Algebraic Identities for Pipelining Higher Order Functions

Ali E. Abdallah
The University of Reading
Department of Computer Science
Reading, RG6 6AY, U.K.
Email: A.Abdallah@reading.ac.uk

Abstract

A calculus of function decompositions for parallel programs derivation based on Bird-Meertens Formalism is presented. This is mainly a collection of useful algebraic identities between forms of algorithmic expressions. The main focus is on highlighting the algebra of a compositional functional form known as a *pipe pattern*. This functional form is suitable for highly parallel implementations. We show how this form can be systematically refined into pipelined networks of communicating processes in Hoare's CSP. It turns out that all familiar higher order functions such as *map*, *filter*, *foldl*, and *foldr* can be identitied with new algorithmic forms expressed as instances of this pipe pattern. This greatly facilitates a transformational programming methodology for deriving pipelining algorithms from abstract functional specifications. A catalogue of useful algebraic transformation rules which allow the decomposition of higher order functions to be derived from the decomposition of their arguments is presented. Novel methods for parallelizing directed reduction when the operator involved is not associative are developed. We show how the calculus can be applied for deriving highly parallel solutions to a variety of problems.

1 Introduction

Higher order functions such as *map*, *filter*, *foldl*, and *foldr* provide a high degree of abstraction in functional programs. Not only they provide the basic building blocks for writing clear and succint high level specifications for a large class of problems, their elegant algebraic properties can be fully exploited through a process of mathematical calculation to transform these specifications into efficient sequential and parallel implementations.

The importance and usefulness of higher order functions as a powerful tool for program construction and transformational derivation are clearly demonstarted in Bird-Meertens Formalism (BMF) [11, 9, 8, 10, 13] and in the Skeletons approach [16, 22]. Over the past decade, there have been growing interests in applying both BMF and the Skeletons approach for the systematic construction of efficient parallel programs from abstract specifications [16, 17, 22, 1, 2, 18, 19, 29, 32, 33]. The main attractions of this approach are the potential of increasing reusability, portability and scalability of parallel programs without sacrificing significant performance. However, in contrast with this paper, the primary focus of the Skeletons approach has mainly been on exploiting data parallelism rather than pipelined and MIMD parallelism.

In this paper we present an algebra for pipelining higher order functions based on Bird-Meertens Formalism. This is mainly a collection of useful algebraic identities between forms of algorithmic expressions. The main focus is on highlighting the algebra of a functional form called *pipe pattern* which is suitable for efficient highly pipelined implementations. It turns out that all familiar higher order functions can be identitied with new algorithmic forms expressed as instances of this pipe pattern. We develop a catalogue of useful algebraic transformation rules which allow the decomposition of higher order functions to be derived from the decomposition of their arguments and, in particular, when these arguments are defined recursively. We show how the resulting form can be systematically refined into reconfigurable pipelined networks of communicating processes in Hoare's CSP. We also show how the calculus can be applied for deriving highly parallel solutions to a variety of problems.

The formalization of the parallel interpretations of Skeletons within a concurrency framework such as CSP [20] is very useful. It does not only lead to a better understanding of their operational behaviours but it also allows reasoning about their correctness and analysis of their performance. In addition, the establishment of refinement concepts between functional and concurrent behaviours is a step towards the systematic generation of parallel code on various architectures such as array processors, MIMD, pipelined, and systolic machines.

Previous work in this area [1, 2] has mainly concentrated on providing systematic decomposition methods for exploiting pipelined parallelism in instances of the higher order function $foldr$. In this paper, we provide a more comprehensive treatment of all the common higher order functions such as *map* , $filter$, $foldl$, and $foldr$. The remainder of this paper is organised as follows. Section 2 contains a brief summary of notations and also introduces the concept of refinement from functions to processes in CSP. The concept of *pipe patterns* is introduced in Section 3 together with its rich algebraic properties. Sections 4, 5, 6, and 7 provide a calculus for pipelining the higher order functions *map*, *filter* and *directed reductions*, respectively. The usefulness of the calculus is illustrated by systematically deriving several highly pipelined algorithms for various applications. Section 8 concludes this paper.

2 Functions, processes, and refinement

Throughout this paper, we will use the functional notation and calculus developed by Bird and Meertens [11, 12, 14] for specifying algorithmics and reasoning about them and will use the CSP notation and its calculus developed by Hoare [20] for specifying processes and reasoning about them. We give a brief summary of the notation and conventions used in this paper. The reader is advised to consult the above references for details.

2.1 Higher order functions

Lists are finite sequences of values of the same type. The list concatenation operator is denoted by $++$ and the list construction operator is denoted by $:$. The elements of a list are displayed between square brackets and separated by comas. Functions are usually defined using higher order functions or by sets of recursive equations. The operator $*$ (pronounced "map") takes a function on the left, a list on the right, and applies the function to each element of the list. Informally, we have:

$$f * [a_1, a_2, \cdots, a_n] \quad = \quad [f(a_1), f(a_2), \cdots, f(a_n)]$$

The operator $\triangleleft$ (pronounced "filter") takes a predicate on the left; a list of values on the right; and returns, in order, all values in the list which satisfy the predicate.

The operator $/$ (pronounced "reduce") takes an associative binary operator on the left, a list of values on the right and returns the "summation" of all the elements of the list. This can be informally described as follows

$$(\oplus)/[a_1, a_2, \cdots, a_n] \quad = \quad a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

The left reduction operator $(\oplus \not\rightarrow e)$ (also known as $foldl$) generalises reduction by introducing an additional parameter e, as a starting value of the computation and gradually accumulates the result by successively applying the operator $\oplus$ while traversing the list from left to right. Informally, we have:

$$(\oplus \not\rightarrow e)\ [a_1, a_2, \cdots, a_n] \quad = \quad (\cdots((e \oplus a_1) \oplus a_2) \oplus \cdots) \oplus a_n$$

The right reduction operator $(\oplus \not\leftarrow e)$ (also known as $foldr$) is similar to $(\oplus \not\rightarrow e)$ except that the computation proceeds by traversing the list in the opposite direction, that is, from right to left.

$$(\oplus \not\leftarrow e)\ [a_1, a_2, \cdots, a_n] \quad = \quad a_1 \oplus (a_2 \oplus\ (\cdots \oplus (a_n \oplus e) \cdots)\ )$$

Note that the operator $\oplus$ used with directed reductions (both left and right) may not be associative. Its operands may not be of the same type indeed!

2.2 Algebraic Laws

One important asset of the functional programming framework and, in particular, Bird-Meertens Formalish (BMF) is its richness in algebraic laws which allow the transformation of a program from one form to another while preserving its meaning. Here is a short list of frequently used algebraic rules which will be used later in this paper. Historically, the "promotion rules" are intended to express the idea that an operation on a compound structure can be "promoted" into its components.

map distributivity:

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

map promotion:

$$f* \circ +\!\!\!+/ = +\!\!\!+/ \circ (f*)*$$

reduce promotion:

$$\oplus/ \circ +\!\!\!+/ = \oplus/ \circ (\oplus/)*$$

2.3 CSP Processes

In general, we use identifiers with lower case letters to name functional values and with upper case letters to name processes or types. We will use, in addition to the CSP notation, functions which return processes and functional operators such as map ($*$) and reduce ($/$). When parsing expressions, we assume that functional application has the highest precedence and associates to the left, but all other functional operators have equal precedence and associate to the right. For example, the expression $F * s +\!\!\!+ t$ means $F * (s +\!\!\!+ t)$ and not $(F * s) +\!\!\!+ t$. In CSP, the notation $P \;\not\prec\; b \;\not\succ\; Q$ is just an infix form for the traditional selection construct **if** b **then** P **else** Q. We find it convenient to use pattern matching conventions as in the following definition

$$\begin{array}{rcl} P & = & left?\text{“}a\text{”} \rightarrow Q \\ & & | \\ & & left?\langle x,\, y\rangle \rightarrow R(x,y) \end{array}$$

to mean

$$P \;=\; left?z \rightarrow (\, Q \;\not\prec x = a \not\succ\; R(fst\ z, snd\ z)\,)$$

2.3.1 Refinement from Functions to Processes

The refinement from functions to CSP processes is based on the formal treatment given in [1]. In general, a function $f :: [A] \rightarrow [B]$ is refined by a pipe process Q, written as $f \prec Q$, implies that f can be viewed as an abstract specification of Q. In this case, whenever Q consumes on its input channel a stream of values s, the output produced on its output channel must be identical to $f\ s$.

By convention, the end of each stream is denoted by eot, a special symbol to indicate"end of transmission". The operator $\rhd$ [1] is similar to the CSP piping operator $\gg$ except that the left operand of $\rhd$ is a producer (a process which can only output). The producers EOT and $Prd(s)$, for all lists s, are defined as follow:

$$\begin{array}{lcl} EOT & = & !eot \rightarrow SKIP \\ Prd\,[a_1, a_2, \cdots, a_n] & = & !a_1 \rightarrow !a_2 \rightarrow \cdots !a_n \rightarrow EOT \end{array}$$

2.3.2 Examples of refinements into sequential processes

We will give typical refinement of several key functions which will be encountered in subsequent sections. The corresponding refinement proofs can be found in [1].

