# *Communication lifting: fixed point computation for parallelism*

WILLEM G. VREE and PIETER H. HARTEL
*Department of Computer Systems, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*
*(e-mail:* {wimv, pieter}@fwi.uva.nl

**Abstract**

Communication lifting is a program transformation that can be applied to a synchronous process network to restructure the network. This restructuring in theory improves sequential and parallel performance. The transformation has been formally specified and proved correct and it has been implemented as an automatic program transformation tool. This tool has been applied to a small set of programs consisting of synchronous process networks. For these networks communication lifting generates parallel programs that do not require locking. Measurements indicate performance gains in practice both with sequential and parallel evaluation. Communication lifting is a worthwhile optimization to be included in a compiler for a lazy functional language.

## Capsule Review

This paper concerns a program transformation, ‘communication lifting’, for lazy functional languages. The transformation applies to programs which represent a certain class of process networks. The essence is to replace a number of ‘synchronized’ streams by a stream of tuples reducing the stream management overhead. The effect is that the pipeline parallelism is transformed onto a parallel master-slave type of computation. This work appears to have many applications.

## 1 Introduction

A process network is a system of communicating processes, which are connected by streams. The communicating processes are functions and the streams are potentially infinite lists of values upon which the functions operate. Programming with process networks has a long history, which dates back to the seminal work of Kahn (1974). Many special purpose languages such as Lucid (Ashcroft and Wadge, 1977), Esterel (Berry and Cosserat, 1984), Signal (Gautier *et al.*, 1987) and Lustre (Caspi *et al.*, 1987) have been developed to support programming with streams and process networks. The disadvantage of developing a new language is that it also requires a new implementation to be built. The approach that we will take is to build process networks using a subset of a standard lazy functional language, while taking special measures to guarantee good performance, both sequentially and in parallel. The evaluation mechanism of lazy functional languages naturally supports programming with potentially infinite lists of values (Peyton Jones, 1987). The process networks are thus embedded in a general purpose programming language, obviating the need for special compilers and language support systems. This approach has been advocated amongst others by Kelly (1989).

In an implementation of a lazy functional language, a stream is represented by a finite list, which is terminated by a *suspension*. This is a calculation that is suspended until further notice. A suspension can be revived and executed at any time to compute further elements of the stream, together with a new trailing suspension. The latter can be executed in turn to build further list elements, etc. In an implementation of a lazy functional language this mechanism of executing and building suspensions is completely automatic. Lazy evaluation of programs implementing process networks may incur considerable cost because each element in each stream requires executing a suspension and constructing a new one. When a large number of streams is involved, the cost may be prohibitively high. In practice large networks will indeed arise, for instance in simulations of digital circuits. Here each flip-flop is represented by two coupled *nand* functions that are mapped over streams of clock and data values. Even a small circuit will contain a large number of flip-flops, so that the simulation of such a circuit will require managing a large number of streams (Vree, 1989).

The cost of managing a large number of streams can be considerably reduced when the network is synchronous. In a synchronous network, executing one suspension will cause all suspensions on connected streams to be executed as well. All such closely related suspensions are said to belong to the same *generation*. It should thus be possible to revive and execute all suspensions in the same generation at the same time. The computations can be organized such that the management cost is shared between all streams. All streams are advanced by one generation at the same time.

The joint management of all suspensions in a synchronous network can be performed as follows. The *zip* of all streams in a network is a single stream, such that the original stream elements of one generation are gathered in one *state tuple*. The network as a whole will compute generation after generation, while managing only a single stream of tuples. Within a generation the functions that used to operate on stream elements now operate on the elements of one large tuple. The step from one generation of the state tuple to the next causes only normal forms to be shared. Parallel evaluation of the components of the state tuple is thus attractive because the parallel computations do not require locking.

Practical networks are often synchronous; for instance a digital logic simulation is synchronous because all circuits are essentially driven by a common clock. Most of the special purpose languages that have been developed for programming with networks are also synchronous. It is thus important to develop efficient implementations of synchronous network programs. We claim that it is an advantage to be able to build such an efficient implementation without having to resort to developing a new language and a compiler for that language.

The purpose of this paper is to present a program transformation called *communication lifting* that takes a synchronous process network consisting of $n$ streams into a network with a single stream of $n$-tuples. The transformation is rooted in the theory of recursive programs, based on the explicit calculation of fixed points of sets of recursive equations. This is the subject of section 2. Section 3 discusses the efficient implementation of programs that consist of synchronous process networks. Section 4 formally defines synchronous process networks. Communication lifting on simple process networks is described in Section 5. Section 6 describes a set of transformations that bring a more general synchronous process network in the form required for communication lifting proper. Performance measurements are reported in Section 7. A comparison with related work is given in Section 8 and the conclusions follow in Section 9. The correctness proofs of the program transformation may be found in the appendix.

## 2 Theoretical considerations: fixed points

Explicit calculation of the fixed point of a recursive program is both unusual (Allison, 1986) and inefficient (Manna *et al.*, 1973). Efficient computation rules such as the *normal order rule*, that can be shown to be safe (Vuillemin, 1973), are generally preferred. Direct fixed point iteration is inefficient because it calculates a sequence of approximations to the fixed point (if one exists). Each subsequent approximation is either the same as the previous, or better. The basic idea behind communication lifting is that as successive approximations are often the same, or almost the same, it may be more efficient to calculate the *changes* in the approximations only.

Consider as an example the system of Equations fib.a and fib.b over streams (infinite lists) to calculate the Fibonacci numbers $\tilde{a} = 0, 1, 1, 2, 3, 5, \ldots$. (The required auxiliary functions such as the family of functions *map_n* are defined in Figure 4.) The variables denoting streams are marked with a small wavy line ($\tilde{a}$) to render these variables typographically distinct from other identifiers.

$$\tilde{a} = 0 : \tilde{b} \qquad \text{(fib.a)}$$

$$\tilde{b} = 1 : map\_2\ (+)\ \tilde{a}\ \tilde{b} \qquad \text{(fib.b)}$$

Using the function *fix* as the fixed point operator and the standard domain construction, the solution of these equations is given by:

$$\langle \tilde{a}, \tilde{b} \rangle = fix\ fib \qquad \text{(fib.fix)}$$

$$fib\ \langle \tilde{a}, \tilde{b} \rangle = \langle 0 : \tilde{b},\ 1 : map\_2\ (+)\ \tilde{a}\ \tilde{b} \rangle \qquad \text{(fib.def)}$$

The communication lifting transformation causes the changes in the approximations to be computed as follows. Define a new stream $\tilde{z}$ as the *zip* of the fixed point:

$\tilde{z} \ = zip\_2 \ (fix \ fib)$

$\equiv$ (by fib.fix)

$\tilde{z} \ = zip\_2 \ \langle \tilde{a}, \tilde{b} \rangle$

$\equiv$ (by fib.a and fib.b)

$\tilde{z} \ = zip\_2 \ \langle 0 : \tilde{b}, \ 1 : map\_2 \ (+) \ \tilde{a} \ \tilde{b} \rangle$

$\equiv$ (unfold *zip_2*)

$\tilde{z} \ = \langle 0, 1 \rangle : zip\_2 \ \langle \tilde{b}, \ map\_2 \ (+) \ \tilde{a} \ \tilde{b} \rangle$

$\equiv$ (define $plus \ x = fst \ x + snd \ x$ and use the definition of $\tilde{z}$)

$\tilde{z} \ = \langle 0, 1 \rangle : zip\_2 \ \langle map\_1 \ snd \ \tilde{z}, \ map\_1 \ plus \ \tilde{z} \rangle$

$\equiv$ (property of *zip_n*, see law 3.35 in Jeuring (1992)
and define $nextstate \ x = \langle snd \ x, \ plus \ x \rangle$)

$\tilde{z} \ = \langle 0, 1 \rangle : map\_1 \ nextstate \ \tilde{z}$

$\equiv$ (property of *iterate_0*)

$\tilde{z} \ = iterate\_0 \ nextstate \ \langle 0, 1 \rangle$

Hence (the *zip* of) the fixed point of a system of equations over streams can be calculated as follows: start with an initial state ($\langle 0, 1 \rangle$) and *iterate_0* over the state transformation function *nextstate*. Both the state transformation function and the initial state are systematically derived from the system of equations.

