fashion. In general both the data streams produced at the output ports of such a system and the data streams that flow along the internal channels of such a system can be expressed as transformations of the data streams presented at the input ports of the system. When we restrict the basic building blocks to ones that can either split or merge streams, each stream encountered in the system is the result of a particular interleaving of some substreams of the input streams. If we furthermore assume that the communication behavior of each basic building block is independent from the data values in its input streams and is controlled
by a (small) finite state machine, then only periodically sampled substreams can be interleaved.

Stream transformers that produce periodically sampled substreams are called periodic stream samplers, and in this paper we will restrict our attention to stream processing systems whose functionality can be specified and analyzed by this subclass of stream transformers. Even under this severe restriction fairly complicated systems can be constructed, whose functional correctness is not immediately obvious. For example, all permutation circuits can be constructed.

In this paper we present the Periodic-Drop-Take calculus, or PDT-calculus for short. This is a calculus consisting of only two generic operators (stream transformers), viz. the periodic drop operator and the periodic take operator$^1$. It will be shown that every sequence of drop and take operators denotes a periodic stream sampler. Moreover every periodic stream sampler can be represented by such a sequence, and it is possible to prove equality for any pair of operator sequences that denote the same periodic stream sampler. These proofs require only four generic rules.

This paper is organized as follows. Section 2 discusses periodic stream samplers. Section 3 introduces the PDT-calculus and shows that it is sound and complete equational theory for the class of periodic stream samplers. Section 4 introduces split-merge systems and demonstrates how the PDT-calculus is used to specify and prove the functionality of these systems. Sections 5 and 6 deal with the solution of certain systems of linear equations that arise in the analysis of split-merge systems. Finally section 7 contains some concluding remarks and suggestions for further research.

2 Periodic stream samplers

In this paper we consider stream processing systems whose computations only rearrange data items both within and between streams. Since no computation on individual data items is performed, the precise nature of the data items contained in these streams is irrelevant. It does not matter whether the items are bits, integers, or data packages with a complicated internal structure. So let $\mathbb{D}$ stand for an arbitrary domain of data values. Then a stream $A$ is a function from the natural numbers to the domain of data values, and $A(i)$ denotes the data item with rank $i$ in stream $A$. The set of all data streams with items from $\mathbb{D}$ is denoted by function space $\mathbb{N} \rightarrow \mathbb{D}$, and the set of all stream transformers by $(\mathbb{N} \rightarrow \mathbb{D}) \rightarrow (\mathbb{N} \rightarrow \mathbb{D})$.

For any stream $A: \mathbb{N} \rightarrow \mathbb{D}$ we can specify an infinite substream $B$ of $A$ by enumerating, in increasing order, the ranks of the elements of $A$ that belong to $B$. For any such enumeration $f: \mathbb{N} \rightarrow \mathbb{N}$, substream $B$ is then given by the relation $B(i) = A(f(i))$. In the sequel we are interested in periodically sampled

$^1$Note that, although they bear similar names, these operators differ from the drop and take operators used in most functional programming languages for list processing [3], because of their periodic nature. In fact these operators are more like filter operators, albeit that data items are filtered based on their ranks in the stream instead of being filtered based on their values.
substreams that are obtained by using specific functions $f$, which we will call periodic block maps.

**Definition 2.1 (Periodic block map)** A total function $f: \mathbb{N} \rightarrow \mathbb{N}$ is a periodic block map, when there exist a natural number $s \geq 1$ and an increasing function $g: [0..s] \rightarrow \mathbb{N}$, such that for $0 \leq i$ and $0 \leq t < s$

$$f(si + t) = (g(s)-g(0))i + g(t) \quad \text{(1)}$$

$$f(si + t) < (g(s)-g(0))(i+1) \quad \text{(2)}$$

Function $g$ is called a generator of $f$, number $s$ is called the generator size, and number $g(s)-g(0)$ is called the generator period. For $g$ an increasing function from an initial segment of the natural numbers to the natural numbers, $g^\triangle$ denotes the block map generated by $g$. \hfill \Box

Let $f$ be a periodic block map with generator $g$ of size $s$. From (1) it follows that $g(t) = f(t)$, for $0 \leq t \leq s$. Hence $g = f \triangle [0..s]$, i.e. $g$ is the restriction of $f$ to the initial segment $[0..s]$ of the natural numbers. It is not hard to show that when $f \triangle [0..s]$ is a generator of $f$, then $f \triangle [0..ns]$ is also a generator of $f$, for $n \geq 1$. Hence each periodic block map has infinitely many generators. Moreover, when both $g$ and $h$ are generators of $f$, with sizes $s_g$ and $s_h$ respectively, then it can be shown using results from [4] that also $f \triangle [0..s]$ with $s = \gcd(s_g, s_h)$, is a generator of $f$. So, when ordered by set inclusion, the generators of a periodic block map form a semi-infinite chain, in which the size of each generator is a multiple of the size of the least element of the chain.

**Definition 2.2 (Primitive generator)** For periodic block map $f : \mathbb{N} \rightarrow \mathbb{N}$ we define the primitive generator $f^\triangledown$ of $f$ as the smallest generator of $f$ under the $\subseteq$-ordering. In addition we define the block size, respectively block period, of any periodic block map as the size, respectively period, of its primitive generator. \hfill \Box

Note that for any periodic block map $f$ we have $(f^\triangledown)^\triangle = f$, and for any generator $g$ we have $(g^\triangledown)^\triangledown \subseteq g$. Moreover, the size of any generator $g$ of $f$ is a multiple of the block size of $f$, i.e. a multiple of the size of $f^\triangledown$.

Next observe that substreams can be defined not only by indicating the items in the original stream that should be retained in the substream, but also by indicating the items that should be discarded. Moreover, if the items to be retained can be indicated using a periodic block map, then so can the items to be discarded. This observation leads to the notion of a complement for periodic block maps.

**Definition 2.3 (Complementary block maps)** Two periodic block maps $f_1$ and $f_2$ are complementary, when they have equal block periods, and when their ranges partition the natural numbers. Since any pair of complementary block maps is completely determined by either of its components, every periodic block map $f$ that occurs as part of a complementary pair has a unique complement, which is denoted by $\overline{f}$. \hfill \Box
Indeed we have \( \overline{f} = f \), as expected for complements. Furthermore, the sum of the block sizes of two complementary periodic block maps equals their common block period. Just as periodic block maps have complements, so have their generators. Since each periodic block map has infinitely many generators, each pair of complementary block maps gives rise to infinitely many pairs of complementary generators.

**Definition 2.4 (Complementary generators)** Let \( g \) be a generator of \( f \). Then the complement of \( g \) is the unique generator \( \overline{g} \) of \( \overline{f} \) such that the ratio of the sizes of \( g \) and \( \overline{g} \) equals the ratio of the block sizes of \( f \) and \( \overline{f} \). \( \square \)

From this definition it follows that if \( g = f \sqsubset [0 .. n(sf)] \), then \( \overline{g} = \overline{f} \sqsubset [0 .. n(\overline{s}f)] \).

In other words a generator and its complement cover the same number of periods of their generated periodic block maps.

According to the definition above the identity function \( id_N \) does not have a complement, because if that complement existed, it would have an empty range, i.e. it would be the empty function \( \emptyset_N \) on the natural numbers. The empty function, however, is not a total function. In fact the identity function is the only periodic block map that has no complement.

**Property 2.1** Every periodic block map \( f \neq id_N \) has a unique complement.

**Proof.** Let \( f \) be a periodic block map other than the identity with generator \( g \) of size \( s \). We define its complement \( \overline{f} \) as follows. Since the sum \( s + \overline{s} \) of the generator sizes of any two complementary periodic block maps must equal the sum of their (common) block period we first define

\[
\overline{s} = (g(s) - g(0)) - s
\]  

(3)

Since \( g \) is increasing, it follows that \( g(s) \geq g(0) + s \), hence \( \overline{s} \geq 0 \). If \( \overline{s} = 0 \) we cannot define \( \overline{g} \), but this is no problem since in that case \( f \) is the identity function. For \( \overline{s} > 0 \) we proceed as follows. Let \( \overline{f} \) be the increasing function generated by \( \overline{g} \), where \( \overline{g} : [0 .. \overline{s}] \to N \) is the increasing function given by

\[
\{ \overline{g}(t) \mid 0 \leq t < \overline{s} \} = \{ j \mid 0 \leq j < s + \overline{s} \} \setminus \{ g(t) \mid 0 \leq t < s \}
\]

(4)

\[
\overline{g}(\overline{s}) = s + \overline{s} + \overline{g}(0)
\]

(5)

From (3) and inequality (2) it follows that \( g(t) < g(s) - g(0) = s + \overline{s} \), for \( 0 \leq t < s \). Therefore the right-hand side of equation 4 is a set consisting of precisely \( \overline{s} \) elements. Since \( \overline{g} \) must be increasing, this set uniquely defines \( \overline{g}(t) \), for \( 0 \leq t < \overline{s} \). Moreover \( \overline{g}(t) < \overline{g}(\overline{s}) - \overline{g}(0) \), and therefore \( \overline{f} \) satisfies inequality (2). \( \square \)

To get rid of this anomaly we stipulate that henceforth also \( \emptyset_N \) is a block periodic map, with block period 1 and block size 0. In accordance with this definition the primitive generator of the empty periodic block map is the function \( \emptyset_N^\uparrow : [0 .. 0] \to N \) given by \( \emptyset_N^\uparrow(0) = 1 \).