The identity function $id_{[\mathrm{A}]}$

The process $COPY$ is a refinement of the identity function $id_{[A]} :: [A] \rightarrow [A]$ over lists of values.

$$COPY \;=\; \mu X \bullet ?x \rightarrow !x \rightarrow (\, SKIP \;\lhd x = eot \rhd\; X\,)$$

The function **map**

For any function $f :: A \rightarrow B$, the function $f* :: [A] \rightarrow [B]$ can be refined by the process:

$$MAP(f) \;=\; \mu X \bullet ?x \rightarrow (\, EOT \;\lhd x = eot \rhd\; !(f\ x) \rightarrow X\,)$$

The function **filter**

For any predicate $p :: A \rightarrow bool$, the filter function $(p\triangleleft) :: [A] \rightarrow [A]$ can be refined by the process:

$$FILTER(p) \;=\; \mu X \bullet ?x \rightarrow (\, EOT \;\lhd x = eot \rhd\; !x \rightarrow X \;\lhd p\ x \rhd\; X\,)$$

We find it convenient to describe transformation rules and specification refinement using the conventions from the CIP transformation project [15]). Hence, the notation

$$\begin{array}{c} A \\ \hline B \end{array} \;\langle\; C$$

means that the program template A is a valid refinement of the program template B, provided that the condition C holds.

3 Algebra of Pipe Patterns

3.1 Pipe Patterns

Pipe patterns are instances of the following higher function *pipe* defined as:

$$pipe\ f\ = (\circ)/\ \circ\ (f\,*)$$

For a given list of values $[a_1, a_2, .., a_n]$, the pipe pattern $pipe\ f\ [a_1, a_2, .., a_n]$ is identical to the following composition of functions.

$$f\ a_1\ \circ\ f\ a_2,\ ..,\ \circ\ f\ a_{n-1}\ \circ\ f\ a_n$$

Changing the initial seed sequence s of a pipe pattern results in another pipe pattern.

$$pipe\ \ f\ \ (g * s)\ \ =\ \ pipe\ \ (f \circ g)\ \ s$$

Note that this definition of a pipe pattern is more general than other previously given definitions of pipe patterns [1, 2] and pipe skeletons [16, 22]. The latters are just instances of $foldr\ (\circ)\ id\ fs$, where fs is a list of functions. Also observe that when *fs* is a list of functions of type $[A \to A]$, the composition of all the functions in this list can be written as a pipe pattern as follows.

$$(\circ)/\ fs\ \ =\ \ pipe\ \ id\ \ fs$$

It is often the case that all the functions in the list fs are just different instances of a single function f with two arguments. The instances are obtained by initialising the first argument of f to seeds (values) drawn from a given list s. As we shall see, this abstraction of a pipe is not only easy to use but it enjoys many nice algebraic properties. Firstly, it is a homomorphism. Secondly, subject to specific conditions on f, the sequence type of the seeds can be replaced by a bag (commutativity condition) or a set (idempotence must also hold) without affecting the meaning of the pattern. Thirdly, this pattern can be refined into a communicating network in CSP by only refining a typical element of the pipe. Moreover, it is possible to apply simple algebraic rules to reconfigure the pipe pattern by mapping it into a fixed size pipe network.

3.2 Transformation to CSP processes

To refine the pattern $pipe\ f\ s$ into a CSP network of processes, we wii make use of the result [1], that function composition is refined by process piping; though, order of the operands must be reversed.

$$\begin{array}{l} f :: [A] \to [B];\ g :: [B] \to [C] \\ g \circ f \\ \hline F \gg\ G \end{array} \ \langle\ f \prec F \wedge g \prec G$$

By an inductive argument, using the associativity of $\gg$, this result can be generalised so that the finite composition of functions in the pattern *pipe* $f\ [a_1, a_2, \ldots, a_n]$.

$$f\ a_1 \quad \circ \quad f\ a_2 \ \ .. \quad \circ \quad f\ a_{n-1} \quad \circ \quad f\ a_n$$

is refined by the piping the corresponding CSP processes:

$$F(a_n) \quad \gg \quad F(a_{n-1}) \ \ .. \quad \gg \quad F(a_{n-1}) \quad \gg \quad F(a_n)$$

where for all values a, the process $F(a)$ is a refinement of the function $f\ a$. This pipe network is concisely captured as an instance of the following CSP pattern:

$$PIPE\ F\ s \quad = \quad (\gg)/\ F * s$$

The pipe network $PIPE\ F\ [a_1, a_2, \ldots, a_n]$ is pictured in Figure 1. In summary, we have established:

$$\forall\, a : A \bullet \quad f\ a \ \prec\ F(a) \ \Rightarrow\ \ pipe\ f\ s\ \prec\ PIPE\ F\ (reverse\ s)$$

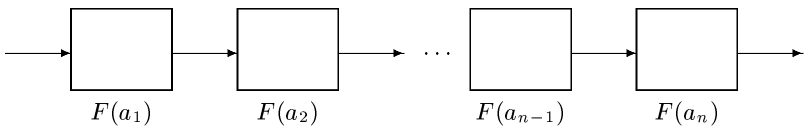

Figure 1: The CSP network $PIPE\ F\ [a_1, a_2, \ldots, a_n]$

3.3 Reshuffling Pipe Stages

A function $f :: A \to B \to B$ is said to commute through composition iff it satisfies the following condition:

$$\mathbf{CC(f)} \quad \forall\, a, b : A \bullet \quad f\ a \circ f\ b \quad = \quad f\ b \circ f\ a$$

When a function f commutes through composition, the elements of the pipe pattern *pipe* f can be rearranged in any order without affecting the overall functionality. That is, the list of seeds s can be safely replaced by a bag holding the same elements as s.

(pipe reshuffle)

$$\frac{pipe\ f\ s}{pipe\ f\ t}\ \updownarrow \quad \langle\ CC(f)\ \wedge\ t \in permutations\ s$$

In particular, the pipe functionality is preserved when the pipe stages are reversed.

$$CC(f) \;\;\Rightarrow\;\; \mathit{pipe}\ f\ s \;=\; \mathit{pipe}\ f\ (\mathit{reverse}\ s)$$

It is useful to note that if a function f commutes through composition and g is a function such that $f \circ g$ is well defined, then $f \circ g$ also commutes through composition.

$$CC(f) \;\;\Rightarrow\;\; CC(f \circ g)$$

Moreover, if a function f commutes through composition then so does the function $\mathit{pipe}\ f$.

$$CC(f) \;\;\Rightarrow\;\; CC(\mathit{pipe}\ f)$$

The justification for this property becomes very clear Section 3.4. When a function f satisfies, in addition to $CC(f)$, the following idempotence condition:

$$\mathbf{I(f)} \quad \forall\, a : A \bullet \quad f\ a \circ f\ a \;=\; f\ a$$

then duplicate stages can be safely removed from the pipe. In other words, the list of seeds can be replaced by a set of seeds without affecting the pipe functionality.

$$CC(f)\ \wedge I(f) \Rightarrow \quad \mathit{pipe}\ f\ [a_1, a_2, \cdots a_n] \;=\; \mathit{pipe}\ f\ \{a_1, a_2, \cdots a_n\}$$

Finally, for all values k, the constant function $\mathit{const}\ k$ is a left zero for the function composition operator.

$$\mathit{const}\ k \;\circ\; \mathit{pipe}\ f\ s \;=\; \mathit{const}\ k$$

However, it is not (quite) a right zero!

$$\mathit{pipe}\ f\ s \;\circ\; \mathit{const}\ k \;=\; \mathit{const}\ (\mathit{pipe}\ f\ s\ k)$$

3.4 Transformation to a fixed length pipe

Current parallel machines have usually a fixed number of processing elements, say p. To map the general pipe pattern to a fixed length pipe, we first specify a partition function $\mathit{parts}\ p$ which partitions any list of values s into exactly p groups of consecutive elements of s. This function is specified as follows:

$$\begin{array}{l} \mathit{parts} :: \mathit{num} \rightarrow [A] \rightarrow [[A]] \\ (+\!\!+)/\,(\mathit{parts}\ p\ s) = s \end{array}$$