To complete the communication lifting transformation for the *fib* example, the required output stream $\tilde{a}$ must be recovered from the stream of pairs $\tilde{z}$. This, and gathering the newly introduced definitions yields a complete program:

$$\begin{aligned}
\tilde{a} \ &= \ map\_1 \ fst \ (iterate\_0 \ nextstate \ \langle 0, 1 \rangle) \\
nextstate \ s \ &= \ \langle snd \ s, \ plus \ s \rangle \\
plus \ s \ &= \ fst \ s + snd \ s
\end{aligned}$$

As a finishing touch, the definition of *nextstate* can be simplified by unfolding the definitions of *plus*, *fst* and *snd* and by using pattern matching thus:

$$\begin{aligned}
\tilde{a} \ &= \ map\_1 \ fst \ (iterate\_0 \ nextstate \ \langle 0, 1 \rangle) \\
nextstate \ \langle a, b \rangle \ &= \ \langle b, c \rangle \\
& \quad where \\
& \quad c \ = \ (+) \ a \ b
\end{aligned}$$

The communication lifting transformation is discussed in more detail in the following sections of the paper. For now we note that for the application of *map_2* in the network one definition is generated under the *where* of the *nextstate* function, which captures basically the same calculation as the *map_2*. Also for each application of (:) in the network, there is one element in the state tuple processed by the *nextstate* function.

The *iterate_0* function controls the succession of generations of state tuples and thereby captures all the recursion originally scattered throughout the network. Given the tuple produced by the previous generation, the *nextstate* function (which is not recursive!) calculates the tuple of the current generation. The expression (*iterate_0 nextstate* $\langle 0,1\rangle$) computes the sequence of changes in the fixed point approximation in the form of the sequence of state tuples:

$$\langle 0,1\rangle, \quad \langle 1,1\rangle, \quad \langle 1,2\rangle, \ldots$$

This is a form of program synthesis as found in for instance Bird and Wadler (1988, pp. 131–132). Our method is a powerful generalization of the procedure described there.

## 3 Practical considerations: parallelism

There are two practical aspects to communication lifting. The first is the reduction in the cost of managing a large number of streams, because after the transformation only a single stream remains to be managed. We will come back to this issue in Section 7.

The second aspect is the possibility to evaluate the components of the state tuple in parallel. This has to be contrasted with the pipe-line parallelism of a process network. Before discussing this point further, the *fib* example must be extended slightly to introduce a possibility for parallel evaluation. The example as it stands does not allow parallel evaluation at all because only one addition is performed (on a stream of numbers). The extension consists of adding two more streams $\tilde{o}$ and $\tilde{p}$ to the network, to produce a running total of the Fibonacci sequence in the stream $\tilde{o}$:

$$\begin{aligned}
\tilde{o} &= 0 : \tilde{p} \\
\tilde{p} &= \mathit{map\_2}\ (+)\ \tilde{o}\ \tilde{b} \\
\tilde{a} &= 0 : \tilde{b} \\
\tilde{b} &= 1 : \tilde{c} \\
\tilde{c} &= \mathit{map\_2}\ (+)\ \tilde{a}\ \tilde{b}
\end{aligned}$$

The original sub expression *map_2* $(+)\ \tilde{a}\ \tilde{b}$ has been put in a separate equation $\tilde{c}$. This makes it easier to draw a diagram for the network. The Fibonacci-sum program thus obtained will serve as the running example of the paper. The program generates a stream of Fibonacci numbers $\tilde{a} = 0,1,1,2,3,5,8\ldots$ and then adds these numbers to produce the stream $\tilde{o} = 0,1,2,4,7,12,20\ldots$.

To compare different ways of parallel evaluation it is illustrative to look at diagrams of the untransformed process network and the communication lifted version. The diagram for a synchronous process network shows the streams as connections between the functions applied to the streams. The diagram of the Fibonacci-sum network is shown in Figure 1. The name of a stream in the network is used as the label on the corresponding edge. The label in a box is (the curried version of) the appropriate stream processing function.

Figure 1 shows that the two processes *map_2* $(+)$ may perform the additions with

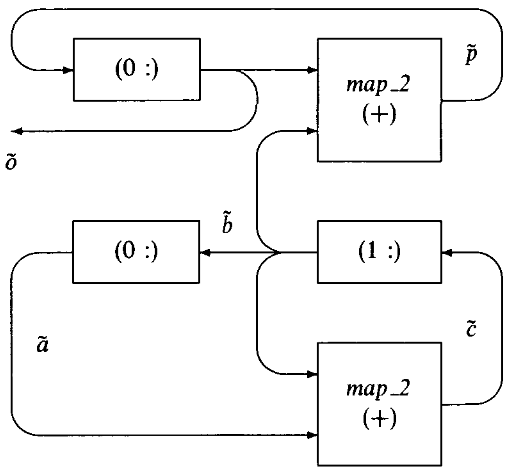


Fig. 1. The original Fibonacci-sum network with pipe-line parallelism.

pipe-line parallelism. With this example no speedup can be achieved in practice as the additions represent too little work. In practical networks sufficient coarse grain work should be available for the pipe-line to deliver speedups.

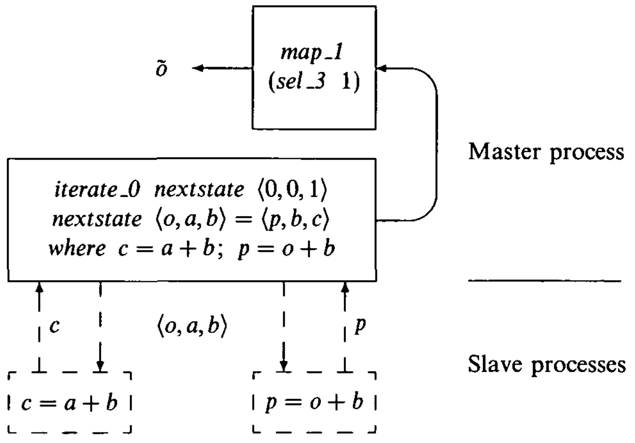
Communication lifting of the extended example produces the following program:

$$\begin{aligned}
\tilde{o} &= \mathit{map\_1}\ (\mathit{sel\_3}\ 1)\ (\mathit{iterate\_0}\ \mathit{nextstate}\ \langle 0,0,1\rangle)\\
\mathit{nextstate}\ \langle o,a,b\rangle &= \langle p,b,c\rangle\\
&\quad \mathit{where}\\
&\quad p = (+)\ o\ b\\
&\quad c = (+)\ a\ b
\end{aligned}$$

This transformation is analogous to that of the Fibonacci program. The formal derivation will be given in Section 5.

The communication lifted program can be executed with master-slave style parallelism as follows. The master process is responsible for generating the successive states, so it executes the calls to *iterate_0* and *nextstate*. The first state is the constant tuple $\langle 0,0,1\rangle$, to which the master thus applies *nextstate*. This calls on two slaves to perform the two additions $p = o + a$ and $c = a + b$. The slaves have both access to the current tuple where the values of $o$, $a$ and $b$ are stored. Once the slave processors are finished, the next tuple is ready. The master process now selects the appropriate component of the new tuple for output via the application of *map_1* (*sel_3* 1). The cycle is then complete and the whole sequence may begin again. This is schematically shown in Figure 2. Here the dashed lines represent the flow of data used by the master-slave communication. The solid lines are streams as before.

The master-slave parallelism shown in Figure 2 corresponds exactly to the pipe-line that is present in the original program of Figure 1, where two processes *map_2* (+) and *map_2* (+) are connected by streams. A pipe-line has thus been transformed into a master slave relation by communication lifting. The calculations in each step



$\tilde{o}$
*map_1*
(*sel_3* 1)
*iterate_0 nextstate* ⟨0, 0, 1⟩
*nextstate* ⟨*o*, *a*, *b*⟩ = ⟨*p*, *b*, *c*⟩
*where* *c* = *a* + *b*; *p* = *o* + *b*
Master process
*c*
⟨*o*, *a*, *b*⟩
*p*
*c* = *a* + *b*
*p* = *o* + *b*
Slave processes

Fig. 2. Fibonacci-sum network after communication lifting with master-slave parallelism shown using dashed lines.

as performed by the slaves, are only dependent on the output produced by the previous step and are otherwise completely independent of each other.