As an example of complementary periodic block maps consider the functions \( f_1 \) and \( f_2 \) defined in Figure 1. From their graphs it is obvious that \( f_1 \) and \( f_2 \) are indeed complementary block maps. The block size of \( f_1 \) is 2, and the block size of \( f_2 \) is 3. The sum of these block sizes is 5, which equals the block period of
both $f_1$ and $f_2$. The primitive generators are given by $f_1 <i>[0..2]$ and $f_2 <i>[0..3]$ respectively.

As indicated, our interest in periodic block maps is not intrinsic, but stems from the fact that they can be used to define a particular type of stream transformers, viz. the periodic stream samplers.

**Definition 2.5 (Periodic stream sampler)** A stream transformer $P: (\mathbb{N} \rightarrow D) \rightarrow (\mathbb{N} \rightarrow D)$ is a periodic stream sampler, when there exists a periodic block map $f$ such that for all streams $A: \mathbb{N} \rightarrow D$

$$(P(A))(i) = A(f(i)), \quad \text{for } 0 \leq i$$

Any substream of $A$ that can be obtained as the result of the application of a periodic stream sampler $P$ to $A$ is called a periodically sampled substream of $A$. \(\square\)

Note that in this definition periodicity refers to the way in which a stream is sampled and not to the data items in that stream, which may or may not exhibit a periodic pattern.

Since periodic stream samplers are functions (on the domain of streams $\mathbb{N} \rightarrow D$), we can compose them using standard function composition, denoted by “$\circ$”. Intuitively it should be clear that the resulting composite is again a periodic stream sampler, but using the definitions given above a formal proof is cumbersome. However, since it is an immediate consequence of the calculus developed in the next section, we here omit the proof altogether. Of course, being plain function composition, the composition of periodic stream samplers is associative. Furthermore, let $I$ be the periodic stream sampler corresponding to periodic block map $id_\mathbb{N}$. Then for all streams $A$ we have $I(A) = A$, so for any periodic stream sampler $P$ we have $I \circ P = P = P \circ I$. Hence $I$ is the unit of composition, and the set of all periodic stream samplers equipped with composition “$\circ$” is a monoid, which we shall denote by $\mathbf{M}_{PSS}$. 

Figure 1: Example of a pair of complementary periodic block maps.
Following common algebraic practice we will denote composition by juxtaposition and omit parentheses. So in the sequel we will write $\mathcal{P}\mathcal{Q}\mathcal{R}$ instead of $((\mathcal{P} \circ \mathcal{Q}) \circ \mathcal{R})$. Moreover, since streams are denoted in math font and stream transformers in calligraphic font, we will also omit parentheses when applying a stream transformer to a stream. So henceforth we will write $\mathcal{P}\mathcal{A}$ instead of $\mathcal{P}(A)$.

Next we observe that each periodic stream sampler has a unique defining periodic block map.

**Property 2.2** Let $\mathcal{P}$ be a periodic stream sampler, and let $f$ and $g$ be periodic block maps that both satisfy equation 6. Then $f = g$.

**Proof.** Choose for $\mathcal{A}$ the stream with property $A(i) = i$. Then $f(i) = A(f(i)) = (\mathcal{P}\mathcal{A})(i) = A(g(i)) = g(i)$, for all $0 \leq i$. $\square$

Hence there exists a 1-1 correspondence between the periodic block maps and the periodic stream samplers. We use this correspondence to define the size, period and fraction of a periodic stream sampler.

**Definition 2.6 (Period, size, and fraction)** For periodic stream sampler $\mathcal{P}$ we denote the unique periodic block map that corresponds to $\mathcal{P}$ by $m\mathcal{P}$. Furthermore, we define the size $s\mathcal{P}$ of $\mathcal{P}$ as the size of $m\mathcal{P}$, and the period $p\mathcal{P}$ of $\mathcal{P}$ as the period of $m\mathcal{P}$. Finally, we define the fraction $f\mathcal{P}$ of $\mathcal{P}$ as the quotient of the size and the period of $\mathcal{P}$, i.e. $f\mathcal{P} = \frac{s\mathcal{P}}{p\mathcal{P}}$. $\square$

The fraction of a periodic stream sampler is a quantity with very useful properties. First note that the fraction of a periodic stream sampler is a rational number between 0 and 1, boundaries included, and that unit $\mathcal{I}$ is the only periodic stream sampler with fraction 1. Furthermore, note that the fraction of a product of two periodic stream samplers is the product of the fractions its divisors, i.e. $f(\mathcal{P}\mathcal{Q}) = f\mathcal{P} \ast f\mathcal{Q}$. Hence the fraction of a product is at most the fraction of any of its divisors. So for a periodic stream sampler to have divisors other than the unit and itself, it must have a fraction strictly less than 1. Using fractions it is also easy to establish that $\mathcal{I}$ is the only element of $M_{PSS}$ that has an inverse, which of course is $\mathcal{I}$ itself.

### 3 PDT-calculus

In this section we present two families of operators that can be used to obtain periodically sampled substreams: the family of drop operators, and the family of take operators. The drop operators are fundamental in the sense that they form a minimal set of generators for the monoid of periodic stream samplers. The take operators are for computational convenience only. They facilitate compact specifications and short calculations.

A stream can be periodically sampled by partitioning that stream in blocks of equal length and *dropping* a data item of a fixed prescribed rank from every block. The operators that perform these actions are called drop operators.
Definition 3.1 (Drop operator) For $1 \leq l$ and $0 \leq k \leq l$ we define the stream transformer $D^k_{l+1} : (\mathbb{N} \to \mathbb{D}) \to (\mathbb{N} \to \mathbb{D})$ by

$$(D^k_{l+1}A)(li+j) = \begin{cases} A((l+1)i+j) & 0 \leq j < k \\ A((l+1)i+j+1) & k \leq j < l \end{cases}$$

Superscript $k$ is called the rank of the operator and subscript $l+1$ its period. □

From this definition it follows that operator $D^k_{l+1}$ is a periodic stream sampler with block size $l$, and block period $l+1$, whose periodic block map has primitive generator $g$ given by

$$g(j) = \begin{cases} j & 0 \leq j < k \\ j+1 & k \leq j \leq l \end{cases}$$

Note that operator $D^0_1$ is not defined, because dropping from each block of length 1 its single data item results in the empty stream, which is not infinite.

Let $P$ be a periodic stream sampler with corresponding periodic block map $f = mP$. Then the periodically sampled substream $PA$ can be obtained by dropping from each block those items whose ranks are not in the range of $f$, or equivalently those items whose ranks are in the range of $\overline{f}$. Let $g$ be a generator of $f$. Moreover, let $s$ be the size of $g$ and $\overline{s}$ the size of $\overline{g}$. Then $P$ can be represented by the drop operator sequence

$$D^{g(0)}_{s+1}P^{g(1)}_{s+2} \cdots D^{g(\overline{s}-1)}_{s+\overline{s}}$$

(7)

In this representation of $P$ the highest ranked item of each block, corresponding to the rightmost drop operator, is dropped first. Then the next highest is dropped, but since already one item per block has been dropped, the block period is reduced by one. When $g$ is a primitive generator, then the period $s+\overline{s}$ of the rightmost operator of the sequence equals the period of $P$, otherwise it is a multiple of the period of $P$. A representation that corresponds to a primitive generator is called a canonical form.

Next consider what happens when we construct the drop operator sequence representation of the stream transformer $I$. In that case $\overline{s} = 0$, so we obtain the empty sequence. Since the empty sequence is indeed both the left and the right unit of sequence catenation, this is exactly as it should be. Thus we see that $I$ and the drop operators are sufficient to generate all periodic stream samplers.

Property 3.1 Every periodic stream sampler can be written as a, possibly empty, sequence of drop operators.² By convention the empty sequence is denoted by $I$.

Moreover, we have encountered the first rule of the PDT-calculus. It establishes the fact that operator $I$ is the unit operator.