That is the result of $\mathit{parts}\ p\ s$ is a list $[v_1, v_2, .., v_p]$ satisfying:

$$s \;=\; v_1 +\!\!+ v_2 +\!\!+ .. +\!\!+ v_p$$

To transform *pipe f s* to a composition of exactly p functions, we reason as follows:

$$
\begin{array}{lll}
\mathit{pipe}\ f & = & \{\text{def. of } \mathit{pipe}\} \\
& & (\circ)/ \circ f* \\
& = & \{\text{because } (+\!\!+)/ \circ \mathit{parts}\ p = \mathit{id}\} \\
& & (\circ)/ \circ f* \circ (+\!\!+)/ \circ \mathit{parts}\ p \\
& = & \{\text{map promotion}\} \\
& & (\circ)/ \circ (+\!\!+)/ \circ f** \circ \mathit{parts}\ p \\
& = & \{\text{reduce promotion } (\circ)/\} \\
& & (\circ)/ \circ (\circ)/* \circ f** \circ \mathit{parts}\ p \\
& = & \{\text{map distributivity}\} \\
& & (\circ)/ \circ ((\circ)/ \circ f*)* \circ \mathit{parts}\ p \\
& = & \{\text{folding } \mathit{pipe}\} \\
& & (\circ)/ \circ (\mathit{pipe}\ f)* \circ \mathit{parts}\ p \\
& = & \{\text{folding } \mathit{pipe}\} \\
& & (\mathit{pipe}\ (\mathit{pipe}\ f)) \circ \mathit{parts}\ p
\end{array}
$$

The above result can be neatly summarised in the following law:

(convertion to fixed length pipe)

$$
\frac{\mathit{pipe}\ f\ s}{\mathit{pipe}\ (\mathit{pipe}\ f)\ (\mathit{parts}\ p\ s)}\ \updownarrow
$$

Transformation to CSP processes

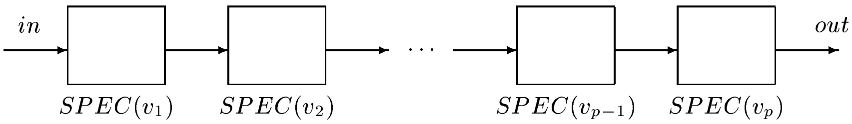

Figure 2: A fixed length pipe for $SPEC(v_1 +\!\!+ v_2 +\!\!+ .. +\!\!+ v_p)$.

4 Recursive shared data decomposition

We now turn to investigate the decomposition of functions recursively defined over lists using *tail recursion with shared data.* In recursive definitions, we use the term shared data to indicate values that are needed for each recursive call. To achieve the decomposition of such functions, it is necessary to introduce a new data-tupling transformation technique which eliminates shared data from recursive calls. The whole derivation is then encapsulated as a compact transformation rule which allows the synthesis of the derived form to be done in a single transformation step.

Starting form

A typically such definition has of the form:

$$\begin{array}{lcl} h\;[]\;x & = & e \\ h\;(a\colon s)\;x & = & f\;a\;x\;(h\;s\;x) \end{array}$$

The function h takes a list s, a value x, and recursively computes the result by traversing the input list from right to left. The value of x is "shared" among all the recursive call of h, giving the justification for the name of the pattern. An alternative definition of h can be described as right reduction:

$$\begin{array}{lcl} h\;s\;a & = & (\oplus_a \not\leftarrow e)\;s \\ & & \text{where}\;\; x \oplus_a y = \;\; f\;a\;x\;y \end{array}$$

An Example

Consider counting the number of occurrences of a value a in a given list s. The function *count* can be formulated as follows:

$$\begin{array}{lcl} \mathit{count}\;s\;a & = & (\oplus_a \not\leftarrow 0)\;s \\ & & \text{where}\;\; x \oplus_a n = \;\; \mathbf{if}\; x = a\; \mathbf{then}\; n+1\; \mathbf{else}\; n \end{array}$$

Other instances of h will be encountered later in the context of polynomial evaluation (Section 4) and spell cheking (Section 5).

Decomposition

We propose to generalize the function h into a new function, say th, that returns a pair of values, or a tuple in the more general case. The first component of the pair is the value of the relevant shared data whereas the second component of the pair is the required computational result. Hence, we suggest the following definition

$$th\;s\;x \;\;=\;\; \langle x,\, h\;s\;x \rangle$$

Our first task is to redefine h as an instance of the th. By definition we have

$$h\;s \;\;=\;\; \mathit{final} \circ (th\;s)$$

where the function *final* returns the second element of a pair. The second task is to synthesize a new definition for th which does not involve shared parameters. For the base case, we have

$$\begin{array}{rcll} th\ []\ x & = & \langle x,\, h\ []\ x\rangle & \{\ \textsc{Instantiation}\ \} \\ & = & \langle x,\, e\rangle & \{\ \textsc{Unfolding}\ \} \end{array}$$

and for the general case, we reason:

$$\begin{array}{rcll} th\ (a\colon s)\ x & = & \langle x,\, h\ (a\colon s)\ x\rangle & \{\ \textsc{Instantiation}\ \} \\ & = & \langle x,\, f\ a\ x\ (h\ s\ x)\rangle & \{\ \textsc{Unfolding}\ \} \\ & = & f'\ a\ \langle x,\, h\ s\ x\rangle & \{\ \textsc{Abstraction}\ \} \\ & = & f'\ a\ (th\ s\ x) & \{\ \textsc{Folding}\ \} \\ & = & ((f'\ a)\circ(th\ s))\ x & \{\ \textsc{Def. of}\ (\circ)\ \} \end{array}$$

where the subsidiary function f' is defined as follows.

$$f'\ a\ \langle x,\, y\rangle \quad = \quad \langle x,\, f\ a\ x\ y\rangle$$

By defining the function initial as

$$initial\ x \quad = \quad \langle x,\, e\rangle$$

we can redefine th as

$$\begin{array}{rcl} th\ [] & = & initial \\ th\ (a\colon s) & = & (f'\ a)\circ(th\ s) \end{array}$$

The new inductive definition of th is in a compositional form but this time without recursive shared data. This can be directly decomposed as follows.

$$th\ s = (\circ)/\ f' * s \ \circ\ initial$$

The above derivation can be encapsulated in a single transformation rule.

(Recursive shared data decomposition)

$$\begin{array}{lcl} \multicolumn{3}{l}{h :: [A] \to B \to B;\ f :: A \to B \to B \to B;\ e :: B} \\ h\ []\ x & = & e \\ h\ (a\colon s)\ x & = & f\ a\ x\ (h\ s\ x) \\ \hline \multicolumn{3}{c}{\updownarrow} \\ h\ s & = & \mathit{final}\ \circ\ (pipe\ f'\ s)\ \circ\ initial \\ f'\ a\ \langle x,\, y\rangle & = & \langle x,\, f\ a\ x\ y\rangle \\ initial\ x & = & \langle x,\, e\rangle \\ \mathit{final}\ \langle x,\, y\rangle & = & y \end{array}$$

This decomposition is most useful when such a recursively definitiond function is used as an argument for a high order function such as map or filter. This transformation technique is closely related to the one used by Meertens [24]. Pettorossi [27] has used a different kind of tupling technique. His technique tuples partial computation (as opposed to the value of an argument) in order to avoid repeating the evaluation of partial results and, therefore, achieve efficiency in (sequential) functional programs.

5 Pipelined decomposition of map

We present several algebraic laws which allow the decomposition of $(f*)$ to be directly derived from the decomposition of f. We also consider the decomposition of $(f*)$ when the function f is described as a right reduction or is recursively defined over a list of values. We show how the general decomposition of $f*$ can be realized in CSP as an efficient pipelined network of processes. Finally, we give examples of applying this rule.

5.1 Map Decomposition Laws

The first decomposition law is captured by the distributivity property of the map operator $(*)$ over function composition $(\circ)$. We have

$$\textbf{(}*\textbf{DC1)} \quad (g \circ f)* \quad = \quad (g*) \circ (f*)$$

By a simple inductive argument the above law can be generalised to any finite composition of functions. Hence, this generalisation allows the decomposition of the map function $h*$ to be directly derived from the decomposition of its argument h.

$$((\circ)/\, fs)* \quad = \quad (\circ)/\ map * fs$$

This law can be formulated in terms of *pipe* as follows:

$$\textbf{(}*\textbf{DC2)} \quad (pipe\ id\ fs)* \quad = \quad pipe\ \ map\ \ fs$$

A more powerful map decomposition law expresses the map of a pipe as a pipe of several map stages as follows:

$$\textbf{(}*\textbf{DC3)} \quad (pipe\ \ f\ \ s)* \quad = \quad pipe\ \ (map \circ f)\ \ s$$