It is possible to build a parallel implementation with a special primitive function that implements the required *master-slave* style parallelism. The tuples must then be constructed such, that at least two components require enough computation to outweigh the parallel overhead. The independence of the calculations on the tuple components should allow for a relatively cheap and simple mechanism to implement the parallel evaluation. Parallelism of a more general nature, such as pipe-line parallelism, is more difficult to harness efficiently. A more rigid paradigm (master-slave) allows the implementation more scope for optimisations than a more lenient paradigm (pipe-line). A more lenient paradigm offers the programmer better possibilities for clarity and conciseness. These claims are substantiated in Section 7.

## 4 The definition of a synchronous process network

In this section the notations involved in the communication lifting program transformation are formally introduced. A network of synchronous processes is a graph, with synchronous processes as vertices and streams as edges. A process is synchronous if it is one of (:), *map_n*, *iterate_n* or *tl*. A synchronous process network should be represented by a number of equations over streams, according to the syntax in Figure 3. There must be one equation in the network for the stream $\tilde{o}$, which by convention, generates the output of the network. No two equations in a network may have the same left hand side. The network graph must be connected, with the equation for $\tilde{o}$ as the root. Equations of the form $\tilde{v} = \tilde{w}$ are not permitted, they can always be eliminated by substitution. All these restrictions are necessary to allow communication lifting to be implemented as an automatic program transformation, for instance as part of a compiler.

The free stream variables of a synchronous process network are taken to be

$$\begin{array}{lcll}
p & ::= & d_1 \ \dots \ d_n & \\
d_i & ::= & \tilde{v}_i = e_i & \text{(equations of the form } \tilde{v} = \tilde{w} \text{ are not permitted)} \\
e & ::= & s \ : \ e & \\
 & | & \mathit{map\_n} \ f \ e_1 \ \dots \ e_n & \\
 & | & \mathit{iterate\_n} \ f \ s \ e_1 \ \dots \ e_n & \\
 & | & \mathit{tl} \ e & \\
 & | & \tilde{v} & \\
s & \in & V & \text{(arbitrary values)} \\
\tilde{f}, \tilde{g} & \in & \tilde{F} & \text{(arbitrary functions on streams)} \\
f, g & \in & F & \text{(arbitrary functions on stream elements)} \\
\tilde{o}, \tilde{v}, \tilde{w}, \tilde{x}, \tilde{y}, \tilde{z} & \in & \tilde{D} & \text{(streams)} \\
s, v, w, x, y, z & \in & D & \text{(arbitrary stream elements)} \\
i, j, k, l, m, n & \in & \mathbb{N} & \text{(natural numbers)}
\end{array}$$

Fig. 3. Abstract syntax of the process network language.

provided as input to the network from outside. A network may have any number of input streams, but need not have any.

The definitions of a number of useful stream processing and auxiliary functions are shown in Figure 4. According to the syntax, only *tl*, (:), *map_n* and *iterate_n* can be used as stream processing functions. The other functions and operators are also shown here because of their use in the transformation process. The choice of functions that may be applied to streams seems rather limited. However, considering that the functions that may be applied to *stream elements* are not constrained, the abstract syntax is actually quite general. Only three functions are required to build a synchronous process network: one to extend a stream up front (:), one to trim the first element off a stream (*tl*) and a third function to perform an arbitrary computation on a stream element (*map_n*). A fourth function *iterate_n* has been included because it captures the concept of a function that carries its own local state.

To be completely general, functions that add and remove arbitrary stream elements should have been supported as well, for instance *filter*. We have chosen not to include such functions as it makes it more difficult to guarantee that the networks constructed are synchronous. Work is in progress on an extension of the method to also support some forms of asynchronous networks, in which functions such as *filter* play a role.

Functions with a suffix *_n* represent a whole family of functions, because $n$ is a natural number. In an enumeration such as $e_1 \dots e_n$, $n$ may also be equal to 0, which means that there is not even a single expression $e_i$ present. The function *sel_0* is ill defined, but *map_0* and *iterate_0* are valid functions. Note that the definition of *map_n* does not correspond with the usual definition, because there is no test whether any of the input streams $\tilde{v}_1 \dots \tilde{v}_n$ are empty. This is consistent with the view that streams are infinite (lists).

## 5 Communication lifting of a synchronous process network

The communication lifting transformation of a synchronous process network, as defined according to the syntax of Figure 3, will now be presented in two steps. As

$$
\begin{array}{lll}
\mathit{hd}\ (x : \tilde{x}) & = x & \\
\mathit{tl}\ (x : \tilde{x}) & = \tilde{x} & \\
\tilde{x}!0 & = \mathit{hd}\ \tilde{x} & \\
\tilde{x}!(i+1) & = (\mathit{tl}\ \tilde{x})!i & \\
\mathit{take}\ 0\ \tilde{x} & = \mathit{nil} & \\
\mathit{take}\ (i+1)\ \tilde{x} & = \mathit{hd}\ \tilde{x}\ :\ \mathit{take}\ i\ (\mathit{tl}\ \tilde{x}) & \\
\mathit{sel\_n}\ i\ \langle v_1,\ldots,v_n\rangle & = v_i & (\forall n \geq 1 \wedge 1{\leq}i{\leq}n \text{ select } i\text{-th component}) \\
\mathit{map\_n}\ f\ \tilde{v}_1\ \ldots\ \tilde{v}_n & = w : \mathit{map\_n}\ f\ (\mathit{tl}\ \tilde{v}_1)\ldots(\mathit{tl}\ \tilde{v}_n) & \\
 & \quad \mathit{where} & \\
 & \quad w\ =\ f\ (\mathit{hd}\ \tilde{v}_1)\ldots(\mathit{hd}\ \tilde{v}_n) & \\
\mathit{iterate\_n}\ f\ s\ \tilde{v}_1\ \ldots\ \tilde{v}_n & = s : \mathit{iterate\_n}\ f\ s'\ (\mathit{tl}\ \tilde{v}_1)\ldots(\mathit{tl}\ \tilde{v}_n) & \\
 & \quad \mathit{where} & \\
 & \quad s'\ =\ f\ s\ (\mathit{hd}\ \tilde{v}_1)\ldots(\mathit{hd}\ \tilde{v}_n) & \\
\mathit{zip\_n}\ \langle \tilde{v}_1,\ldots,\tilde{v}_n\rangle & = w : \mathit{zip\_n}\ \langle \mathit{tl}\ \tilde{v}_1,\ldots,\mathit{tl}\ \tilde{v}_n\rangle & \\
 & \quad \mathit{where} & \\
 & \quad w\ =\ \langle \mathit{hd}\ \tilde{v}_1,\ldots,\mathit{hd}\ \tilde{v}_n\rangle &
\end{array}
$$

Fig. 4. Definition of stream processing and auxiliary functions and operators.

the first step, we present the communication lifting of a simplified process network. The second step (Section 6) brings a more general network that conforms to the abstract syntax of Figure 3 into the simplified form.

The communication lifting transformation proper is given by Rule *T0* of Figure 5. The notation employed is more or less standard (see for example Ferguson and Wadler (1988)). The transformation rules take a syntactic argument enclosed in emphatic brackets $[\![$ and $]\!]$. Pattern matching is used to choose between alternative clauses. The matching order is top down, thus Clause 0b is a *catch-all* clause, fitting any network that does not match Clause 0a. The matching of some clauses (e.g. Clause 0a) is further constrained by a guard, written as a conditional $(\mathit{if}\ldots) \Longrightarrow$ connecting the left- and right-hand sides of the clause. A clause protected with a guard matches only if both the pattern and the guard are satisfied. If either fails, the next clause will be tried.

A simplified synchronous process network must match the left hand side of Clause 0a. This means that the first $k$ equations must contain an application of (:) and that the next $m$ equations must contain an application of *map_n*. The remaining $l$ equations must use *tl*. At this stage applications of *iterate_n*, equations of the form $\tilde{v} = \tilde{w}$, or nested expressions are not permitted. These restrictions are necessary to make the presentation of the communication lifting transformation proper reasonably succinct. The lifting of the restrictions is the subject of the next section.

Some of the stream variables $\tilde{x}_i$ and $\tilde{y}_{ij}$ in the left hand side of Clause 0a will be the same as some of the variables $\tilde{o}$, $\tilde{v}_i$, $\tilde{w}_i$ or $\tilde{t}_i$. The $\tilde{x}_i$ and $\tilde{y}_{ij}$ that are not defined within the network act as the input streams to the network. These input streams are identified by the set $\{\tilde{u}_1\ldots\tilde{u}_h\}$.