²In this property and in the remainder of this paper operator sequences are assumed to be of finite length.
Rule 3.1 (Unit) For $1 \leq l$ and $0 \leq k \leq l$ we have

$$II = I$$
$$ID_{l+1}^k = D_{l+1}^k = D_{l+1}^kI$$  \hspace{1cm} (8)

In the sequel we will show that the converse of property 3.1 also holds, viz. that every sequence of unit and drop operators is also a periodic stream sampler. We do so by showing that any such operator sequence can be rewritten to a unique canonical form. Since in particular the catenation of the drop operator sequence representations of any pair of periodic stream samplers can be rewritten to canonical form, it immediately follows that the composition of two periodic stream samplers is again a periodic stream sampler, as was already claimed in the previous section.

As it turns out, transforming an arbitrary sequence of drop operators to canonical form can be done in three stages. In each stage of the transformation a rule of the PDT-calculus is required that establishes a characteristic property of formula 7. The first of these characteristics is concerned with the periods of the operators in the sequence.

Definition 3.2 (Period-consecutive) A sequence of $m \geq 1$ drop operators $D_{l_1}^k \ldots D_{l_m}^k$ is period-consecutive, when there exists a natural number $l \geq 1$ such that $l_i = l+i$, for $1 \leq i \leq m$. □

An arbitrary drop operator sequence can be made period-consecutive by a technique called drop expansion. Consider the substream obtained by dropping the data item with rank $k$ from every block of length $l+1$. Obviously the same substream is obtained, when we drop $m$ data items with ranks $k, (l+1)+k, \ldots, (l+1)(m-1)+k$ from every block of length $(l+1)m$ in the following manner. First we drop from every block of length $(l+1)m$ the item at location $(l+1)(m-1)+k$. From each resulting block of length $(l+1)m-1$ we subsequently drop the item at location $(l+1)(m-2)+k$. Repeating this procedure another $m-2$ times we obtain the required substream. Hence we have the following rule

$$D_{l+1}^k = D_{lm+1}^k \ldots D_{lm+2}^k \ldots D_{lm}(l+1)(m-1)+k$$  \hspace{1cm} (9)

for $1 \leq l, m$, and for $0 \leq k \leq l$. The operator sequence on the right-hand side of this equation is called the $m$-fold expansion of the operator on the left-hand side. Vice-versa the left-hand side is called the $m$-fold contraction of the right-hand side.

Note that the $m$-fold expansion of a drop operator yields a period-consecutive sequence of drop operators. Since a single drop operator is already a period-consecutive sequence of drop operators, viz. of length 1, drop expansion seems of little use. This is a misconception, however, because proper application can transform any pair of drop operators $D_{l+1}^k \ldots D_{l+1}^p$ into a period-consecutive sequence. To see this, notice that the last drop operator of the $q$-fold expansion $D_{l+1}^k \ldots D_{l+1}(q-1)+k$ of $D_{l+1}$ has period $(l+1)q$. Moreover, the first drop operator of the $(l+1)$-fold expansion $D_{q(l+1)+1}^p \ldots D_{q(l+1)(l+1)}^p$ of $D_{q+1}$ has period...
$q(l+1)+1$. Hence the catenation of both expansions is a period-consecutive sequence of drop operators of length $l+q+1$. So we have

**Property 3.2** Using formula 9 every pair of drop operators can be rewritten to a period-consecutive sequence of drop operators. □

Now that we know how to transform pairs of drop operators to period-consecutive form, we can address the question how to transform an arbitrary sequence of drop operators of length at least three to period-consecutive form. Suppose we are confronted with a such sequence of length $m+1$ and that we somehow (e.g. by recursion) have managed to transform its prefix of length $m$ to period-consecutive form. Then we are again left with a pair. The first component of this pair is the period-consecutive transform of the prefix, and the second component of the pair is the last drop operator of the original sequence. On this pair we would like to apply the same technique as before, i.e. expanding both components in such a way that the last period of the expansion of the first component is one less than the first period of the expansion of the second component. Therefore we need a rule that can expand the prefix-transform, i.e. an arbitrary period-consecutive sequence of drop operators.

**Rule 3.2 (Drop expansion/contraction)** For $1 \leq l, m, n$, and for $0 \leq k_1, \ldots, k_n < l+n$, we have

$$D_{l+1}^{k_1} \cdots D_{l+n}^{k_n} = \left( D_{m+1}^{k_1} \cdots D_{m+n}^{k_n} \right) \left( D_{m+1}^{(l+n)+k_1} \cdots D_{m+2n}^{(l+n)+k_n} \right) \cdots \left( D_{m+1}^{(l+n)+(m-1)+k_1} \cdots D_{m+mn}^{(l+n)+(m-1)+k_n} \right)$$

The right-hand side of this equation is called the $m$-fold expansion of the left-hand side. Vice-versa the left-hand side is called the $m$-fold contraction of the right-hand side. □

Note that taking $n = 1$ in this rule yields formula 9. So we may conclude

**Property 3.3** Using drop expansion every sequence of drop operators can be rewritten to a period-consecutive sequence of drop operators. □

The second characteristic of formula 7 is concerned with the ranks of the drop operators.

**Definition 3.3 (Rank-increasing)** A sequence of $m \geq 1$ drop operators $D_{l_1}^{k_1} \cdots D_{l_m}^{k_m}$ is rank-increasing, when $k_i < k_j$, for all $1 \leq i < j \leq m$. □

Sequences that are both rank-increasing and period-consecutive merit a special name.

**Definition 3.4 (Pseudo-canonical form)** A non-empty sequence of operators is a pseudo-canonical form, when it is the sequence consisting of the single unit operator, or when it is a period-consecutive, rank-increasing sequence of drop operators. □
Pseudo-canonical forms are to periodic stream samplers what generators are to periodic block maps. Each periodic stream sampler has infinitely many pseudo-canonical forms. Nevertheless they are in some sense unique.

**Property 3.4** For \( m \geq 0 \) and \( s \geq 1 \), let \( P \) be a periodic stream sampler with fraction \( f_P = \frac{s}{s+m} \). If the period of \( P \) is a divisor of \( s+m \), then \( P \) has a unique pseudo-canonical form \( D_{k_1} \cdots D_{k_m} \) of length \( m \).

**Proof.** Since the period of \( P \) is a divisor of \( s+m \), there exists a unique generator \( g \) of \( mP \) with size \( s \) and period \( g(s) - g(0) = s+m \). The drop operator sequence corresponding to \( g \) has the required shape with \( k_i = \overline{g}(i-1) \), for \( 1 \leq i \leq m \). □

Note that if a drop operator sequence is rank-increasing this property is preserved under application of both expansion and contraction. This implies that we need at least one additional rule that can be used to make drop operator sequences rank-increasing. This rule is the drop exchange rule.

**Rule 3.3 (Drop exchange)** For \( 1 \leq l \), and for \( 0 \leq k \leq h \leq l \), we have

\[
D_{l+1}^k D_{l+2}^h = D_{l+1}^h D_{l+2}^{k+1}
\]

(10)

□

It should be obvious that by repeated application of the drop exchange rule, every period-consecutive drop operator sequence can be transformed into one that is also rank-increasing.

**Property 3.5** Using the drop exchange rule any period-consecutive sequence of drop operators can be transformed into another period-consecutive sequence that is also rank-increasing, i.e. into a pseudo-canonical form. □

As an aside we note that the drop exchange rule reveals a specific property of monoid \( M_{PSS} \). Taking \( k = h \) we obtain

\[
D_{l+1}^h D_{l+2}^h = D_{l+1}^h D_{l+2}^{h+1}
\]

from which it follows that monoid \( M_{PSS} \) is not cancellative.

Finally, the third characteristic of representation 7, is one that it only possesses when it is constructed using a primitive generator.

**Definition 3.5 (Primitive)** A period-consecutive sequence of drop operators is primitive, when it is not the \( m \)-fold expansion of another drop operator sequence, for some \( m \geq 2 \). □

Drop operator sequences that posses all three characteristics are unique and are called canonical forms.

**Definition 3.6 (Canonical form)** A non-empty finite sequence of operators is a canonical form, either when it is the sequence consisting of the single unit operator, or when it is a period-consecutive, rank-increasing, primitive sequence of drop operators. □

---

3By performing the well-known insertion sort algorithm on the sequence of ranks.

4A monoid \( M \) is called cancellative, when \( ab = ac \Rightarrow b = c \), for all \( a, b, c \in M \).
By definition pseudo-canonical forms can be made primitive using drop contraction. Since drop contraction can neither destroy the fact that a sequence is rank-increasing nor the fact that a sequence is period-consecutive, the result is a canonical form.

**Property 3.6** Using drop contraction every pseudo-canonical form can be transformed into a canonical form. □

Combining properties 3.3, 3.5, and 3.6 we obtain the result we are after.