Observe that by instantiating f to id, the law $*$ DC2 can be obtained. This law $*$ DC3 will be frequently encountered in later examples. An informal justification for its correctness:

$$\begin{array}{lll}
 & (pipe\ \ f\ \ [a_1, a_2, \cdots a_p])* & \\
= & ((\circ)/\, f * [a_1, a_2, \cdots a_p])* & \{\ \text{Def. } pipe\ \} \\
= & ((\circ)/\, [f\ a_1, f\ a_2, \cdots, f\ a_p])* & \{\ \text{Def. } f*\ \} \\
= & (\circ)/\, map * [f\ a_1, f\ a_2, \cdots, f\ a_p] & \{\ *\text{DC2}\ \} \\
= & (\circ)/\, [map\ (f\ a_1), map\ (f\ a_2), \cdots, map\ (f\ a_p)] & \{\ \text{Def. } (*)\ \} \\
= & (\circ)/\, [(map \circ f)\ a_1, (map \circ f)\ a_2, \cdots, (map \circ f)\ a_p] & \{\ \text{Def. } (\circ)\ \} \\
= & (\circ)/\, (map \circ f) * [a_1, a_2, \cdots, a_p] & \{\ \text{Def. } (*)\ \} \\
= & pipe\ \ (map \circ f)\ \ [a_1, a_2, \cdots, a_p] & \{\ \text{Def. } pipe\ \}
\end{array}$$

5.2 Map decomposition for recursively defined arguments

By combining recursive shared data decomposition and map decomposition rule we can formulate a more powerful map decomposition rule. Consider the decomposition of $(h\ s)*$ where s is a given list of values and the function h is defined as an instance of *tail recursion with shared data* as follows:

$$\begin{array}{lcl} h\ []\ x & = & e \\ h\ (a\colon s)\ x & = & f\ a\ x\ (h\ s\ x) \end{array}$$

As an example, assume that we are given two lists of words: ws, representing a document, and ks, representing a list of keywords. Consider the problem of counting the number of occurrences in the document ws of each word in the keywords list ks. A straightforward specification of this problem is captured as follows:

$$\mathit{wordcount}\ ws\ ks \quad = \quad (\mathit{count}\ ws) * ks$$

The *recursive shared data decomposition* rule aloows the function $(h\ m)$ to be decomposed as follows:

$$\begin{array}{lcl} h\ s & = & \mathit{final}\ \circ\ (\mathit{pipe}\ f'\ s)\ \circ\ \mathit{initial} \\ f'\ a\ \langle x, y\rangle & = & \langle x, f\ a\ x\ y\rangle \\ \mathit{initial}\ x & = & \langle x, e\rangle \\ \mathit{final}\ \langle x, y\rangle & = & y \end{array}$$

By applying the map decomposition law $*$DC3 and using the distributivity property of the operator $(\,*\,)$, we get

$$h\ s \quad = \quad \mathit{final}*\ \circ\ ((\mathit{pipe}\ \ (\mathit{map}\circ f')\ s)\ \circ\ \mathit{initial}*$$

This completes the decomposition of $(h\ s)*$ and, therefore, the whole derivation is encapsulated in the following compact transformation rule:

(Map decomposition of Recursion with shared data)

$$\begin{array}{l} h :: [A] \to B \to B;\ f :: A \to B \to B \to B;\ e :: B \\ \begin{array}{lcl} h\ []\ x & = & e \\ h\ (a\colon s)\ x & = & f\ a\ x\ (h\ s\ x) \end{array} \\ \hline \updownarrow \\ \begin{array}{lcl} h\ s* & = & \mathit{final}*\ \circ\ ((\mathit{pipe}\ \ (\mathit{map}\circ f')\ s)\ \circ\ \mathit{initial}* \\ f'\ a\ \langle x, y\rangle & = & \langle x, f\ a\ x\ y\rangle \\ \mathit{initial}\ x & = & \langle x, e\rangle \\ \mathit{final}\ \langle x, y\rangle & = & y \end{array} \end{array}$$

Thus we have shown how to transform $(h\ s)*$ into a new functional form which exhibits a high degree of pipelined parallelism. This parallelism is scalable with the size of s. For example, the above rule expresses $(h\ [a_1, a_2, \cdots, a_n])*$ as a composition of $(n+2)$ functions as follows.

$$h\ [a_1, a_2, \cdots, a_n]* = \mathit{final}* \ \circ\ (f\ a_1)*\ \circ\ (f\ a_2)*\ \circ\ \cdots\ \circ\ (f\ a_n)*\ \circ\ \mathit{initial}*$$

This form can be implemented in CSP as a pipelined network of $(n+2)$ processes. What remains to be shown is how the behaviour of the individual processes in the pipe can be synthesized.

Transformation to CSP

For any function $f : A \to B$, the function $f*$ can be refined by the process $MAP(f)$. This allows $(h\ s)*$ to be refined into the following pipe network, $H(s)$, of CSP processes:

$$H(s) \quad = \quad MAP(\mathit{initial}) \ \gg\ (PIPE\ \ (MAP \circ f')\ \ (\mathit{reverse}\ s)) \ \gg\ MAP(\mathit{final})$$

By unfolding MAP, the complete behaviour of the individual processes involved in the pipe network are synthesised as follows:

$$\begin{array}{lcl}
MAP(\mathit{initial}) & = & \mu Z \bullet \begin{array}[t]{l} left?\text{“}eot\text{”} \to right!eot \to SKIP \\ \mid \\ left?x \to right!\langle x,\ e\rangle \to Z \end{array} \\
(MAP \circ f')\ (a) & = & MAP(f'\ a) \\
 & = & \mu Z \bullet \begin{array}[t]{l} left?\text{“}eot\text{”} \to right!eot \to SKIP \\ \mid \\ left?\langle x,\ y\rangle \to right!\langle x,\ f\ a\ x\ y\rangle \to Z \end{array} \\
MAP(\mathit{final}) & = & \mu Z \bullet \begin{array}[t]{l} left?\text{“}eot\text{”} \to right!eot \to SKIP \\ \mid \\ left?\langle x,\ y\rangle \to right!y \to Z \end{array}
\end{array}$$

The above network can be depicted as follows:

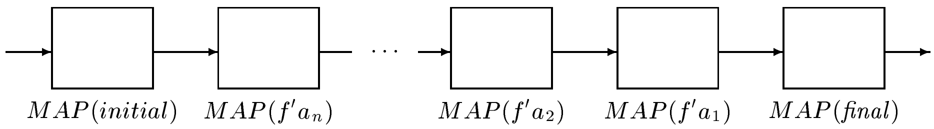

Figure 3: The map network H $[a_1, a_2, \cdots, a_n]$

Examples of instances of this derivation include pipelined polynomial evaluation [5], several parallel rendering algorithms [6, 34] and matrix multiplication. To map the pipe network into a fixed size pipe we use a technique similar to one introduced in the next section (filter).

6 Parallel decomposition of filter

In this section we will establish some transformation rules which allow the parallel decomposition of $p \triangleleft$ to be derived from the decomposition of the predicate p.

We will generalize (lift) the logical conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$) operators respectively into ($\dot\wedge$), ($\dot\vee$) and ($\dot\neg$) which take predicates (boolean valued functions) as arguments and return predicates as results. These operators are defined as follows:

$$
\begin{array}{l}
\dot\wedge, \dot\vee \;\; :: (A \rightarrow \textsc{bool}) \rightarrow (A \rightarrow \textsc{bool}) \rightarrow (A \rightarrow \textsc{bool}) \\
\dot\neg \;\; :: (A \rightarrow \textsc{bool}) \rightarrow (A \rightarrow \textsc{bool}) \\
(p \;\dot\vee\; q) \;\; x \quad = \quad p\, x \vee q\, x \\
(p \;\dot\wedge\; q) \;\; x \quad = \quad p\, x \wedge q\, x \\
(\dot\neg \; p) \;\; x \qquad = \quad \neg\, (p\; x)
\end{array}
$$

The lifted binary operators $\dot\wedge$ and $\dot\vee$ inherit from their counterparts $\wedge$ and $\vee$ associativity, commutativity, and idempotence. Moreovere, $\dot\wedge$ enjoys the values *const True* as identity and *const False* as zero. Given a value c, *const c* denotes the constant function that returns c for all values of its argument. A useful generalisation, by lifting and summation, of De-Morgan's law is

$$
\textbf{(DML)} \qquad \dot\neg\, (\dot\vee)/\, ps \quad = \quad (\dot\wedge)/\, (\dot\neg) * ps
$$

6.1 Filter Decomposition Laws

The following is a list of basic laws concerning the filter operator ($\triangleleft$):

$$
\begin{array}{lll}
\textbf{(}\triangleleft\ \textbf{id)} & (const\ True) \triangleleft & = \;\; id_{[A]} \\
\textbf{(}\triangleleft\ \textbf{zero)} & (const\ False) \triangleleft & = \;\; const\ [] \\
\textbf{(}\triangleleft\ \textbf{idem)} & (p \triangleleft) \circ (p \triangleleft) & = \;\; (p \triangleleft) \\
\textbf{(}\triangleleft\ \textbf{com)} & (p \triangleleft) \circ (q \triangleleft) & = \;\; (q \triangleleft) \circ (p \triangleleft)
\end{array}
$$

When the filtering predicate is a conjunction of two predicates, the decomposition of filter can be simply obtained by succesive filtering with each predicate.