The guard in Clause 0a ensures that the streams $\tilde{z}_1\ldots\tilde{z}_l$, and the states $s_1\ldots s_k$, and

$$
\begin{array}{llll}
TO\ [\![ & \tilde{o} & = s_1 : \tilde{x}_1 & \qquad\qquad (0a) \\
& \tilde{v}_2 & = s_2 : \tilde{x}_2 & \\
& \vdots & & \\
& \tilde{v}_k & = s_k : \tilde{x}_k & \\
& \tilde{w}_1 & = map\_n_1\ f_1\ \tilde{y}_{11}\ldots\tilde{y}_{1n_1} & \\
& \vdots & & \\
& \tilde{w}_m & = map\_n_m\ f_m\ \tilde{y}_{m1}\ldots\tilde{y}_{mn_m} & \\
& \tilde{t}_1 & = tl\ \tilde{z}_1 & \\
& \vdots & & \\
& \tilde{t}_l & = tl\ \tilde{z}_l\ ]\!] \quad (\mathit{if} & \{\tilde{z}_1\ldots\tilde{z}_l\} \cap \mathcal{N} = \emptyset\ \wedge \\
& & & \mathit{freevars}\{s_1\ldots s_k\} \cap \mathcal{N} = \emptyset\ \wedge \\
& & & \mathit{freevars}\{f_1\ldots f_m\} \cap \mathcal{N} = \emptyset) \quad \Longrightarrow
\end{array}
$$

$$
\begin{array}{lll}
\tilde{o} & = map\_1\ (sel\_k\ 1)\ (iterate\_h\ nextstate\ \langle s_1,\ldots,s_k\rangle\ \tilde{u}_1\ldots\tilde{u}_h) & \\
\tilde{t}_1 & = tl\ \tilde{z}_1 & \\
\vdots & & \\
\tilde{t}_l & = tl\ \tilde{z}_l & \\
\multicolumn{2}{l}{nextstate\ s\ u_1\ldots u_h\ = \langle x_1,\ldots,x_k\rangle} & \\
& \quad where & \\
& \quad \langle o, v_2,\ldots,v_k\rangle\ = \ s & \\
& \quad w_1\ = f_1\ y_{11}\ldots y_{1n_1} & \\
& \quad \vdots & \\
& \quad w_m\ = f_m\ y_{m1}\ldots y_{mn_m} &
\end{array}
$$

where $\tilde{u}_1\ldots\tilde{u}_h$ are defined as

$$
\begin{array}{ll}
\{\tilde{u}_1\ldots\tilde{u}_h\} & = (\{\tilde{x}_1\ldots\tilde{x}_k\} \cup \{\tilde{y}_{ij} : i \in \{1\ldots m\} \wedge j \in \{1\ldots n_i\}\}) \setminus (\mathcal{N} \cup \{\tilde{t}_1\ldots\tilde{t}_l\}) \\
\mathcal{N} & = \{\tilde{w}_1\ldots\tilde{w}_m\} \cup \{\tilde{o}, \tilde{v}_2\ldots\tilde{v}_k\}
\end{array}
$$

$$TO\ [\![\ p\ ]\!]\ \ (\mathit{otherwise}) \Longrightarrow\ \ p \qquad\qquad (0b)$$

Fig. 5. Communication lifting transformation.

the functions $f_1\ldots f_m$ are not dependent on any of the streams $\tilde{o}, \tilde{v}_2\ldots\tilde{v}_k$ and $\tilde{w}_1\ldots\tilde{w}_m$. This is necessary because the transformation removes the definitions of the streams $\tilde{v}_i$ and $\tilde{w}_j$.

The best way to understand Rule *TO* is to try it out on a simple example. The program produced by applying Rule *TO* to the Fibonacci network of Section 3 yields:

$$
\begin{array}{lll|lll}
\tilde{o} & = & s_1 : \tilde{x}_1 & \tilde{o} & = & 0 : \tilde{p} \\
\tilde{v}_2 & = & s_2 : \tilde{x}_2 & \tilde{a} & = & 0 : \tilde{b} \\
\tilde{v}_3 & = & s_3 : \tilde{x}_3 & \tilde{b} & = & 1 : \tilde{c} \qquad \overset{TO}{\Longrightarrow} \\
\tilde{w}_2 & = & map\_2\ f_1\ \tilde{y}_{11}\ \tilde{y}_{12} & \tilde{p} & = & map\_2\ (+)\ \tilde{o}\ \tilde{b} \\
\tilde{w}_2 & = & map\_2\ f_2\ \tilde{y}_{21}\ \tilde{y}_{22} & \tilde{c} & = & map\_2\ (+)\ \tilde{a}\ \tilde{b}
\end{array}
$$

Here we have shown the correspondence between formal and actual identifiers of the Rule *T0*. The guard of Clause 0a is satisfied: because there are no equations of type $\tilde{t} = tl\ \tilde{z}$, and because $s_1 = 0$, $s_2 = 0$, $s_3 = 1$, $f_1 = (+)$, and $f_2 = (+)$ do not depend on $\mathcal{N} = \{\tilde{o}, \tilde{a}, \tilde{b}, \tilde{p}, \tilde{c}\}$. For the Fibonacci-sum program the set $\{\tilde{u}_1 \ldots \tilde{u}_h\}$ is empty, which means that the network does not use external input streams:

$$\xrightarrow{T0} \quad \begin{array}{l} \tilde{o} \;=\; map\_1\ (sel\_3\ 1)\ (iterate\_0\ nextstate\ \langle 0,0,1\rangle) \\ nextstate\ s \;=\; \langle p,b,c\rangle \\ \qquad\qquad\quad where \\ \qquad\qquad\quad \langle o,a,b\rangle \;=\; s \\ \qquad\qquad\quad p \;=\; (+)\ o\ b \\ \qquad\qquad\quad c \;=\; (+)\ a\ b \end{array}$$

## 6 Simplifying transformations

A synchronous process network specified according to the syntax of Figure 3 has to be transformed into a simplified form that is acceptable to Rule *T0*. Figure 6 shows the simplifications that are performed by the successive application of the Rules *T1*, *T3* and *T5*. We will explain the purpose of each of these rules in turn.

First we note, that the Fibonacci-sum program as discussed in Section 3 is actually the outcome of applying the simplifying transformations to the following program:

$$\begin{array}{l} \tilde{o} = iterate\_1\ (+)\ 0\ (tl\ \tilde{a}) \\ \tilde{a} = 0 : 1 : map\_2\ (+)\ \tilde{a}\ (tl\ \tilde{a}) \end{array} \quad \xrightarrow{T1}$$

This program cannot be transformed directly by Rule *T0* because: it uses the *iterate_1* function; it uses nested function applications and the output stream $\tilde{o}$ is not an application of (:). The purpose of the simplifying transformations is to eliminate these constructs.

### *Rule T1: removing nested function applications*

The first problem to solve is to remove nested function applications. This is the purpose of Rule *T1*, which defines a new equation for each nested expression.

Rule *T1* applies Rule *T2* to all equations of the network. Clause 2a introduces extra stream equations for all nested expressions that occur in the synchronous process network. The pattern $\tilde{v} = f \cdots (g \ldots) \cdots$ is matched by an equation that starts with an application of some function $f$ and that contains a nested application $(g \ldots)$. If there is more than one nested application that can be matched by $(g \ldots)$, the left most is chosen. This choice is arbitrary as *T2* is reapplied to the results $\tilde{v} = f \cdots \tilde{w} \cdots$ and $\tilde{w} = (g \ldots)$ so that remaining nested applications will be dealt with eventually. According to the syntax, the functions $f$ and $g$ must be one of *tl*, (:), *map_n* or *iterate_n*. On the right-hand side of Clause 2a the newly introduced equation bears a name $\tilde{w}$ that must not appear anywhere else in the network. Rule *T2* is applied recursively until all arguments of each application of $f$ are simple stream identifiers. The default Clause 2b terminates the recursion.