**Theorem 3.1** Every non-empty finite sequence of operators can be rewritten to canonical form using the unit rule, drop expansion, drop contraction and drop exchange. □

In logic terms we have established that PDT-calculus is complete, because we can establish the equivalence of any pair of operator sequences by transforming them to their unique canonical forms.

In algebraic terms we have established that $M_{PSS} = \langle G \mid E \rangle$, where $G$ is the set consisting of the unit operator and all drop operators, and $E$ is the set of all equations obtained by instantiation of the unit rule, the drop expansion/contraction rule, and the drop exchange rule.

As a first example of PDT-calculus we bring the drop operator sequence $D^3_0 D^3_1 D^1_3$ onto canonical form. We derive

$$D^3_0 D^3_1 D^1_3$$

$$= \{ \text{3-fold drop expansion of } D^3_0 \text{ and 2-fold drop expansion of } D^1_3 \}$$

$$D^3_0 D^3_1 D^3_6 D^3_7 D^1_3$$

$$= \{ \text{2-fold drop expansion of } D^3_0 \cdots D^3_7 \text{ and 8-fold drop expansion of } D^1_3 \}$$

$$D^3_0 D^3_1 D^1_8 D^1_{10} D^1_{11} D^1_{12} D^1_{13} D^1_{14} D^1_{15} D^1_{16} D^1_{17} D^1_{18} D^1_{19} D^1_{20} D^1_{21} D^1_{22} D^1_{23} D^1_{24}$$

$$= \{ \text{drop exchange: } D^3_0 D^1_{10} \}$$

$$D^3_0 D^3_1 D^3_6 D^1_8 D^1_{10} D^1_{11} D^1_{12} D^1_{13} D^1_{14} D^1_{15} D^1_{16} D^1_{17} D^1_{18} D^1_{19} D^1_{20} D^1_{21} D^1_{22} D^1_{23} D^1_{24}$$

$$= \{ \text{drop exchange: } D^3_1 D^1_{15} \}$$

$$D^3_0 D^3_1 D^3_6 D^3_9 D^3_{10} D^1_{11} D^1_{12} D^1_{13} D^1_{14} D^1_{15} D^1_{16} D^1_{17} D^1_{18} D^1_{19} D^1_{20} D^1_{21} D^1_{22} D^1_{23} D^1_{24}$$

$$= \{ \text{9 drop exchange: } D^3_2 \cdots D^3_{17} \}$$

$$D^3_0 D^3_1 D^3_6 D^3_9 D^3_{10} D^3_{11} D^3_{12} D^3_{13} D^3_{14} D^3_{15} D^3_{16} D^3_{17} D^3_{18} D^3_{19} D^3_{20} D^3_{21} D^3_{22} D^3_{23} D^3_{24}$$

$$= \{ \text{8 drop exchange: } D^3_{10} \cdots D^3_{18} \}$$

$$D^3_0 D^3_1 D^3_6 D^3_9 D^3_{10} D^3_{11} D^3_{12} D^3_{13} D^3_{14} D^3_{15} D^3_{16} D^3_{17} D^3_{18} D^3_{19} D^3_{20} D^3_{21} D^3_{22} D^3_{23} D^3_{24}$$

$$= \{ \text{7 drop exchange: } D^3_{12} \cdots D^3_{19} \}$$

$$D^3_0 D^3_1 D^3_6 D^3_9 D^3_{10} D^3_{11} D^3_{12} D^3_{13} D^3_{14} D^3_{15} D^3_{16} D^3_{17} D^3_{18} D^3_{19} D^3_{20} D^3_{21} D^3_{22} D^3_{23} D^3_{24}$$

$$= \{ \text{6 drop exchange: } D^3_{14} \cdots D^3_{20} \}$$

$$D^3_0 D^3_1 D^3_6 D^3_9 D^3_{10} D^3_{11} D^3_{12} D^3_{13} D^3_{14} D^3_{15} D^3_{16} D^3_{17} D^3_{18} D^3_{19} D^3_{20} D^3_{21} D^3_{22} D^3_{23} D^3_{24}$$

$$= \{ \text{4 drop exchange: } D^3_{12} \cdots D^3_{13} \}$$

$$D^3_0 D^3_1 D^3_6 D^3_9 D^3_{10} D^3_{11} D^3_{12} D^3_{13} D^3_{14} D^3_{15} D^3_{16} D^3_{17} D^3_{18} D^3_{19} D^3_{20} D^3_{21} D^3_{22} D^3_{23} D^3_{24}$$

$$= \{ \text{3 drop exchange: } D^3_{16} \cdots D^3_{22} \}$$
The calculus developed thus far has several disadvantages. Periodic samplers that take only a single element from a large block are represented by lengthy drop operator sequences. Also, as we have seen above, rewriting a drop operator sequence to canonical form can be a tedious enterprise. Therefore we will now introduce a second operator into our calculus that alleviates this situation.

**Definition 3.7 (Take operator)** For \( l \geq 1 \) and \( 0 \leq k \leq l \) we define the stream transformer \( T_{k+l}^{l+1} : (\mathbb{N} \rightarrow \mathbb{D}) \rightarrow (\mathbb{N} \rightarrow \mathbb{D}) \) by

\[
(T_{k+l}^{l+1}A)(i) = A((l+1)i+k)
\]

\( \square \)

From this definition it follows that operator \( T_{k+l}^{l+1} \) is a periodic stream sampler with block size 1 and block period \( l+1 \), whose periodic block map has primitive generator \( g \) given by \( g(0) = k \) and \( g(1) = l+1 \).

Obviously, every take operator can be replaced by a sequence of drop operators. Instead of taking one value out of every \( l+1 \) values, one drops the other \( l \) values. In other words, the periodic block map of \( T_{k+l}^{l+1} \) is the complement of the periodic block map of \( D_{k+l}^{l+1} \). Hence we have the following rule that allows us to eliminate or introduce take operators.

**Rule 3.4 (Complement)** For \( l \geq 1 \) and \( 0 \leq k \leq l \) we have

\[
T_{l+1}^k = \begin{cases} 
D_{l+1}^l & k = 0 \\
D_2^0 \cdots D_{k+l}^{k+1-l} & 0 < k < l \\
D_2^0 \cdots D_{l+1}^{l-1} & k = l 
\end{cases}
\]

\( \square \)

So far we have given a minimum set of rules necessary to transform an arbitrary sequence of take and drop operators to canonical form. As with most calculi this calculus can be made more effective, in the sense that there exist shorter proofs, by additional rules.

The first such rule is concerned with the elimination (or introduction) of a drop operator in the context of a take operator. Using the complement rule and the drop exchange rule it is easy to show that the following rule is valid.
Rule 3.5 (Drop elimination/introduction) For $l \geq 1$ and $0 \leq h \leq l$ we have
\[
T_{i+1}^{h}D_{i+2}^{k} = \begin{cases} 
T_{i+2}^{h} & 0 \leq h < k \\
T_{i+2}^{h+1} & k \leq h \leq l 
\end{cases}
\] (12)

The second rule is concerned with the composition of a pair of arbitrary take operators. The result of such a composition is again a take operator, whose period is the product of the periods of the two operators, and whose rank is a linear combination of the ranks of the two operators, as is easily shown by application of definition 3.7.

Rule 3.6 (Take composition) For all $l, q \geq 1$ and for all $0 \leq k \leq l$ and $0 \leq p \leq q$ we have
\[
T_{i+1}^{k}T_{q+1}^{p} = T_{(l+1)(q+1)}^{(q+1)k+p}
\] (13)

We conclude this section with an example that shows how take operators are used to obtain compact specifications. For $1 \leq l$ let function $\pi$ be a permutation of the set $\{k \mid 0 \leq k \leq l\}$. Then a stream processing system with input stream $A$ and output stream $B$ that satisfies for $0 \leq i$ and $0 \leq k \leq l$ the relation
\[
B((l+1)i+k) = A((l+1)i+\pi(k))
\]
is called a block permutator. Using take operators this specification can be compactly written as
\[
T_{i+1}^{k}B = T_{i+1}^{\pi(k)}A
\]
If $\pi(k) = l-k$, then the system is called a block reverser with block size $l+1$.

Note that a specification in terms of drop operators would have required a drop operator sequence of length $l$ at each side of the equation. Examples of how take operators may shorten the length of a proof are given in the next section.

4 Split-merge systems

In this section we apply the PDT-calculus developed in the previous section to verify the functional correctness of systems built from two types of simple components, viz. split components and merge components. Together with the one-place buffer component these components are the main building blocks for buffers of arbitrary structural complexity (see [5]). Because one-place buffer components implement the unit stream transformer $I$, their presence or absence is irrelevant for the functional correctness of a system. However, they do

\footnote{In contrast to showing this by application of the complement rule and rewriting the resulting drop operator sequences to canonical form, which is quite cumbersome.}
influence the performance of a system. In fact, removal of a single one-place buffer component from a properly behaving system may cause deadlock. Nevertheless we will ignore one-place buffer components, because performance is not an issue in this paper.