$$
\textbf{(}\triangleleft\ \textbf{DC1)} \qquad (p \;\dot\wedge\; q) \triangleleft \quad = \quad (p \triangleleft) \circ (q \triangleleft)
$$

This property can be generalised to any conjunction of predicates. That is, when the filtering predicate is expressed as $((\dot\wedge)/\, ps)$, ps being a list of predicates, the whole filter function can be directly expressed as composition of a list of filters.

$$
((\dot\wedge)/\, ps) \triangleleft \; = (\circ)/\, filter * ps
$$

This is just an instance of the pipe pattern.

$$(\lhd\ \textbf{DC2})\quad ((\dot{\wedge})/\, ps)\lhd \;=\; pipe\ filter\ ps$$

Another frequently used generalisation of the above law is:

$$(\lhd\ \textbf{DC3})\quad ((\dot{\wedge})/\, q * s)\lhd \;=\; pipe\ \ (filter \circ q)\ \ s$$

Observe that the law $\lhd$ DC2 can be obtained by by replacing q with the the identity function. The justification for $\lhd$ DC3 is as follows.

$$\begin{array}{lll} ((\dot{\wedge})/\, q * s)\lhd & = (\circ)/\, filter * (q * s) & \{\ \text{BY}\ \lhd\ \text{DC2}\ \} \\ & = (\circ)/\, (filter \circ q) * s & \{\ *\ \text{DISTRIBUTIVITY}\ \} \\ & = pipe\ \ (filter \circ q)\ \ s & \{\ \text{DEF. OF}\ pipe\ \} \end{array}$$

Finally, by appealing to DML, an additional useful filter decomposition law can be derived from $\lhd$ DC3.

$$(\lhd\ \textbf{DC4})\quad (\dot{\neg}\,(\dot{\vee})/\, ps)\lhd \;=\; pipe\ \ (filter \circ (\dot{\neg}))\ \ ps$$

In Section 6.5 we investigate the decomposition of $(p\lhd)$ when p is *any* combination of predicates and in particular a disjunction of other predicates.

6.2 Reconfigurability

From the law ($\lhd$ com), it follows that the function *filter* satisfies the composition commutativity condition (CC). Hence, the pipe stages of the filter pattern can be reshuffled without affecting the overall functionality of the pipe.

$$pipe\ \ filter\ \ [a_1, a_2, .., a_n] \;=\; pipe\ \ filter\ \ (bag\ [a_1, a_2, .., a_n])$$

In particular, for all functions q such that $filter \circ q$ is well defined, we have:

$$pipe\ \ (filter \circ q)\ \ s \;=\; pipe\ \ (filter \circ q)\ \ (reverse\ s)$$

It also follows from the law ($\lhd$ idem) that duplicate stages in the pipe can be removed without affecting the correctness of the network. In other words, the list data type can be replaced by a set data type. Hence, we have

$$pipe\ \ filter\ \ [a_1, a_2, .., a_n] \;=\; pipe\ \ filter\ \ \{a_1, a_2, .., a_n\}$$

Consider partitioning the list s into a fixed number p of sublists (they don't have to be consecutive segments!). We will use a more flexible partition function, *setparts* p, which does not take into account the order nor the repetition of elements in a list. For any list s, *setparts* p s is a list $[v_1, v_2, .., v_p]$ such that:

$$mkset\ s \;=\; mkset\ (v_1 +\!\!+\ v_2 +\!\!+\ ..\ +\!\!+\ v_p)$$

where the function $mkset$ is informally defined as:

$$mkset\ \ [a_1, a_2, .., a_n] \quad = \quad \{a_1, a_2, .., a_n\}$$

We have:

$$pipe\ \ filter\ \ s \quad = \quad pipe\ \ (pipe\ filter)\ \ (setparts\ p\ s)$$

6.3 Transformation to CSP

The law ($\lhd$ DC3) identifies filtering of conjunctions with an instance of the *pipe* pattern.

$$((\dot{\wedge})/\, q * s) \lhd \quad = \quad pipe\ \ (filter \circ q)\ \ s$$

Massively parallel pipe network

To transform the latter into a pipe network of CSP processes, it is sufficient to refine the pipe element $(filter \circ q)(a)$ into an appropriate CSP process, say $F(a)$. It has already been shown [1] that, for all predicates p, the function $filter\ p$ is refined by the process $FILTER(p)$. Hence, the pipe element $(filter \circ q)(a)$ is refined by the process $(FILTER \circ q)(a)$. Consequently, the refinement of the whole pipe pattern into CSP is synthesised as follows.

$$pipe\ \ (filter \circ q)\ \ s \quad \prec \quad PIPE\ \ (FILTER \circ q)\ \ s$$

By simply unfolding the definition of $FILTER$, the completeCSP behaviour of each pipe stage is derived as follows.

$$\begin{aligned} (FILTER \circ q)\ (a) &= FILTER(q\ a) \\ &= \mu X \bullet\ left?x \rightarrow (\ right!eot \rightarrow SKIP\ \ \lhd x = eot \rhd \\ &\qquad\qquad\qquad\qquad right!x \rightarrow X\ \ \lhd q\ a\ x \rhd\ \ X\ ) \end{aligned}$$

Refinement to a fixed length pipe network

The following algebraic law:

$$pipe\ \ (filter \circ q)\ \ s \quad = \quad pipe\ \ (pipe\ (filter \circ q))\ \ (setparts\ p\ s)$$

can be used to transforms the filter pipe pattern into a fixed length pipe network of size is p. For all lists v, the pipe element $(pipe\ \ (filter \circ q))\ \ v$ is identical to $((\dot{\wedge})/\, q * v) \lhd$. Hence, its refinement to CSP can be directly obtained as an instance of the sequential process $FILTER$, namely $FILTER((\dot{\wedge})/\, q * v))$. Therefore, the resulting fixed length pipe network is expressed in CSP as:

$$PIPE\ \ \ (FILTER \circ (\dot{\wedge})/ \circ q *)\ \ \ (setparts\ p\ s)$$

6.4 Example: Pipelined Spell Checker

Given a list of words, *dic*, that contains exactly all possible correct configurations of words in a specific dictionary, consider the problem of constructing a spell checker that takes a document and returns all those words which are incorrectly spelt. With the aid of the predicate *correct dic* which tests whether a given word is correctly spelt, the functional specification of this problem can be captured as an instance of $\triangleleft$ as follows.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{spell} :: [\textsc{word}] \rightarrow [\textsc{word}] \rightarrow [\textsc{word}];\ \mathit{correct} :: [\textsc{word}] \rightarrow \textsc{word} \rightarrow \textsc{bool}} \\ \mathit{spell\ dic} & = & (\dot{\neg}\, \mathit{correct\ dic}) \triangleleft \\ \mathit{correct\ dic} & = & (\dot{\vee}) / (=) * \mathit{dic} \end{array}$$

By systematic algebraic manipulations, the filtering predicate ($\dot{\neg}$ *correct dic*) is decomposed into a conjunction of simpler predicates. Hence, the whole filtering function *spell* is transformed into a pipe pattern as follows.

$$\begin{array}{lcll} \mathit{spell\ dic} & = & (\dot{\neg}\, \mathit{correct\ dic}) \triangleleft & \{\ \textsc{Definition}\ \} \\ & = & (\dot{\neg}\, (\dot{\vee}) / (=) * \mathit{dic}) \triangleleft & \{\ \textsc{Unfolding}\ \} \\ & = & ((\dot{\wedge}) / (\dot{\neg}) * (=) * \mathit{dic}) \triangleleft & \{\ \textsc{De-Morgan Law}\ \} \\ & = & ((\dot{\wedge}) / ((\dot{\neg}) \circ (=)) * \mathit{dic}) \triangleleft & \{\ *\ \textsc{Distributivity}\ \} \\ & = & ((\dot{\wedge}) / (\neq) * \mathit{dic}) \triangleleft & \{\ \textsc{Definition} \neq\ \} \\ & = & \mathit{pipe}\ \ (\mathit{filter} \circ (\neq))\mathit{dic} & \{\ \triangleleft\ \ \mathrm{DC3}\ \} \end{array}$$

As shown in the previous section, this pipe pattern can be refined into the following pipe network of CSP processes:

$$SPELL(dic) \quad = \quad PIPE \ \ (FILTER \circ (\neq)) \ \ dic$$

Figure 4 depicts this network assuming *dic* is $[a_1, a_2, \cdots, a_n]$.

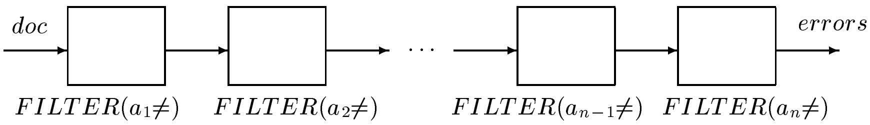

Figure 4: A massively parallel pipe implementation of $SPELL([a_1, a_2, \cdots, a_n])$.