$$
\begin{array}{lll}
\mathit{T1}\ \llbracket\ d_1 \ldots d_n\ \rrbracket & \Rightarrow \mathit{T2}\ \llbracket d_1 \rrbracket\ \ldots\ \mathit{T2}\ \llbracket d_n \rrbracket & (1) \\
\mathit{T2}\ \llbracket\ \tilde{v}\ =\ f \cdots (g \ldots) \cdots\ \rrbracket & \Rightarrow \mathit{T2}\ \llbracket\ \tilde{v}\ =\ f \cdots \tilde{w} \cdots\ \rrbracket & (2a) \\
 & \phantom{\Rightarrow}\ \mathit{T2}\ \llbracket\ \tilde{w}\ =\ (g \ldots)\ \rrbracket & \\
 & \phantom{\Rightarrow}\ \text{where } \tilde{w} \text{ is a new variable} & \\
\mathit{T2}\ \llbracket\ \tilde{v}\ =\ e\ \rrbracket & \Rightarrow \tilde{v}\ = e & (2b) \\
\mathit{T3}\ \llbracket\ d_1\ \ldots\ d_n\ \rrbracket & \Rightarrow \mathit{T4}\ \llbracket d_1 \rrbracket\ \ldots\ \mathit{T4}\ \llbracket d_n \rrbracket & (3) \\
\mathit{T4}\ \llbracket\ \tilde{v}\ =\ \mathit{iterate\_n}\ f\ s\ \tilde{x}_1 \ldots \tilde{x}_n\ \rrbracket & \Rightarrow \tilde{v}\ = s : \tilde{w} & (4a) \\
 & \phantom{\Rightarrow}\ \tilde{w}\ = \mathit{map\_m}\ f\ \tilde{v}\ \tilde{x}_1 \ldots \tilde{x}_n & \\
 & \phantom{\Rightarrow}\ \text{where } \tilde{v} \text{ and } \tilde{w} \text{ are new variables and } m = n + 1 & \\
\mathit{T4}\ \llbracket\ d\ \rrbracket & \Rightarrow d & (4b) \\
\mathit{T5}\ \llbracket\ \tilde{o}\ =_k\ \mathit{map\_n}\ f\ \tilde{x}_1 \ldots \tilde{x}_n & & (5a) \\
\qquad d_2\ \ldots\ d_m\ \rrbracket\ \rho & \Rightarrow \mathit{T5}\ \llbracket\ \tilde{o}\ =_k\ f\ (\mathit{hd}\ \tilde{x}_1)\ \ldots\ (\mathit{hd}\ \tilde{x}_n) : \tilde{v} & \\
 & \qquad\quad \tilde{v}\ =_k\ \mathit{tl}\ \tilde{w} & \\
 & \qquad\quad \tilde{w}\ =_k\ \mathit{map\_n}\ f\ \tilde{x}_1 \ldots \tilde{x}_n & \\
 & \qquad\quad d_2\ \ldots\ d_m\ \rrbracket\ \rho & \\
 & \phantom{\Rightarrow}\ \text{where } \tilde{v} \text{ and } \tilde{w} \text{ are new variables} & \\
\mathit{T5}\ \llbracket\ \tilde{o}\ =_k\ \mathit{tl}\ \tilde{w} & & (5b) \\
\qquad \tilde{w}\ =_j\ s : \tilde{x} & & \\
\qquad d_3\ \ldots\ d_m\ \rrbracket\ \rho & \Rightarrow \mathit{T5}\ \llbracket\ (\tilde{w}\ =_j\ s : \tilde{o} & \\
 & \qquad\quad d_3\ \ldots\ d_m)\ [\tilde{o}/\tilde{x}]\ \rrbracket\ \rho & \\
\mathit{T5}\ \llbracket\ \tilde{v}\ =_k\ \mathit{tl}\ \tilde{w} & & (5c) \\
\qquad \tilde{w}\ =_j\ s : \tilde{x} & & \\
\qquad d_3\ \ldots\ d_m\ \rrbracket\ \rho & \Rightarrow \mathit{T5}\ \llbracket\ (\tilde{w}\ =_j\ s : \tilde{x} & \\
 & \qquad\quad d_3\ \ldots\ d_m)\ [\tilde{x}/\tilde{v}]\ \rrbracket\ \rho & \\
\mathit{T5}\ \llbracket\ \tilde{v}\ =_k\ \mathit{tl}\ \tilde{w} & & (5d) \\
\qquad \tilde{w}\ =_j\ \mathit{map\_n}\ f\ \tilde{x}_1 \ldots \tilde{x}_n & & \\
\qquad d_3\ \ldots\ d_m\ \rrbracket\ \rho & & \\
\qquad\qquad (\mathit{if}\ (j,k) \notin \rho) & \Longrightarrow \mathit{T5}\ \llbracket\ \tilde{w}\ =_j\ f\ (\mathit{hd}\ \tilde{x}_1) \ldots (\mathit{hd}\ \tilde{x}_n) : \tilde{v} & \\
 & \qquad\quad \tilde{v}\ =_j\ \mathit{map\_n}\ f\ \tilde{y}_1 \ldots \tilde{y}_n & \\
 & \qquad\quad \tilde{y}_1\ =_k\ \mathit{tl}\ \tilde{x}_1 & \\
 & \qquad\qquad\quad \vdots & \\
 & \qquad\quad \tilde{y}_n\ =_k\ \mathit{tl}\ \tilde{x}_n & \\
 & \qquad\quad d_3\ \ldots\ d_m\ \rrbracket\ (\rho \cup \{(j,k)\}) & \\
 & \phantom{\Rightarrow}\ \text{where } \tilde{y}_1 \ldots \tilde{y}_n \text{ are new variables} & \\
\mathit{T5}\ \llbracket\ p\ \rrbracket\ \rho \qquad (\mathit{otherwise}) & \Longrightarrow p & (5e)
\end{array}
$$

Fig. 6. Simplifying transformations: $p' = \mathit{T5}\ (\mathit{T3}\ (\mathit{T1}\ p))\ \emptyset$.

Applying Rule *T1* to the Fibonacci-sum program as given above introduces four new equations for $\tilde{b}$, $\tilde{c}$, $\tilde{d}$ and $\tilde{q}$ to remove the nested function applications. The result of this transformation is:

$$
\xrightarrow{\mathit{T1}}\quad
\begin{array}{l}
\tilde{o} = \mathit{iterate\_1}\ (+)\ 0\ \tilde{q} \\
\tilde{q} = \mathit{tl}\ \tilde{a} \\
\tilde{a} = 0 : \tilde{b} \\
\tilde{b} = 1 : \tilde{c} \\
\tilde{c} = \mathit{map\_2}\ (+)\ \tilde{a}\ \tilde{d} \\
\tilde{d} = \mathit{tl}\ \tilde{a}
\end{array}
\quad\xrightarrow{\mathit{T3}}
$$

### *Rule T3: removing calls to the function* iterate_n

The purpose of Rule *T3* is to simplify the network by replacing all occurrences of the function *iterate_n* by applications of the simpler *map_m* function.

Rule *T3* applies Rule *T4* to all the equations of the network, which after application of Rule *T1* contain a single function application each. Each equation with *iterate_n* is transformed into an equation with (:) and an equation with *map_m*. Clause 4b retains all other equations as they are.

Application of Rule *T3* to the Fibonacci-sum network as delivered by Rule *T1* replaces the equation for $\tilde{o}$ by two new equations. The equations for $\tilde{a} \ldots \tilde{d}$ and $\tilde{q}$ remain unchanged:

$$\xrightarrow{T3} \quad \begin{aligned} \tilde{o} &= 0 : \tilde{p} \\ \tilde{p} &= map\_2 \;\; (+) \; \tilde{o} \; \tilde{q} \\ \tilde{q} &= tl \;\; \tilde{a} \\ \tilde{a} &= 0 : \tilde{b} \\ \tilde{b} &= 1 : \tilde{c} \\ \tilde{c} &= map\_2 \;\; (+) \; \tilde{a} \; \tilde{d} \\ \tilde{d} &= tl \;\; \tilde{a} \end{aligned} \quad \xrightarrow{T5}$$

### *Rule T5: removing redundant calls to* tl

The goal of the final simplifying Rule *T5* is to remove as many applications of *tl* as possible. Rule *T5* also ensures that the output stream is defined as an application of (:). These apparently different purposes have to be served by one transformation, as both involve calls to the function *tl*.