A split component is a component with a single input stream \( A \) and two output streams \( C \) and \( D \). For \( 1 \leq l \) and \( 0 \leq k \leq l \) Figure 2 contains both the program text, the circuit diagram, and the functional specification of split component \( \text{Split}^k_{l+1} \).

\[
\text{Split}^k_{l+1} = \quad \begin{array}{l}
\text{proc} \ (\text{in} \ a, \text{out} \ c,d).
\end{array}
\]

\[
\begin{array}{c}
|\var x \\
\langle (a?x; d!x)^k \\
; (a?x; c!x) \\
; (a?x; d!x)^{l-k} \\
\rangle^* \\
\|
\end{array}
\]

\[
C = T^k_{l+1} A \\
D = D^k_{l+1} A
\]

Figure 2: Specification, diagram, and program text of component \( \text{Split}^k_{l+1} \).

Of these three descriptions the program text is the one that fully describes the component. Circuit diagrams mainly serve to illustrate the structure of large systems build from these components and functional specifications are used to determine the functionality of larger systems. The program text resembles a procedure declaration such as one might encounter in any higher level programming language. It consists of a heading and a body. The heading specifies that the component is named \( \text{Split}^k_{l+1} \) and that it has an input port \( a \) along which it receives the stream of input values \( A \) from its environment and output ports \( c \) and \( d \) along which its sends the streams of output values \( C \) and \( D \) to its environment. The body consists of a declaration part and a command. The command defines the order of the communication events in which the component is involved. It expresses that \( \text{Split}^k_{l+1} \) is capable of an infinite repetition, denoted by the Kleene star ‘*’. In each iteration of this infinite repetition, the component performs \( l+1 \) pairs of communication actions, each pair consisting of an input action followed by an output action. The input action is always \( a?x \), and the output action is almost always \( d!x \), apart from the output action with rank \( k \) which is \( c!x \). Note that in the diagram the output port that produces the selected input is marked with a black dot. From this elaborate description of the behavior of component \( \text{Split}^k_{l+1} \) it should be obvious that it satisfies the functional specification \( C = T^k_{l+1} A \) and \( D = D^k_{l+1} A \).

Besides components that split streams we will consider components that merge streams. Figure 3 contains the program text, the circuit diagram, and the functional specification of merge component \( \text{Merge}^k_{l+1} \).

Component \( \text{Merge}^k_{l+1} \) is the opposite of component \( \text{Split}^k_{l+1} \) in the sense that for every \( l+1 \) outputs it inputs the output with rank \( k \) along port \( e \) and the
remaining ones along port $f$. It can be verified by inspection of the program texts that connecting a merge component to the corresponding split component in such a way that the black dots match will yield a system whose output stream equals its input stream, i.e. a buffer. Also note that connecting the outputs of the split component to inputs of the merge component in the opposite way (dot to non-dot) will result in a system that deadlocks after it has accepted the first data item from its input stream$^6$.

As a first example of an application of the PDT-calculus we consider the construction of a block reverser. Recall that for $l \geq 1$ a system with single input stream $A$ and single output stream $B$ is a block reverser of order $l+1$, when $T_{l+1}^k B = T_{l+1}^{l-k} A$. Figure 4 shows how a block-reverser of order $l+1$ can be constructed from a block-reverser of order $l$.

For $l > 1$ the correctness of this construction follows from two simple calculations. To begin with it follows directly from the functional specification of the split and merge component that for outputs with rank $k = 0$ we have

$$T_{l+1}^0 B = E = T_{l+1}^l A$$

(14)

Next assume that $REVERSE_l$ is indeed a block reverser of order $l$, i.e. it satisfies $T_l^p D = T_l^{l-1-p} C$, for $0 \leq p \leq l-1$. Then, using the drop elimina-

$^6$Assuming synchronous communication.
tion/introduction rule, we find for outputs with rank \( k \), where \( 0 < k \leq l \), that

\[
T_{i+1}^kB = T_{i}^{k-1}D_{i+1}^0B = T_{i}^{k-1}D = T_{i}^{l-k}C = T_{i}^{l-k}D_{i+1}^0A = T_{i+1}^{l-k}A
\]

(15)

For \( l = 1 \) a similar construction can be used, provided we make a proper choice for \( \text{REVERSE}_1 \). Obviously, reversing blocks of length one amounts to doing nothing, so \( \text{REVERSE}_1 \) must be a system with functional specification \( D = C \), i.e. a buffer. Taking \( l = 1 \) in equation 14 and replacing equation 15 by

\[
T_2^1B = D_2^0B = D = C = D_2^1A = T_2^0A
\]

(16)

then proves the functional correctness of the system. Note that dropping component \( \text{REVERSE}_1 \) and connecting stream \( C \) directly to \( D \) will likewise produce a functionally correct block reverser. As argued above, in case of synchronous communication, such a system will exhibit a deadlock.

As a second example of an application of the PDT-calculus to split-merge systems we consider the system \( S \) given in Figure 5. It is a SISO (single-input single-output) system that is obtained by composition, denoted by “||” in the program text, of four split and four merge components. These components are connected by directed channels that run from an output port of one component.
to an input port of another component. In the program text these connections are made explicit through instantiation of port names with the appropriate channel names in the generic split and merge components. Furthermore it is assumed that the channels have no storage capacity, which means that communication is synchronous. A good impression of the system structure is obtained by the corresponding diagram. In comparison with Figure 4 we have in this diagram omitted the circular stream symbols from all internal channels, both to keep the diagram simple and to stress the fact that channels have no storage capacity.

In the sequel we shall demonstrate that system \( S \) is in fact a buffer. As a first step we associate a specification of the form \( XA \) with each internal and external stream of the system. For instance the stream that flows along internal channel \( c \) is given by \( XcA \), etc. For the output stream in particular we have \( B = X_bA \), so demonstrating that \( S \) is a buffer means that we have to show that \( X_b = I \). Next we consider the components of the system. From their functional specifications we obtain for every component two equations, each of which relates an input stream of that component to an output stream of that component. Moreover, we add an equation for the input stream, viz. \( Xa = IA \). Since \( A \) is the only stream that occurs in these equations, and since \( A \) occurs on both sides of each equation, we can eliminate \( A \), whereafter we are left with equations that contain only stream transformers. Thus we obtain for any system \( S \) a set of equations \( E_{sv}(S) \), where the subscript indicates that we have introduced a single variable per stream. For system \( S \) defined in Figure 5 we obtain the following set of seventeen equations with thirteen unknown stream transformers.

\[
\begin{align*}
X_a &= I \\
X_{dl} &= D^0_3 X_a \\
X_{dr} &= T^0_3 X_a \\
X_{el} &= D^2_0 X_c \\
X_{er} &= T^2_0 X_c \\
X_{fl} &= D^0_2 X_c \\
X_{fr} &= T^0_2 X_c \\
X_{gl} &= D^1_2 X_c \\
X_{gr} &= D^3_0 X_c
\end{align*}
\]

By substitution most of the unknown stream transformers can be eliminated. Only \( X_b \) and \( X_c \) cannot be eliminated, since they do not occur at the left-hand side of any equation. After we have performed the elimination process, we are left with the following set of six equations and two unknown periodic stream samplers.

\[
\begin{align*}
T_4^4 T_3^0 &= T_3^2 D^0_2 X_b \\
T_2^3 T_3^0 &= T_3^0 T_2^0 X_b \\
D_4^3 D_3^0 &= D_2^2 X_c \\
D_2^3 D_2^0 X_b &= D_2^1 X_c \\
D_3^0 T_2^0 X_b &= T_2^1 X_c
\end{align*}
\]