By appealing to the result of Section 6.3, this massively parallel network can be directly transformed into a fixed length pipe network of size p. In this case, the list *dic* is partitioned into a set of sublists $\{v_1, v_2, .., v_p\}$ and the fixed size network is obtained by piping p copies of (sequential or parallel) implementations of solutions to the same problem (*spell* with a smaller dictionary) . The resulting network is illustrated in Figure 5.

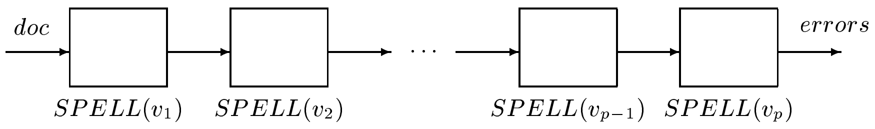

Figure 5: A fixed length pipe implementation of $SPELL([a_1, a_2, \cdots, a_n])$.

6.5 Filter Decomposition Rule

When the filtering predicate p is expressed as a combination of simpler predicates, say $p = combine\ [p_1, p_2, \cdots, p_n]$, the filtering process can be done in parallel by appropriately combining the functions $p_1 *$, $p_2 *$, ... and $p_n *$. To show this, consider deriving a parallel decomposition of the filter pattern $(h\ s) \triangleleft$ where s is a list of values and h is a function defined by tail recursion with shared data (Section 4) as follows:

$$
\begin{array}{lcl}
h :: [A] \to B \to \text{BOOL};\ e :: \text{BOOL} & & \\
spec\ s & = & (h\ s) \triangleleft \\
h\ []\ x & = & e \\
h\ (a{:}\ s)\ x & = & f\ a\ x\ (h\ s\ x)
\end{array}
$$

Derivation

To achieve the desired decomposition, we appeal to the shared data tupling technique which was introduced in Section 4. Hence, we define

$$th\ s\ x \quad = \quad \langle x,\ h\ s\ x \rangle$$

and we propose the following generalisation

$$newspec\ s \quad = \quad (th\ s) *$$

In the previous section, we have established that $th\ s$ can be transformed into the following pipe pattern

$$
\begin{array}{lcl}
th\ s & = & (pipe\ f'\ s) \ \circ \ initial \\
f'\ a\ \langle x,\ y \rangle & = & \langle x,\ f\ a\ x\ y \rangle \\
initial\ x & = & \langle x,\ e \rangle
\end{array}
$$

and, hence, by $*$ DC3, $newspec$ can be expressed as

$$newspec\ s \quad = \quad ((pipe\ \ (map \circ f')\ s) \ \circ \ initial *$$

The remaining task is to redefine *spec* in terms of *newspec*. We have

$$\begin{array}{lcll}
spec\ s & = & filters \circ newspec & \\
filters\ [] & = & [] & \\
filters\ (\langle x, b\rangle : xs) & = & x : filters\ xs, & \text{if } b \\
 & = & filters\ xs, & \text{otherwise}
\end{array}$$

The proof by induction of the above step is trivial. By unfolding *newspec*, the new definition of *spec* is expressed as the following pipe pattern

$$spec\ s \quad = \quad filters \circ ((pipe\ \ (map \circ f')\ s) \circ \ initial *$$

The above derivation can be encapsulated in the following transformation rule:

(Filter decomposition of Recursion with shared data)

$$\begin{array}{l}
spec :: [B] \rightarrow [A] \rightarrow [A];\ h :: [B] \rightarrow A \rightarrow \text{BOOL} \\
e :: \text{BOOL};\ f :: B \rightarrow A \rightarrow \text{BOOL} \rightarrow \text{BOOL} \\
\begin{array}{lcl}
spec\ s & = & (h\ m) \triangleleft \\
h\ []\ x & = & e \\
h\ (a : s)\ x & = & f\ a\ x\ (h\ s\ x)
\end{array}
\end{array}$$

$$\Updownarrow$$

$$\begin{array}{l}
spec\ s \ = \ filters \circ ((pipe\ \ (map \circ f')\ s) \circ \ initial * \\
\begin{array}{lcll}
f'\ a\ \langle x, y\rangle & = & \langle x, f\ a\ x\ y\rangle & \\
initial\ x & = & \langle x, e\rangle & \\
filters\ [] & = & [] & \\
filters\ (\langle x, b\rangle : xs) & = & x : filters\ xs, & \text{if } b \\
 & = & filters\ xs, & \text{otherwise}
\end{array}
\end{array}$$

Transformation to CSP

The filtering pattern *spec s* can be refined into the following pipelined network:

$$SPEC(s) \quad = \quad MAP(initial) \ \gg \ (PIPE\ \ (MAP \circ f')\ \ s) \ \gg \ FILTERS$$

where the CSP process $FILTERS$ is any refinement of the function filters. A typical definition of $FILTERS$ is:

$$\begin{array}{lcl}
FILTERS & = & \mu X \bullet \ left?\text{"}eot\text{"} \ \rightarrow \ right!eot \ \rightarrow \ SKIP \\
 & & \qquad\quad | \\
 & & \qquad\quad left?\langle x, b\rangle \ \rightarrow \ (\,right!x \ \rightarrow \ X \ \ \not< b \not> \ \ X\,)
\end{array}$$

6.6 Example: Filtering Disjunctions

As a special instantiation of the above rule, consider filtering disjunctions of predicates. We are interested in deriving a parallel decomposition of $(\mathit{some}\ ps) \lhd$ where ps is a given list of predicates and some is defined as

$$\mathit{some} \quad = \quad (\dot{\vee})/$$

Another recursive definition of some which matches DC-FIL-R1 is

$$\begin{array}{lcl} \mathit{some}\ []\ x & = & \mathit{false} \\ \mathit{some}\ (p\colon ps)\ x & = & f\ p\ x\ (\mathit{some}\ ps\ x) \\ f\ p\ x\ b & = & (p\ x) \vee b \end{array}$$

Therefore, we can automatically obtain

$$(\mathit{some}\ ps) \lhd \quad = \quad \mathit{filters} \ \circ \ ((\mathit{pipe}\ \ (\mathit{map} \circ f')\ ps) \ \ \circ \ \ \mathit{initial} *$$

A corollary which follows from the above result is

$$\begin{array}{lcl} (p \,\dot{\vee}\, q) \lhd & = & \mathit{filters} \circ (\mathit{extend}\ p) * \circ (\mathit{extend}\ q) * \circ \mathit{initial} * \\ \mathit{extend}\ p\ \langle x,\, b\rangle & = & \langle x,\, b \vee (p\ x)\rangle \\ \mathit{initial}\ x & = & \langle x,\, \mathit{false}\rangle \end{array}$$

7 Parallel Decomposition of Directed Reductions

A lot of research efforts has been devoted towards investigating data parallelism inherent in the general reduction operator. Most of existing data parallel implementations rely on two facts. The binary operator for the reduction must, first, be associative and, second, be computed in constant time. However, there are numerous applicaions where these assumptions may not be hold [11]. Neither directed reduction operators relies on these assumptions.

7.1 Pipelined decomposition of right reduce

The right reduction operator captures a general pattern of computation described in Section 2. An alternative formulation of this pattern is given by the following recursive definition of the higher order function $foldr$: Right reduction captures the computation of those functions which can be defined using *tail recursion*. This pattern has a high degree of implicit parallelism which has been identified in [1] resulting in the following decomposition rule:

(Right Reduction Decomposition)

$$\begin{array}{l} foldr :: (\alpha \to [\beta] \to [\beta]) \to [\beta] \to ([\alpha] \to [\beta]) \\ foldr\ f\ e\ [] = e \\ foldr\ f\ e\ (a\colon s) = f\ a\ (foldr\ f\ e\ s) \\ \hline \updownarrow \\ foldr\ f\ e\ s = (\mathit{pipe}\ \ f\ \ s)\ e \end{array}$$

Provided that for all values $a : A$, the function $(f\ a)$ is refined into a pipe process $F(a)$, $(foldr\ \ f\ \ e\ \ s)$ is refined into the following network of communicating processes

$$FOLDR(f, e, s) \quad = \quad Prd(e) \ \ \triangleright \ \ PIPE \ \ F \ \ (reverse\ \ s)$$

The network
$FOLDR(f, e, [a_1, a_2, \ldots, a_n])$, can be pictured as depicted in Figure 6.