Before discussing Rule *T5* proper, a further explanation is appropriate about the pattern matching of clauses such as Clause 5c, which process several equations simultaneously. The variable $\tilde{w}$ on the left-hand side of the Clause 5c occurs twice in the pattern, which means that both occurrences must match the same stream. This links two equations of the process network. Thus far no ordering on the equations in the process network has been assumed, because rules *T2* and *T4* can be applied in any order. For Rule *T5* it is convenient to regard the equations that define the process network under consideration as a proper set. Any subset of equations satisfying the constraints specified by the patterns may be chosen. Clause 5c will thus select any equation $\tilde{v} = tl \;\; \tilde{w}$ and the corresponding equation $\tilde{w} = s : \tilde{x}$. The remaining equations are named $d_3 \ldots d_m$ and retained so that they can be processed by the recursive call to Rule *T5*.

Figure 6 shows that all equations of Rule *T5* are labelled with a subscript $(=_i)$. This is necessary to ensure that Rule *T5* will terminate on all inputs. The labels are used in the guard $(if \;\; (j,k) \notin \rho) \Longrightarrow$ of Clause 5d to avoid this clause from looping on a pair of definitions such as $\tilde{v} = tl \;\; \tilde{w}$; $\tilde{w} = map\_1 \;\; f \;\; \tilde{v}$. The labelling can be added to a system of equations, by numbering each equation and using the equation number as the label number. The only assumption about the labels is that they are all different when they are first assigned.

Clause 5a unfolds the definition of *map_n* once and introduces two new equations so that $\tilde{o} = \ldots : \ldots$ as required.

Clause 5c resolves a combination of $\tilde{v} = tl\ \ \tilde{w}$ and $\tilde{w} = s : \tilde{x}$ by replacing every free occurrence of $\tilde{v}$ in the process network by $\tilde{x}$. This replacement is expressed as $(\ldots)\ [\tilde{x}/\tilde{v}]$. The equation for $\tilde{v}$ is then removed. The *tl* is thus cancelled against the (:). A combination of $\tilde{v} = tl\ \ \tilde{w}$ and $\tilde{w} = map\_n\ \ldots$ (Clause 5d) can be resolved in a similar way, after unfolding the definition of *map_n* once. Then the *tl* can be cancelled against the (:).

Clause 5b is a special case of Clause 5c. Both clauses cancel an application of *tl* against an application of (:), but the output stream of the network $\tilde{o}$ must be treated specially. If Clause 5b were omitted, Clause 5c applied to $\tilde{v} = tl\ \ \tilde{o}$ would remove the equation for $\tilde{o}$ from the network. This would make it impossible for Rule *T0* later to retrieve the output stream from the network.

Rule *T5* has no case for combinations such as $\tilde{v} = tl\ \ \tilde{w}$ with $\tilde{w} = tl\ \ \tilde{x}$, because a combination of two or more *tl* applications is resolved by cancellation of the last *tl* against either *map_n* or (:), followed by cancellation of the penultimate *tl* etc. A combination of equations involving applications of *tl* can not be removed by cancellation if the last *tl* is applied to an input stream. This case is adequately handled by Rule *T0* and will not concern Rule *T5*.

When applied to the Fibonacci-sum example program, Rule *T5* cancels the applications of *tl* by using Clause 5c twice. This yields the Fibonacci-sum network in a compact form, ready for the final Rule *T0*. It is the same program as the one we started with in Section 3 and also in Section 5.

$$\xrightarrow{T5}\quad \begin{aligned} \tilde{o} &= 0 : \tilde{p} \\ \tilde{p} &= map\_2\ \ (+)\ \tilde{o}\ \tilde{b} \\ \tilde{a} &= 0 : \tilde{b} \\ \tilde{b} &= 1 : \tilde{c} \\ \tilde{c} &= map\_2\ \ (+)\ \tilde{a}\ \tilde{b} \end{aligned}$$

The simplifying transformations as defined in Figure 6 are necessary and sufficient to bring a system of equations over streams as specified according to the syntax of Figure 3 into the form required by the communication lifting transformation proper as given by Rule *T0*.

## 7 The performance of a number of small networks

The communication lifting transformation consists of a number of fold/unfold steps (Burstall and Darlington, 1977) and uses algebraic properties of stream functions such as *map_n* and *zip_n*. The communication lifting transformation incorporates a strategy that decides which steps to take, to guarantee delivery of an equivalent but completely restructured program. Communication lifting can thus be viewed as a transformation skeleton (Darlington *et al.*, 1991).

Communication lifting as a programming tool is only useful if the transformed programs will run faster, and/or use less space, than the original programs. Three

Table 1. *Synchronous process networks with an indication of their size and purpose. Execution times are reported in milliseconds.*

| system program | source lines | sequential orig. (ms) | sequential trans. (ms) | master-slave trans. (ms) | pipe orig. (ms) | tuple size | stream length |
|---|---|---|---|---|---|---|---|
| `fibsum`[a] | 6 | 627 | 493 | 477 | 536 | 3 | 3000 |
| `flipflop`[b] | 28 | 2434 | 760 | 581 | 974 | 13 | 1600 |
| `fft`[c] | 190 | 3045 | 2956 | 1364 | 1504 | 8 | 5 |
| `wave4`[d] | 264 | 2041 | 2131 | 1111 | 1151 | 3 | 200 |

[a] Sums the first 3000 Fibonacci numbers.
[b] Simulation of a D-flipflop over 1600 state transitions (Vree, 1989).
[c] A 1024-point fast Fourier transform using arrays (Hartel and Vree, 1992).
[d] Predicts the water heights and velocities in a square area of $4 \times 4$ grid points of the North Sea over 200 time steps (Vree, 1989).

important performance issues can be distinguished. The first is the gain or loss in *sequential* performance due to transformation, the second is the difference between sequential and master-slave parallel performance, the third issue is the difference between master-slave parallel performance of the transformed programs and pipe-line parallel performance of the untransformed programs. These issues will be discussed in the three following sections. To make measurements possible, communication lifting has been implemented and applied to a small set of synchronous process networks. The transformation has been implemented in Miranda[†] (Turner, 1985) and the input and output of the transformations are also Miranda programs. The abstract syntax of Figure 3 can be embedded in that of Miranda.

## 7.1 Sequential performance

To asses the impact of communication lifting on sequential performance, the set of synchronous process networks has been compiled and executed both before and after the complete set of transformations. The programs are compiled by the FAST compiler (Hartel *et al.*, 1991; Langendoen and Hartel, 1992), which amongst others, provides efficient arrays for the benefit of the `fft` and the `wave4` applications. The programs are executed on a stand alone Motorola 88000 processor board with 64Mb of memory. This allows execution time to be measured with an accuracy of 1 millisecond. For each program, Table 1 gives an indication of the size, the execution times of four versions of the program, the size of the state tuple, the number of elements of the streams that are evaluated and an explanation of the purpose of the program. The programs are sorted by the number of lines of source text. This count is exclusive of standard library functions as provided by Miranda, comments and blank lines.

The columns *sequential/orig.* and *sequential/trans.* show the sequential execution

[†] Miranda is a trademark of Research Software Ltd.

times (in milliseconds) of a program before and after transformation. The sequential performance of most programs is improved, which shows that communication lifting is a viable optimisation technique for sequential programs. The significant performance gain in the `flipflop` program is due to the fact that instead of managing 13 lists, as is the case before the transformation, the transformed program only needs to manage 2 lists, which can be done more efficiently. The effect is strongest for the `flipflop` program, because it uses more streams than the others. For the `fibsum` program, there are three streams before the transformation and still two streams after the transformation. So only a small reduction of the stream management effort is the result.

The large networks that occur in real programs will lead to huge tuples. However, the implementation creates the state tuples in a single heap allocation, whereas a network of streams gives rise to more heap claims of smaller cells, which is thus more costly. The selection of an element of a tuple is performed in unit time, which is the same as in a stream network. When a small amount of work is involved in computing the stream elements, communication lifting will allow sequential performance gains on practical programs. For programs that involve large amounts of work on the stream elements, such as `wave4` and `fft`, sequential performance will not be affected much.

## 7.2 *Parallel performance of the master-slave system*

The sequential performance improvement for most of the programs indicates that communication lifting is a valid point of departure for parallel evaluation of independent tuple elements. Parallel evaluation always introduces overhead. This overhead should be kept small in comparison to the amount of real work involved in the evaluation of the state tuples. For example, two of the three tuple elements in the transformed `wave4` program represent a large amount of computation. To assess the parallel performance of the programs after communication lifting, an implementation has been built that supports the required master-slave parallelism. The Motorola 88000 system has four CPUs, four instruction caches and four coherent data caches and 64 MB shared memory. Cache coherency is handled by the hardware. The four processors are numbered $0, 1, 2$ and 3. Processor 0 is the master processor, which begins execution. The other three processors are initially idle. The runtime system of the FAST compiler supports master-slave parallelism through a special primitive function *pforce*, which when applied to a tuple allocates the evaluation of each component of the tuple to a separate processor. The scheduling strategy is as follows: processor $p$ first evaluates component $p$ to full normal form (not just to head normal form). Then, as soon as this terminates, processor $p$ evaluates component $p+4$ to full normal form, then $p+8$ etc. Parallel evaluation continues until all components of the state tuple have been evaluated to full normal form. The function *pforce* thus has a completely strict semantics.