In the next section we will show under which conditions and in what manner such sets of equations can be solved. Here we simply verify that \((X_b, X_c) = \)
$(I, D_0^5 D_6^5)$ is a solution. Since all split and merge components are deterministic and neither create nor destroy data items, each data item follows a unique path through the system. This means that if the set $E_{sv}(S)$ has a solution, it is unique. Substituting the postulated solution in the above equations we obtain the set of equalities

\[
T_1^3D_3^0 = T_3^2D_2^0 \quad D_1^3D_3^0 = D_3^2D_6^0 D_6^5 \quad D_2^2D_2^0 = D_1^4 D_5^0 D_6^5 \\
T_2^0T_3^0 = T_3^0T_2^0 \quad D_2^0 T_3^0 = T_4^2 D_6^0 D_6^5 \quad D_3^0 T_2^0 = T_2^1 D_3^0 D_6^5
\]

the validity of which we have to verify. The standard approach is by reduction to canonical form, which leads to trivial but lengthy calculations. Shorter proofs using the additional rules that have been established for the take operators are given in Table 1.
<table>
<thead>
<tr>
<th>1. $T_3^0D_3^0 = T_3^2D_2^0$</th>
<th>2. $T_3^0T_3^0 = T_3^2T_2^0$</th>
<th>3. $D_3^0D_3^0 = D_4^2D_5^0D_6^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_3^3D_3^0$</td>
<td>$T_3^0T_3^0$</td>
<td>$D_3^4D_3^0$</td>
</tr>
<tr>
<td>= { D-Exp }</td>
<td>= { T-Elim }</td>
<td>= { D-Exp }</td>
</tr>
<tr>
<td>$T_3^4D_5^3$</td>
<td>$T_6^0$</td>
<td>$D_3^4D_5^0D_6^0$</td>
</tr>
<tr>
<td>= { D-Elim }</td>
<td>= { T-Intro }</td>
<td>= { D-Exch }</td>
</tr>
<tr>
<td>$T_3^4D_6^3$</td>
<td>$T_3^0T_2^0$</td>
<td>$D_4^0D_5^4D_6^3$</td>
</tr>
<tr>
<td>= { D-Elim }</td>
<td></td>
<td>= { D-Exch }</td>
</tr>
<tr>
<td>$T_6^5$</td>
<td></td>
<td>$D_4^0D_5^4D_6^0$</td>
</tr>
<tr>
<td>= { T-Intro }</td>
<td></td>
<td>= { D-Exch }</td>
</tr>
<tr>
<td>$T_3^2T_2^1$</td>
<td></td>
<td>$D_4^2D_5^0D_6^0$</td>
</tr>
<tr>
<td>= { Compl }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_3^2D_2^0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. $D_2^0T_3^0 = T_4^2D_5^0D_6^5$</th>
<th>5. $D_3^2D_2^0 = D_4^1D_5^0D_6^5$</th>
<th>6. $D_2^0T_2^0 = T_3^1D_5^0D_6^5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_2^0T_3^0$</td>
<td>$D_3^2D_2^0$</td>
<td>$D_3^0T_2^0$</td>
</tr>
<tr>
<td>= { Compl }</td>
<td>= { D-Exp }</td>
<td>= { Compl }</td>
</tr>
<tr>
<td>$T_2^4T_3^0$</td>
<td>$D_3^2D_4^1D_5^2D_6^4$</td>
<td>$D_3^0D_2^1$</td>
</tr>
<tr>
<td>= { T-Elim }</td>
<td>= { D-Exch }</td>
<td>= { D-Exp }</td>
</tr>
<tr>
<td>$T_3^0$</td>
<td>$D_3^0D_4^1D_5^2D_6^4$</td>
<td>$D_3^0D_4^1D_5^3D_6^5$</td>
</tr>
<tr>
<td>= { D-Intro }</td>
<td>= { D-Exch }</td>
<td>= { D-Exch (2x) }</td>
</tr>
<tr>
<td>$T_3^2D_5^0$</td>
<td>$D_3^0D_4^1D_5^3D_6^4$</td>
<td>$D_3^0D_4^1D_5^3D_6^5$</td>
</tr>
<tr>
<td>= { D-Intro }</td>
<td>= { D-Exch }</td>
<td>= { D-Contr }</td>
</tr>
<tr>
<td>$T_3^2D_6^0D_6^0$</td>
<td>$D_3^0D_4^1D_5^3D_6^5$</td>
<td>$T_2^1D_5^1D_5^0D_6^0$</td>
</tr>
<tr>
<td></td>
<td>= { D-Exch (2x) }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= { D-Contr }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$D_3^1D_4^1D_5^3D_6^5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= { D-Contr }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$D_3^1D_4^1D_5^0D_6^0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= { Compl }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T_2^1D_5^1D_5^0D_6^0$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Correctness proofs for system S.
5 Solving equations for buffer systems

In the previous section we have shown how to associate a set of equations $E_{sv}(S)$ with any SISO system $S$ consisting of split and merge components. Using the PDT-calculus it is possible to establish for any assignment of periodic stream samplers to the variables of an equation whether that assignment satisfies the equation. However, this still makes finding a solution for the set of equations $E_{sv}(S)$ a matter of trial-and-error. In case of linear equations with a single unknown we can do better. For those equations it is in fact possible to calculate the set of all solutions.

So consider a single linear equation of the form

$$P X = Q$$ (17)

where coefficient $P$ and right-hand side $Q$ are constant periodic stream samplers, and $X$ is the unknown. This problem is also known as the division problem for monoid $M_{PSS}$. Since the fraction of a product is at most the fraction of any of its divisors, $f_Q \leq f_P$ is a necessary condition for this equation to have a solution. Moreover, since $I$ is the only periodic stream sampler with fraction 1, $X = I$ is the only possible solution when $f_Q = f_P$. If in that case $P = Q$, then $I$ is indeed a solution. Otherwise there is no solution. Without further knowledge of $P$ and $Q$ this is all we can say about the solutions of equation 17.

So let us assume that $P$ and $Q$ have been transformed to canonical form. Then equation 17 takes on the shape

$$D_{s+1}^{k_1} \cdots D_{s+m}^{k_m} X = D_{l+1}^{l_1} \cdots D_{l+n}^{l_n}$$ (18)

Computing the fractions of the periodic stream samplers at both sides, we find that this equation has no solution, when $\frac{l}{l+n} > \frac{s}{s+m}$. On the other hand, when $\frac{l}{l+n} \leq \frac{s}{s+m}$, there can be zero, one or more solutions. As an example consider the equation

$$D_{l+1}^p X = D_{l+1}^q D_{l+2}^r$$

Calculating fractions we derive $f_X = \frac{l+1}{l+2}$. Because the period of the stream transformer on the right-hand side is a divisor of $l+2$, it follows by property 3.4 that $X = D_{l+2}^x$, for some value of $x$. Hence, on account of the drop-exchange rule, we find the following solution sets:

$$\{ X \mid D_{l+1}^p X = D_{l+1}^{q'} D_{l+2}^{r'} \} = \begin{cases} \{ D_{l+2}^r \}, & p = q \land p+1 \neq r \\ \{ D_{l+2}^{p'}, D_{l+2}^r \}, & p = q \land p+1 = r \\ \{ D_{l+2}^r \}, & p \neq q \land p+1 = r \\ \emptyset, & p \neq q \land p+1 \neq r \end{cases}$$

Note that this solution method relies on the fact that leftmost drop operators on both sides of the equation have the same period, which makes it possible to apply the drop-exchange rule. A similar result holds when the coefficient and the right-hand side are arbitrary pseudo-canonical forms starting with a drop operator of the same period.
Theorem 5.1 For integers $m,n,s$ such that $1 \leq m \leq n$ and $1 \leq s$, and for increasing sequences of ranks $k_1 < \cdots < k_m$ and $l_1 < \cdots < l_n$, we have

$$\{X \mid D_{s+1}^{k_1} \cdots D_{s+m}^{k_m}X = D_{s+1}^{l_1} \cdots D_{s+n}^{l_n}\} =$$

$$\begin{cases}
\emptyset, & k_1 < l_1 \\
\{X \mid D_{s+2}^{k_2} \cdots D_{s+m}^{k_m}X = D_{s+2}^{l_2} \cdots D_{s+n}^{l_n}\} \cup \\
\{D_{s+m+1}^{k_1}Y \mid D_{s+1}^{k_2} \cdots D_{s+m+1}^{k_m+1}Y = D_{s+2}^{l_2} \cdots D_{s+n}^{l_n}\}, & k_1 = l_1 \\
\{D_{s+m+1}^{l_1}Y \mid D_{s+2}^{k_2} \cdots D_{s+m+1}^{k_m+1}Y = D_{s+2}^{l_2} \cdots D_{s+n}^{l_n}\}, & k_1 > l_1
\end{cases}$$