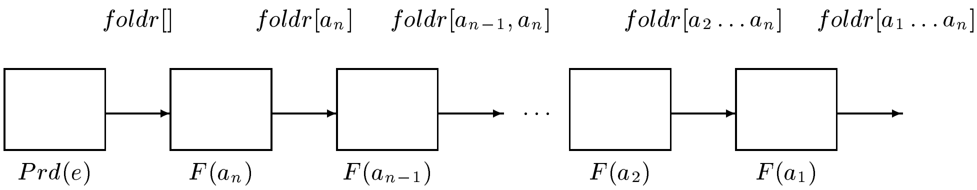

Figure 6: $FOLDR(f, e, [a_1, a_2, \ldots, a_n])$

Several parallel algorithms for list processing were derived by directly applying the above refinement technique [1, 2]. It is shown that the application of this transformation strategy leads to linear time parallel algorithms with linear number of processes from quadratic time sequential programs. However, it is important to note that in order to ensure efficiency of the network $SPEC([a_1, a_2, \ldots, a_n])$, the function f must have an on-line implementation (that is, repeatedly consumes some input and produces some output until it terminates).

7.2 Example: Removal of Duplicates

The function $rmdup :: [\alpha] \rightarrow [\alpha]$ that removes all but the first occurrence of each element in a list is defined as:

$$\begin{array}{lcl} rmdup\ [] & = & [] \\ rmdup\ (x\!:xs) & = & x : filter\ (\neq x)\ (rmdup\ xs) \end{array}$$

Clearly, this function is an instance of $foldr\ \ f\ \ e$, where, $e = []$ and the step function f is defined as:

$$f\ \ x\ \ ys = x : filter\ (\neq x)\ ys$$

Hence, for all lists s, a parallel implementation of $rmdup\ s$ is synthesized as:

$$\begin{array}{lcl} RMDUP(s) & = & FOLDR(f, [], s) \\ & = & EOT \triangleright (PIPE \ \ F \ \ (reverse\ \ s)) \\ F(x) & = & !x \ \rightarrow \ FILTER(\neq x) \end{array}$$

The diagram in Figure 7 depicts how the network $RMDUP([4,3,4,5,3,5,2,3])$ may evolve with time. Observe that, the sequential implementation of *rmdup* has a quadratic time complexity but the pipelined version runs in linear time.

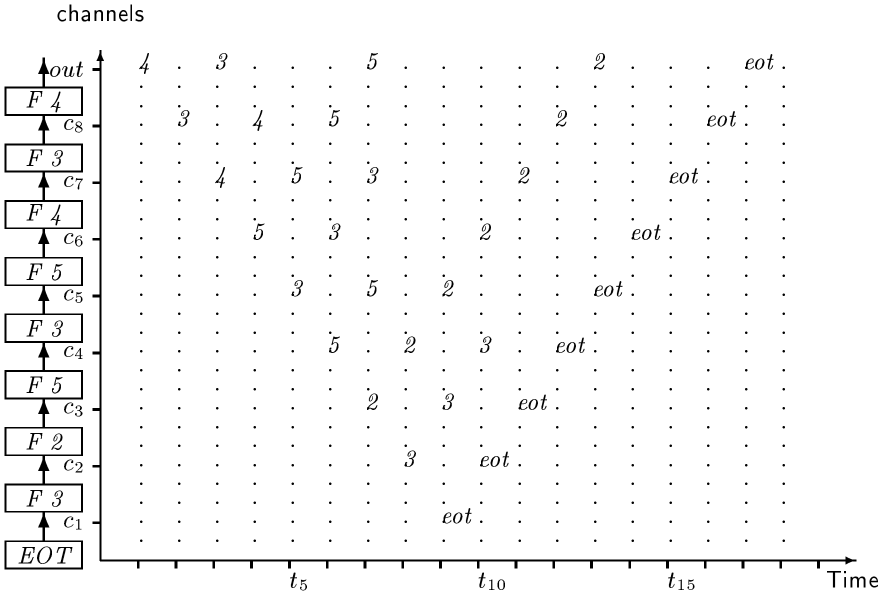

Figure 7: Time diagram for the network $RMDUP[4,3,4,5,3,5,2,3]$

7.3 Parallel decomposition of left reduce

The left reduction operator $(\oplus\not\to e)$ (also known as $foldl$) captures a general pattern for the computation of a class of functions which are usually defined recursively using the well known technique of *parameter accumulation*. A function is usually defined using an additional parameter for accumulating the result after each recursive call. Typically, the computation of a list of values starts from the first element of the list and proceeds towards the last. This is in contrast with tail recursion where computation starts from the last element of the list and proceeds towards the first. The left reduction operator $foldl$ which is used for this purpose is formally defined as follows:

$$
\begin{array}{lcl}
foldl :: ([\beta] \to \alpha \to [\beta]) \to [\beta] \to ([\alpha] \to [\beta]) & & \\
foldl \oplus t\ [] & = & t \\
foldl \oplus t\ (a\colon s) & = & foldl \oplus\ \ (t \oplus a)\ s
\end{array}
$$

Our objective is to show how the above pattern can be transformed into a highly parallel network of communicating processes. We aim to achieve this by transforming *foldl* into a an instance of the right reduce operator *foldr* and then applying the decomposition result of the previous section. To do so, we appeal to the following algebraic law [?] which closely relates both directed reduction operators.

$$(\oplus \not\rightarrow e) \quad s \quad = \quad (\widetilde{\oplus} \not\leftarrow e)\ (reverse\ s)$$

where, the binary operator $\widetilde{\oplus}$ is defined as $x \,\widetilde{\oplus} y = y \oplus x$. Therefore, by using *foldr* decomposition rule, we get the following decomposition rule for *foldl*:

(Left reduction decomposition)

$$(\oplus \not\rightarrow\ e) \quad s \quad = \quad (pipe\ \ (\widetilde{\oplus})\ \ (reverse\ s))\ \ e$$

Finally, the refinement of the CSP network of processes corresponding to the above decomposition of $\oplus \not\rightarrow\ e\ s$ is synthesised as follows

$$FOLDL(\oplus, e, s) \quad = \quad Prd(e) \triangleright (PIPE\ \ ADD\ \ s)$$

where for all values of a, the process $ADD(a)$ refines the function $(a\,\widetilde{\oplus}) = (\oplus a)$.

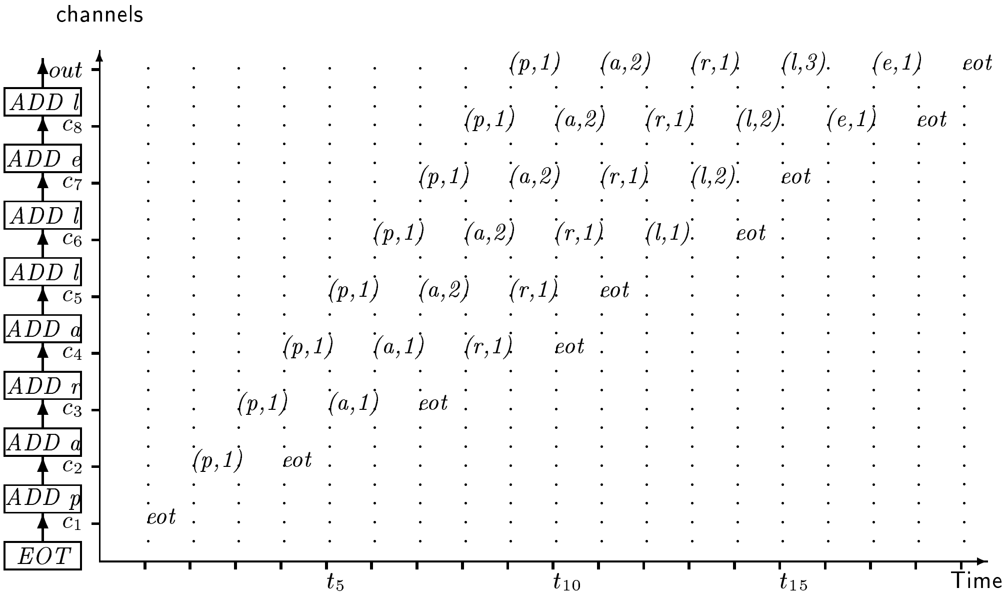

Figure 8: Parallel computation of $MKBAG(\text{“}parallel\text{”})$

7.4 Parallel convertion of lists to bags

The function *mkbag* converts a list into a bag. For any list xs, $(mkbag\ xs)$ is a list of pairs. Each pair has the form (x, i) where x is an elememt of xs and i is a positive number indicating the count of occurrences of x in the list xs. For example, we have:

$$mkbag \text{ "}parallel\text{"} = [(p,1),(a,2),(r,1),(l,3),(e,1)]$$

The function *mkbag* can be defined recursively as follows:

$$\begin{array}{lcl}
mkbag :: [\alpha] \to [(\alpha, num)] & & \\
mkbag\ s & = & mkbag'\ \ []\ \ s \\
mkbag'\ t\ \ [] & = & t \\
mkbag'\ t\ (x: s) & = & mkbag'\ \ (t \uplus x)\ \ s
\end{array}$$

where the operator $\uplus$ is defined as

$$\begin{array}{lcll}
\uplus :: [(\alpha, num)] \to \alpha \to [(\alpha, num)] & & & \\
\quad []\quad \uplus\ x & = & [(x,1)] & \\
((y,i): ys) \uplus\ x & = & (y,i):\ (ys \uplus x), & \text{if}\ \ x \neq y \\
 & = & (y, i+1): ys, & \text{otherwise}
\end{array}$$