During parallel evaluation the master processor behaves as an ordinary slave processor. Processor 0 becomes master again as soon as all tuple components have been evaluated, at which point a new tuple is formed by the master processor. All other processors remain idle until the master encounters the next application of *pforce*.

In the communication lifted version of the Fibonacci-sum program the forming of the tuple is expressed as follows:

$$\begin{aligned}
\tilde{o} &= \mathit{map\_1}\ (\mathit{sel\_3}\ 1)\ (\mathit{iterate\_0}\ \mathit{nextstate}\ \langle 0,0,1\rangle)\\
\mathit{nextstate}\ s &= \mathit{pforce}\ \langle p,b,c\rangle\\
&\phantom{=}\ \ \mathit{where}\\
&\phantom{=}\ \ \langle o,a,b\rangle \ = \ s\\
&\phantom{=}\ \ p = (+)\ o\ b\\
&\phantom{=}\ \ c = (+)\ a\ b
\end{aligned}$$

The two additions $(+)\ a\ b$ and $(+)\ o\ b$ are thus evaluated in parallel. The first tuple ever created by the program contains three numbers, all of which are normal forms: $\langle 0,0,1\rangle$. Because of the strict semantics of *pforce* as required by the master-slave style parallel evaluation, each subsequent tuple will be fully normalized when created. The sharing between computations is maintained as usual. In the example, both additions use the variable $b$, which is obtained from the current tuple and made available to all processors that need the value through the shared heap.

In all four programs that we have used, the only sharing that occurs is between the elements of the state-tuple that is passed as an argument to *nextstate*. Because of the completely strict semantics of *pforce*, all state tuple elements are fully normalized before *nextstate* is entered. Therefore, no locking/blocking mechanism is needed to prevent concurrent reduction of shared expressions. The implementation of master/slave parallelism for communication lifted programs is thus simple and fast. No locking overhead is incurred and scheduling is equally simple and fast. As we will see in the next section it is more complicated to implement pipeline parallelism.

Not all programs that one might wish to transform by communication lifting will have the property that only normal forms are shared. In general, CAF's may be shared between processes so that a locking mechanism is needed. In future work we will identify extra conditions on synchronous process networks to guarantee that only normal forms are shared after communication lifting.

The column *master-slave/trans.* in Table 1 shows the parallel performance of the four test programs after communication lifting on the 4-processor system. The performance of three programs is improved by parallel execution, that of the `fibsum` program is not affected.

The granularity of the `fibsum` program is only one addition, just about sufficient to compensate the overhead of process creation by *pforce*. But the `flipflop` program, still quite fine grained, is already faster then the sequential version. This shows that the overhead of *pforce* is indeed low.

The `fft` program has four coarse grain and four fine grain processes (the tuple size is 8). However, there is a substantial amount of sequential processing before and after these processes are created. About half of the time is spent in the sequential parts of the program. The speedup is about 2.2.

The wave4 program has three processes (the tuple size is 3), two of which contain a significant amount of work. This explains the speedup of about 1.9.

## 7.3 *Parallel performance of a pipeline implementation*

The performance of the master-slave parallel system will now be compared with a pipeline parallel system. The best way to do this is by using two implementations that are similar in as many ways as possible, so that the differences can be attributed to the differences between master-slave and pipeline parallelism.

The master-slave parallel system provides most of the mechanism required to implement pipelines on the shared memory system. The process networks that must be executed in a pipeline parallel fashion may be annotated as shown below:

$$\begin{aligned} pforce\ \langle take\ x\ \tilde{o}, take\ x\ \tilde{a}\rangle\ where\ \tilde{o} &= 0 : map\_2\ (+)\ \tilde{o}\ \tilde{b} \\ \tilde{a} &= 0 : \tilde{b} \\ \tilde{b} &= 1 : map\_2\ (+)\ \tilde{a}\ \tilde{b} \end{aligned}$$

The *pforce* primitive takes the same steps as before, which in this case means that both the expressions *take* $x$ $\tilde{o}$ and *take* $x$ $\tilde{a}$ will be fully evaluated in parallel. The variable $x$ gives the number of elements of each stream that we wish to be evaluated, which should be 3000 in the case of the Fibonacci-sum example. See the column *stream length* in Table 1 for the values required by the other network programs.

The *pforce* expression above behaves as a pipeline that computes the streams $\tilde{a}$ and $\tilde{o}$ in parallel. Figure 7 shows the slightly simplified configurations of the graphs that arise during the first two graph reduction steps taken by the two processors. The processors are notionally separated by the dotted line, there is no physical separation as the processors use a shared heap. The boxes represent suspended computations, all other nodes represent data. Applications of *hd* and *tl* (see the definitions of *map* and *take* in Figure 4) have been omitted to avoid clutter.

Initially, both Processor 1 and Processor 2 are in a state whereby the next action will be to evaluate the suspended computations *map_2* (+) .... Processor 1 requests input from Processor 2. This is indicated by the pointer that crosses the dotted line, and which points at the shared object 1. Both processors will be able to use this shared object without synchronisation because it is data. After some time, both processors will have progressed to the state shown as Step 2 in Figure 7. At this stage, both processors will start to evaluate the suspended applications of +. It is now apparent that for the pipe-line parallel system to work properly, a locking/blocking mechanism is required.

To support pipelines, a low overhead locking/blocking mechanism has been built into the runtime system to avoid two or more processors from reducing the same sub-graph. The locking mechanism uses the XMEM machine instruction to read a memory location and to replace its contents immediately with a known lock value. The hardware implements this as an atomic transaction. Should the lock thus

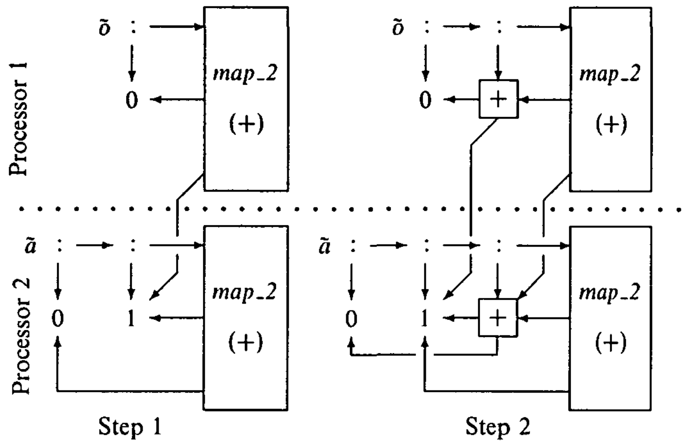


Fig. 7. Two processors performing pipe-line parallel graph reduction in a shared memory system. Processor 1 computes the Fibonacci-sum stream $\tilde{o}$ and Processor 2 computes the Fibonacci stream $\tilde{a}$. The boxes represent suspended computations, all other nodes represent data.

accessed be unavailable, then the processor requesting the lock will block, which is implemented as a busy-wait until the locked object becomes available.

Mapping the two parallel pipeline processes onto a system with two processors is straight forward, but as the number of processes exceeds the number of processors, a suitable static mapping of pipeline processes onto processors is hard to find. We have tried several static mappings, using both fine grained and course grained parallel execution. The best results are shown in the column marked *pipe*/*orig*. of Table 1. We do not expect a dynamically scheduled pipe-line to be much better, because measurements have shown that the statically scheduled pipe-line spends at most 8% of its time busy waiting. A dynamically scheduled pipeline requires some execution time of its own, so it may be at most 8% better than a statically scheduled pipeline.

Because a low overhead locking mechanism has been combined with the best possible static process schedule, the figures in Table 1 represent the best speed up for a statically scheduled pipeline implementation. The master-slave implementation is consistently faster than the pipeline. The parts of the programs that run sequentially on the master slave system offer some additional parallelism for the pipeline system. This effect is particularly strong on the `fft` program. The overhead of the required locking mechanism is apparently not compensated by the extra parallelism available to the pipeline.