Proof. Since $X$ is a solution of the equation

$$D_{s+1}^{k_1} \cdots D_{s+m}^{k_m}X = D_{s+1}^{l_1} \cdots D_{s+n}^{l_n}$$

it follows that $fX = s+m \oplus s+n$. Hence, by property 3.4, we have $X = D_{s+m+1}^{x_1} \cdots D_{x_{n-m}}^{x_{n-m}}$, for an increasing sequence $x_1 < \cdots < x_{n-m}$. Next, define the minimum rank of a drop operator sequence as the minimum of the ranks of its individual drop operators. Since the minimum rank of a drop operator sequence is invariant under application of the drop-exchange rule, it follows that the minimum rank of the left-hand side is at most $k_1$, whereas the minimum rank of the right-hand side is $l_1$. Hence equation 19 has no solutions, when $k_1 < l_1$. Moreover, when $k_1 > l_1$, it follows that $x_1$ must be equal to $l_1$. Hence by application of the drop exchange rule it follows that $X = D_{s+m+1}^{l_1}Y$, where $Y$ satisfies $D_{s+2}^{k_2} \cdots D_{s+m+1}^{k_m+1}Y = D_{s+2}^{l_2} \cdots D_{s+n}^{l_n}$. In case $k_1 = l_1$ similar reasoning results in two possibilities for $x_1$, viz. $x_1 = l_1$ or $x_1 > l_1$. In the first case $X = D_{s+m+1}^{l_1}Y$ with $Y$ as above. In the second case $X$ is a solution of the equation obtained by erasing the first drop operator from both the coefficient and right-hand side. \(\square\)

This theorem provides us with a recursive procedure to solve any equation of shape 19. So in order to solve any linear equation with a single unknown, it remains to be shown that any equation of shape 18 can be transformed into an equation of shape 19. The latter is easily accomplished by means of drop expansion. A t-fold expansion of the coefficient of $X$ and an s-fold expansion of the pseudo-canonical form on the right-hand side will make both sides start with a drop operator that has period $st+1$. So we conclude that the rules of the PDT-calculus are sufficient to solve linear equations with a single unknown periodic stream sampler.

As an example of the application of theorem 5.1 consider the equation $T_3^2X_c = D_3^0T_3^0$ for system $S$ defined in the previous section. Bringing both the coefficient and the right-hand side onto pseudo-canonical form and dropping subscript $c$, we obtain the equation $D_2^0D_3^1D_4^3X = D_2^0D_3^1D_4^2D_5^1D_6^5$. Table 2 contains the solution of this equation by repeated application of theorem 5.1.

Thus we see that there are six possible solutions for $X_c$. However, $X_c$ must also satisfy the equation $D_2^0X_c = D_3^0D_3^0$, or equivalently equation $D_2^0X_c = D_3^0D_3^0D_6^5$. The latter equation has only one solution, viz. $X_c = D_3^0D_6^5$. Of course, it would have been smarter to solve the second equation for $X_c$ first,
\{X \mid D_0^1 D_1^3 D_3^4 X = D_0^2 D_1^3 D_2^4 D_3^5 D_5^6 \} \\
= \{X \mid D_0^1 D_1^3 X = D_0^2 D_1^3 D_2^4 D_3^5 D_5^6 \} \cup \{D_0^0 X \mid D_0^2 D_1^3 X = D_0^4 D_1^3 D_2^4 D_3^5 \} \\
= \{X \mid D_0^1 X = D_0^2 D_1^3 \} \cup \{D_0^0 X \mid D_0^2 X = D_0^4 D_1^3 \} \cup \{D_0^0 D_1^2 D_3^4 X = D_0^4 D_1^3 D_2^4 D_3^5 \} \cup \{D_0^0 D_1^2 X = D_0^4 D_1^3 \} \\
= \{D_0^0 D_1^2, D_0^2 D_1^3, D_0^2 D_1^4, D_1^2 D_2^4, D_1^2 D_1^4, D_0^4 D_1^3, D_0^1 D_3^4 \}

Table 2: Recursive computation of the solution set of a linear equation.

whereafter it would have been a simple matter to check that its unique solution satisfies the first equation. In general it is most efficient to solve the equation with the shortest coefficient first.

Sofar we have established how to solve a linear equation in a single variable. As can be observed from the example in the previous section, however, most equations of $E_{sv}(S)$ contain two variables, one on each side of the equality sign. So we need to know under what conditions the solution of $E_{sv}(S)$ can be reduced to solving, one after another, a number of linear equations in a single variable.

**Theorem 5.2** Let $S$ be an acyclic SISO system consisting of split and merge components with input stream $A$ and output stream $B$. Then the set of equations $E_{sv}(S)$ has at most one solution, which can be found by application of theorem 5.1. In case the solution exists the system is a buffer.

**Proof.** Because the system is acyclic, we can topologically sort the ports and channels of the system, and order the variables of the associated set of equations accordingly. It is easily verified that each equation then has the property that the variable on the left-hand side has a higher rank than the variable occurring on the right-hand side. Hence solving the equations in increasing order will produce equations of type 17, which can be solved by application of theorem 5.1. Moreover, since both split and merge components neither destroy nor create data items, it must be the case that $fX_b = fX_a = 1$. Hence $X_b = I$ or equivalently $B = A$. □

Note that when an acyclic SISO system has no solution this does not mean that that system cannot perform a meaningful computation. For instance, it may be the case that its computation does not have the property that all streams in the system are substreams of the input stream. The block reverser described in section 3 is an example of such a system. Even when the system is a buffer, the associated set of equations may fail to have a solution, due to this situation.
6 Solving equations for arbitrary SISO systems

In the previous section we have shown how the functionality of certain split-merge systems can be computed. That approach is limited, because it only works if the system happens to be a buffer, and even in those cases it might fail. Nevertheless the approach is valuable, because buffers are an extremely important class of split-merge systems. Moreover, most buffer designs that are attractive from a performance perspective, e.g. all buffers described in [5], can be handled with that approach.

In this section we present a second approach that is more general. It enables us to determine the functionality of any SISO split-merge system, i.e. an arbitrary block permutation. The price we pay for this generality is that the set of equations is much larger, because the number of variables is much larger. As it turns out the number of variables and the number of equations are equal, since each variable occurs in two equations and each equation contains two variables.

Consider an arbitrary SISO system \( S \) consisting of split and merge components. As in the previous section we will construct a set \( E_{mv}(S) \) of equations for \( S \) whose solution will yield the functionality of \( S \). In contrast to the previous section, however, we will associate multiple variables (hence the subscript “mv”) with each stream \( C \) of the system. Each of these variables represents a particular substream of \( C \), in such a way that, when properly interleaved, together they will make up the entire stream.

The determination of \( E_{mv}(S) \) is done in two stages. First the number of variables per stream is calculated and thereafter the equations relating these variables are determined.

Let \( C \) be the stream of data items that flows through channel (port) \( c \) of system \( S \). Then the number of variables \( \phi_c \) corresponding to \( C \) will be chosen in such a way that it is proportional to the fraction of data items from the input stream that it contains\(^7\). The constant of proportionality is called \( \phi \) and is the same for all channels.

For input port \( a \) by definition \( \phi_a = \phi \). Since the fractions of the input and the output port must be equal, we also have \( \phi_b = \phi \). For the internal channels of the system the fractions are determined by the constraints imposed by the components. Component \( \text{Split}^{l+1}_{k+1}(a, c, d) \) imposes constraints \((l+1)\phi_c = \phi_a \) and \((l+1)\phi_d = l\phi_a \). Similarly, component \( \text{Merge}^{l+1}_{k+1}(e, f, b) \) imposes \((l+1)\phi_e = \phi_b \) and \((l+1)\phi_f = l\phi_b \). Because the flow of data items through the system is completely deterministic, there is for any specific constant of proportionality \( \phi \) at most one assignment of numbers to the variables \( \phi_c \) such that all the constraints imposed by the components hold. In general this assignment consists of rational numbers, but by an appropriate choice of \( \phi \) we can ensure that the assignment consists of natural numbers only. If an assignment exists, then the number of variables per channel are chosen in accordance with the minimal value of \( \phi \) that makes all \( \phi_c \) natural. Given a number \( \phi_c \) for channel \( c \), we define, for \( 0 \leq k < \phi_c \), the variable \( \lambda^k_c \) by \( \lambda^k_c = T^k_{\phi_c}C \). Note that, when taken together,

\(^7\)Since all streams are infinite, some care has to be taken to define this fraction. However, for each finite prefix of the input stream the fraction is well-defined, so we can take the limit for increasingly longer prefixes.
these variables indeed cover the entire stream $C$. Figure 6 shows the variables associated with the streams that flow in and out of a split component. Note that each arrow is labeled both with a port name and a number that represents the number of variables associated with that port.

$$
\begin{align*}
\mathcal{X}_a^p &= T_{m(l+1)}^p A, & 0 \leq p < m(l+1) \\
\mathcal{X}_c^p &= T_m^p C, & 0 \leq p < m \\
\mathcal{X}_d^p &= T_{ml}^p D, & 0 \leq p < ml
\end{align*}
$$

Figure 6: Variable counts and definitions for component $\text{Split}_{l+1}^k$.