The function *mkbag* is clearly an instance of left reduction in which we have $mkbag\ s = (\uplus \not\rightarrow [])\ s$. Therefore, provided that the process $ADD(x)$ correctly implements the function $(\uplus\ x)$, $mkbag\ s$ can be refined into the following pipelined network of processes:

$$\begin{array}{lcl}
MKBAG(s) & = & FOLDL(\uplus, [], s) \\
 & = & Prd([])\ \rhd\ (PIPE\ \ ADD\ \ s)
\end{array}$$

For all values x, the process $ADD(x)$ is defined as follows:

$$\begin{array}{lll}
ADD(x) = \mu X \bullet ?z \to (& !(x,1) \to EOT & \nprec z = eot \nsucc \\
 & !(x, i+1) \to COPY & \nprec\ x = y\ \nsucc \\
 & !z \to X & \\
 & \text{where}\quad z = (y,i) &)
\end{array}$$

The timed behaviour of the network $MKBAG($"*parallel*"$)$ can be pictured as in Figure 8. Assuming that the length of the list s is n, the sequential implementation of $mkbag(s)$ requires $O(n^2)$ steps but the parallel implementation runs in linear time.

The work in this paper mainly focuses on exploiting MIMD and pipelining parallelism implicit in functional programs. It goes a step further in refining the implicitly parallel functional version into networks of CSP processes. This allows us to address performace issues and reason about the parallel implementation whithin a proper framwork of concurreny in which these phenomena can be clearly understood.

8 Conclusion

Based on Bird-Meertens Formalism, we have established a collection of algebraic identities useful for pipelining higher order functions. We have identified a general functional form called *pipe pattern* which is suitable for highly parallel implementations and explored its algebraic properties. We have also developed several algebraic laws which allow functional decomposition of the higher order functions map and $filter$ to be derived from the decomposition of their arguments. Moreover, we have presented methods for parallelizing directed reduction when the operator involved is not associative. We have also shown, in each case, how to refine the derived functional algorithmic form into a massively pipelined network of communicating CSP processes. We have applied this calculus for the derivation of several highly parallel algorithms drawn from a broad range of examples.

By factorising instances of the pipe pattern within the decomposition of the higher order functions, we were able to tranfer all the algebraic properties of the pipe pattern into these higher order functions without having to do a single proof. For example, once we have established an algebraic law that transforms a pipe pattern into a fixed length pipe network, this transformation immediatly applies to all the decomposition of higher order functions.

The compact transformation rules which have been established in this paper are important for three reasons. Firstly, they involve functional templates which are frequently encountered in program derivation. Secondly, they concisely embody new interesting parallel programming transformation techniques. Thirdly, their underlying derivations are either skillful or quite lengthy. Consequently, whenever a problem is expressed as an instance of the initial template of one of these rules the whole expensive derivation process need not be repeated. A highly parallel implementation of the problem automatically follows, just in one transformation step, by taking the corresponding instance of the derived template of the rule.

The underlying ideas behind the transformation rules vary in depth. The succinctness of the algorithmic notation known as *BMF* and the large repertoire of laws between expressions of the notation were important not only for deriving deep and complex results from simple ones, but also for deriving them quickly with a few transformation steps.

References

[1] A. E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to *CSP* Processes, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag, 1995) 67-96.

[2] A. E. Abdallah, Synthesis of Massively Pipelined Algorithms for List Manipulation, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert

(eds), Proceedings of the *European Conference on Parallel Processing, EuroPar'96*, LNCS **1024**, (Springer Verlag, 1996), pp 911-920.

[3] A. E. Abdallah, and T. Theoharis, Derivation of Efficient Parallel Algorithms on a Ring of Processors, in: E. Luque, eds., Proceedings of the European Conference on Parallel and Distributed Computing and Systems, Barcelona, Spain, (IASTED/ACTA press, June 1997), 118-125.

[4] A. E. Abdallah, A Graphical Tool for the Visualisation and Animation of Communicating Sequential Processes, in D. Pritchard and J. Reeve (eds), Proceedings of the *European Conference on Parallel Processing, EuroPar'98*, Southampton, UK, LNCS **1470**, (Springer Verlag, September 1998), 165-172.

[5] A. E. Abdallah, Functional Process Modelling. In K Hammond and G. Michealson (eds), *Research Directions in Parallel Functional Programming*, (Springer Verlag, October 1999). pp339-360.

[6] A. E. Abdallah, and T. Theoharis, Transformational Derivation of Pipelined Rendering Algorithms. Submitted to the *Journal of Supercomputing*, 2000.

[7] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, 2nd Edition, (Benjamin/Cummings Publishing Company, 1994).

[8] R. S. Bird, J. Gibbons, and G. Jones, Formal derivation of a pattern matching algorithm, *Science of Computer Programming* **12** (1989), 93-104.

[9] Bird, R. S. and Meertens, L. G. L. T. Two Exercices Found in a Book on Algorithmics. In *Program Specification and Transformation*. North Holland, 1986

[10] R. S. Bird, and O. de Moor, List Partitions, *Formal Aspects of Computing* **5** (1) (1993).

[11] R. S. Bird, An Introduction to the Theory of Lists, in M. Broy, ed., *Logic of Programming and Calculi of Discreet Design*, (Springer, Berlin, 1987) 3-42.

[12] R. S. Bird, Functional Algorithm Design, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag, 1995) 2-17.

[13] R. S. Bird, and O. de Moor, *The Algebra of Programming*, (Prentice-Hall, 1996).

[14] R. S. Bird, and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, 1988).

[15] CIP language group, *The Munich project CIP*, LNCS **183**, (Springer-Verlag, 1984).

[16] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation* (Pitman, 1989).

[17] J. Darlington, A. J. Field, P.G. Harrison, and P.H.J. Kelly, Q. Wu, and R.L. While, Parallel Programming Using Skeletons Functions, in *PARLE93, Parallel Architectures and Languages Europe* LNCS , (Springer-Verlag, 93).

[18] S. Gorlatch and C. Lengauer, Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism, *Formal Aspects of Computing* **7** (6) (1995) 663-682.

[19] S. Gorlatch Systematic Efficient Parallelization of Scan and Other List Homomorphisms, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert (eds), Proceedings of the *European Conference on Parallel Processing, Euro-Par'96*, LNCS **1124**, (Springer-Verlag, 96) 401-408.

[20] C. A. R. Hoare, *Communicating Sequential Processes.* (Prentice-Hall, 1985).

[21] G. Jones and M. Sheeran, Circuit design in Ruby, in J. Staunstrup, ed., *Formal methods in VLSI Design* (Noth-Holland, 1990) 13-70.

[22] Paul Kelly, *Functional Programming for Loosley-Coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing, (Pitman, 1989).

[23] W. Luk, and G. Jones, The derivation of regular synchronous circuits, *Proc. International Conference on Systolic Arrays*, (San Diego, 1988).

[24] Lambert Meertens, Paramorphisms. *Formal Aspects of Computing* **4** (5) (1992) 493-424.

[25] B. Möller, Deductive Hardware Design: A Functional Approach. In Bernhard Möller and John Tucker, eds., *Prospects for hardware Foundations*, LNCS **1546**, (Springer Verlag, 1998) 421-468.

[26] Z.G. Mou, and M. Hudak, An Algebraic Model for Divide-and-Conquer Algorithms and its Parallelism, *Journal of Supercomputing*, **2** (3) (1988).

[27] Pettorossi, A. *Methodologies for Transformation and Memoing in Applicative Languages. Ph.D Thesis*, Edinburgh University, 1984.

[28] F. Rabhi and G. Lapalme, *Algorithms: A Functional Programming Approach*, (Addison-Wesley,1999).

[29] F. Rabhi, Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach. In J.R. Davy and P.M.Dew (eds), *Abstract Machine Models for Highly Parallel Computers*, (Addison-Wesley,1999).

[30] C. Runciman, and D. Wakeling, *Functional Programming Applications*, (UCL press,1995).

[31] D. B. Skillicorn, Models for Practical Parallel Computation, *International Journal of Parallel Programming* **20** (2) (1991) 133-158.

[32] D. B. Skillicorn, The Bird-Meertens Formalism as a Parallel Model. In J.S. Kowalik and L. Grandinetti (eds), *Software for Parallel Computation*, (Springer Verlag, 1992).

[33] D. B. Skillicorn, *Foundations of Parallel Programming*, (Cambridge University Press, 1994).

[34] T. Theoharis and A. E. Abdallah, Formal Derivation of Two Parallel Rendering Algorithms. In H R Arabnia, ed., Proceedings of the *International Conference on Parallel and Distributed Techniques and Applications*, (CSREA Press, June 1999), pp 1444-1450.