Now that we have defined the variables associated with system $S$, we can determine the set of equations $E_{mv}(S)$. As with $E_{sv}(S)$, each component of $S$ gives rise to a number of equations. For split components that number of equations equals the number of variables associated with their input port, and for merge components that number equals the number of variables associated with their output port. For a split component Table 3 shows the derivation of the equations from the functional specification of the component, as given in Figure 2. Of course, a similar set of equations can be obtained for a merge component. Thus we arrive at an extremely simple, albeit large, set of equations $E_{mv}(S)$. Each equation has at most one variable at each side of the equality sign, and all coefficients are equal to $I$.

Provided the variables can be ordered such that for each equation the variable on the right-hand side has a lower rank than the variable on the left-hand side, this system has a unique solution.

**Theorem 6.1** Let $S$ be an acyclic SISO-system consisting of split and merge components with input stream $A$ and output stream $B$. Then the set of equations $E_{mv}(S)$ has precisely one solution of the form $T_{\phi}^k B = T_{\phi}^{\pi(k)} A$ for some natural number $\phi$.

**Proof.** Analogous to the proof of theorem 5.2

For systems that contain a cycle matters are more complicated. Consider system $\text{Cyc}_k$ described in Figure 7. The set of equations corresponding to this system depends on the value of $k$ and is given by

$$E_{mv}(\text{Cyc}_k) = 
\begin{cases}
\{\mathcal{X}_d = \mathcal{X}_c^0, \mathcal{X}_b^0, \mathcal{X}_d^1, \mathcal{X}_b^1, \mathcal{X}_d^2, \mathcal{X}_b^2, \mathcal{X}_d^3, \mathcal{X}_b^3, \mathcal{X}_a^0, \mathcal{X}_c^0, \mathcal{X}_a^1, \mathcal{X}_c^1, \mathcal{X}_a^2, \mathcal{X}_c^2, \mathcal{X}_a^3, \mathcal{X}_c^3\}, & k = 0 \\
\{\mathcal{X}_d = \mathcal{X}_c^0, \mathcal{X}_b^0, \mathcal{X}_d^1, \mathcal{X}_b^1, \mathcal{X}_d^2, \mathcal{X}_b^2, \mathcal{X}_d^3, \mathcal{X}_b^3, \mathcal{X}_a^0, \mathcal{X}_c^0, \mathcal{X}_a^1, \mathcal{X}_c^1, \mathcal{X}_a^2, \mathcal{X}_c^2, \mathcal{X}_a^3, \mathcal{X}_c^3, \mathcal{X}_a^1, \mathcal{X}_c^1, \mathcal{X}_a^2, \mathcal{X}_c^2, \mathcal{X}_a^3, \mathcal{X}_c^3\}, & k = 1 \\
\{\mathcal{X}_d = \mathcal{X}_c^0, \mathcal{X}_b^0, \mathcal{X}_d^1, \mathcal{X}_b^1, \mathcal{X}_d^2, \mathcal{X}_b^2, \mathcal{X}_d^3, \mathcal{X}_b^3, \mathcal{X}_a^0, \mathcal{X}_c^0, \mathcal{X}_a^1, \mathcal{X}_c^1, \mathcal{X}_a^2, \mathcal{X}_c^2, \mathcal{X}_a^3, \mathcal{X}_c^3, \mathcal{X}_a^1, \mathcal{X}_c^1, \mathcal{X}_a^2, \mathcal{X}_c^2, \mathcal{X}_a^3, \mathcal{X}_c^3\}, & k = 2
\end{cases}$$
In these equations the periodic stream samplers $X_a^k$ are considered constants and the remaining samplers are considered the unknowns. So solving system $E_{mv}(S)$ amounts to equating each unknown with one of the constants. Due to the very specific nature of the equations this is a trivial task. For $k = 0$ we obtain

$$(X_b^0, X_b^1) = (X_a^0, X_a^1) \quad (X_c^0, X_c^1, X_c^2) = (X_d, X_a^0, X_a^1) \quad X_d = X_c^0$$

Even though the first equation of this triple suggests that the output stream is a copy of the input stream, there is something wrong here, because we see that the stream that flows through channel $d$ has no relation with the input stream. Since the system is not capable of producing its own data items, it follows that the system must be flawed. Inspection of the program texts reveals what is the matter. The split component wants to start with an input from channel $c$, whereas the merge component wants to start with an input from channel $d$. So the system immediately deadlocks. Moreover, since both processes are blocked on input, this deadlock can not be overcome by adding additional buffer components to the system. For $k = 1$ we obtain

$$(X_b^0, X_b^1) = (X_a^0, X_a^1) \quad (X_c^0, X_c^1, X_c^2) = (X_a^0, X_a^0, X_a^1) \quad X_a = X_c^0$$

<table>
<thead>
<tr>
<th>0 ≤ $p$ &lt; $m$</th>
<th>0 ≤ $p$ &lt; $ml$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_c^p$ = { def. $X_c^p$ } \hspace{1cm} $X_a^p$ = { def. $X_a^p$ }</td>
<td>$T_m l D$ = { spec. $D$ } $T_m l D$ = { m-fold drop expansion }</td>
</tr>
<tr>
<td>$T_m l T_{l+1} A$ = { take elimination } $T_m l D$ = { m-fold drop elimination }</td>
<td>$T_m l D$ = { m-fold drop elimination }</td>
</tr>
<tr>
<td>$T_m l D$ = { m-fold drop elimination }</td>
<td>$T_m l D$ = { m-fold drop elimination }</td>
</tr>
<tr>
<td>$X_a^{p(l+1)+k}$ = { def. $X_a^p$ }</td>
<td>$X_a^{p(m-1)(l+1)+k}$ = { def. $X_a^p$ }</td>
</tr>
</tbody>
</table>

Table 3: Derivation of the equations of a split component.
Again we see that the output stream is a copy of the input stream. In this case, however, the internal streams are completely determined, so this is a functionally correct design of a (two-place) buffer system. In addition it can be verified that the system does not deadlock either. For $k = 2$ we obtain

$$(X_0^b, X_1^b) = (X_1^a, X_0^d) \quad (X_0^c, X_1^c, X_2^c) = (X_0^d, X_1^a, X_0^d) \quad X_d = X_0^a$$

In this case we see that the functionality of the system is that of a block reverser with block size 2. This system, however, also suffers from deadlock. After the merge component has accepted two input items, the first of which has been passed on to the split component, both components are blocked on output. So although functionally correct, this system will not work either. Because components are blocked on output, however, this deadlock can be overcome by adding buffer capacity to either the $c$-channel or the $d$-channel. The general problem of how much buffer capacity has to be added and in which channels is outside the scope of this paper.

### 7 Conclusions and future work

This paper describes PDT-calculus, a calculus to reason about streams that are obtained by periodic sampling. The calculus consists of only two generic operators and in its most basic form of only three equation schemes. The calculus is both sound and complete. An algebraic model in terms of a specific monoid of stream transformers, viz. the monoid of periodic stream samplers has been given.

The calculus can be used to prove the functional correctness of any system consisting of split and merge components. Such a proof presupposes the existence of specifications for the system and all its subsystems, i.e. the (desired) identity of substreams associated with all channels of the system should be known.

Apart from a posteriori correctness proofs the calculus can also be applied for system analysis, i.e. to determine the unknown functionality of a system. Two approaches have been given: one general method, and one more specialized method that can only succeed when the system is a buffer. In both methods one or more variables (periodic stream samplers) are associated with each channel of the system and a set of equations relating these variables is set up.
the system is an acyclic SISO-system, then the order of the equations can be chosen in such a way that all equations are linear equations with a single unknown periodic stream sampler. An algorithm for solving an arbitrary linear equation in a single variable has been given.

Moreover, it is not hard to imagine that the calculus can also be used in a correctness by construction design approach, as the construction of the block-reverser in section 4 shows. To make this observation more convincing, however, additional case studies should be done. These should also involve multiple-input multiple-output systems. Furthermore we observe that split and merge components occur in all kinds of systems that implement more complex functionality. e.g. various types of signal processing systems. There they serve to route data from the input stream to functional units and (partial) results to the output stream. In addition they provide the pipelining needed to improve the performance of such systems. Therefore it should be investigated how PDT-calculus can be embedded in larger calculi that deal with the correctness of these more complex systems.

Finally, it should be noted that the calculus deals with the functionality of systems and as such only addresses partial correctness. For total correctness also absence of deadlock has to be shown. This may require adding buffer capacity to some of the internal channels of a system. A subtle combination of both solutions methods may serve to identify the internal channels for which this is necessary, e.g. because the channels in which the the order of the data items are reversed with respect to the input streams can be identified. This also requires further research.

8 Acknowledgements

I would like to thank Arjeh Cohen, Peter Hilbers, Martijn v/d Horst, Johan Lukkien, and Hans Zantema for many fruitful discussions. In particular, I am grateful to Hans Zantema for pointing out the role of fractions in solving equations.

References