It is possible to conceive a pipeline mechanism that does not require locks (Kelly, 1989). Such a mechanism will not be able to exploit more parallelism than the master-slave implementation combined with communication-lifting. The advantage of a real pipeline is lost. Moreover, we expect that such an implementation will result in more overhead than our implementation, because a complex synchronization mechanism is required, that exchanges normalized stream elements between the pipe processes. Communication lifting essentially extracts the synchronization mechanism at compile time, resulting in the iteration of the function *nextstate*. This function contains the knowledge of the communication pattern, while iterate provides the synchronisation in the most efficient way.

## 8 Related work

Many languages have been developed to support programming with process networks, which gives an indication of the importance of work in this area. Lucid (Ashcroft and Wadge, 1977) is one of the first languages based on the notion that variables represent not a single value, but a potentially infinite history of values. This notion is also central to the work on Esterel (Berry and Cosserat, 1984), Signal (Gautier *et al.*, 1987), Lustre (Caspi *et al.*, 1987) and others. Languages such as these offer special operators to manipulate the histories, whereby the aim has often been to make programs look like more conventional programs by hiding the history character of the variables involved.

The language that bears most resemblance to our work is Lustre (Caspi *et al.*, 1987). The differences lie in the realisation: the Lustre implementation is conventional in the sense that it uses a special purpose compiler. Our synchronous process network language is a true subset of a standard lazy functional language, and thus requires no special purpose compiler. However, to achieve a good sequential performance for the synchronous networks embedded in a lazy functional program we use program transformation techniques. Program annotations are used to achieve speedup through parallel evaluation of components of the process networks.

The core of Lustre (its data and sequence operators) is equivalent to the synchronous process network language, as defined in Figure 3. To illustrate this point consider the implementation of the Lustre data operators + and its four sequence operators *pre*, $\rightarrow$, *when* and *current* using only the four stream functions (:), *tl*, *map_n* and *iterate_n* as defined in Figure 4.

Each Lustre sequence represents a stream of values, with which a clock is associated. A clock can be thought of as a stream of boolean values. The *zip* of the stream of values and the associated clock is thus a sensible representation of a Lustre sequence:

$$\begin{aligned} clock\ \tilde{x} &= map\_1\ \ c\ \tilde{x} \\ &\quad where \\ &\quad c\ x\ =\ \langle True, x\rangle \end{aligned}$$

The Lustre sequence operators *pre* and $\rightarrow$ correspond to (:) and *tl*, respectively:

$$\begin{aligned} pre\ \tilde{x} &= \langle True, \bot\rangle\ :\ \tilde{x} \\ \tilde{x} \rightarrow \tilde{y} &= hd\ \tilde{x}\ :\ tl\ \tilde{y} \end{aligned}$$

A Lustre data operator applies some function to the elements of sequences. The + operator for example performs pairwise addition of the elements of two input sequences. The Lustre semantics specify that the two sequences must be *on the same* *clock*, so that additions will only take place when the clocks of both input streams are *True* (see the first clause of the function $p$ below). When both clocks are *False*, an undefined value $\langle False, \bot \rangle$ appears in the output stream (second clause of $p$). Should the sequences be on different clocks, a semantic error is produced (last clause of $p$):

$$
\begin{aligned}
\tilde{x} + \tilde{y} = {} & map\_2 \;\; p \;\; \tilde{x} \;\; \tilde{y} \\
& where \\
& p \;\; \langle True, x \rangle \;\; \langle True, y \rangle \;\; = \langle True, x + y \rangle \\
& p \;\; \langle False, x \rangle \;\; \langle False, y \rangle \;\; = \langle False, \bot \rangle \\
& p \;\; x \qquad\qquad y \qquad\qquad = \bot
\end{aligned}
$$

All other Lustre data operators can be expressed in a similar way using *map_n* and an appropriate auxiliary function. The Lustre sampling operator *when* can also be implemented using *map_2*:

$$
\begin{aligned}
\tilde{e} \;\; when \;\; \tilde{b} = {} & map\_2 \;\; w \;\; \tilde{e} \;\; \tilde{b} \\
& where \\
& w \;\; \langle True, e \rangle \;\; \langle True, True \rangle \;\; = \langle True, e \rangle \\
& w \;\; e \qquad\qquad b \qquad\qquad\quad = \langle False, \bot \rangle
\end{aligned}
$$

The implementation of the Lustre projection operator *current* requires the use of *tl* and *iterate_1*. The state maintained by *iterate_1* is used to remember the last stream element, so that this can be inserted in place of stream elements with a *False* clock value. The *tl* is necessary to remove the first, irrelevant stream element that is produced by *iterate_1*:

$$
\begin{aligned}
current \;\; \tilde{y} = {} & tl \;\; (iterate\_1 \;\; c \;\; \langle True, \bot \rangle \;\; \tilde{y}) \\
& where \\
& c \;\; s \;\; \langle True, e \rangle \;\; = \langle True, e \rangle \\
& c \;\; s \;\; \langle False, e \rangle \;\; = s
\end{aligned}
$$

The Lustre operators can thus all be implemented using the four stream functions (:), *tl*, *map_n* and *iterate_n* as defined in Figure 4. The functional forms of Lustre can all be implemented in Miranda without difficulty.

When viewed as a high level optimization technique, communication lifting is related to deforestation (Wadler, 1988; Gill *et al.*, 1993). Restricted to lists, deforestation removes the need for intermediate list structure. An expression such as $map\_1 \;\; f \;\; (map\_1 \;\; g \;\; \tilde{x})$ is transformed into $map\_1 \;\; (f.g) \;\; \tilde{x}$. For deforestation it is essential, that the producer (here $map\_1 \;\; g$) and the consumer (here $map\_1 \;\; f$) of an intermediate list can be identified. Communication lifting operates on any set of lists and does not require such lists to be in a producer-consumer relationship. Instead communication lifting requires the lists to be manipulated in a synchronous fashion. Communication lifting is thus supplementary to deforestation.

## 9 Conclusions

Many languages have been developed to support programming with synchronous streams (Esterel, Signal, Lustre, etc.). This indicates how important programming with synchronous networks is as a technique for developing practical applications.

The disadvantage of developing a new language is that it also requires a new implementation to be built. The approach we take is to build process networks using a subset of a standard lazy functional language. As an example we show that this subset is equivalent to the special purpose stream-language Lustre.

When a large number of streams is involved, which will be often the case in practical applications, the cost of lazy evaluation is high. Therefore we developed a program transformation called *communication lifting*, which takes a synchronous process network consisting of $n$ streams into a network with only a single stream carrying $n$-tuples.

Measurements show that for three out of four test programs, managing the single stream of $n$-tuples is cheaper than managing the original $n$ streams. The performance of the fourth program stays within 5% of the untransformed version. This result indicates that the use of communication lifting in sequential applications is worthwhile.

We have also investigated the possibility to evaluate the components of the $n$-tuples, resulting from communication lifting, in parallel. This gives rise to a simple master/slave kind of parallelism, which we have implemented on a four processor shared memory machine.

Comparing this master/slave implementation to the conventional way of parallelising process networks, in a pipe-line fashion, shows that communication lifting outperforms a pipe-line implementation which uses an optimal static schedule.

The communication lifting transformation has been specified formally. This makes it possible to prove the correctness of the transformation (see appendix) and to implement communication lifting as an automatic tool. Annotation by the programmer is necessary to indicate which set of streams must be transformed.

## Acknowledgements

We thank Marcel Beemster, Andy Gravell, Koen Langendoen, Henk Muller and the two referees for their comments on a draft version of the paper. Andy Gravell suggested the derivation of the communication lifting transformation in Section 2.

## Appendix - Correctness of the transformation

To establish the correctness of the Rules *T0* to *T5*, a few auxiliary lemmas are used, which are shown below. The zip-, and map-lemmas can easily be proved by complete induction on $i$. The proofs of the remaining lemmas are given. It is essential for all these proofs that streams are infinite lists; the stream elements may assume any value, including $\perp$.

## *The zip-lemma*

Given $\forall n \in \mathbb{N} \wedge \tilde{x}_1, \ldots, \tilde{x}_n \in \tilde{D}$, then:

$$\forall i \in \mathbb{N} : (zip\_n \ \langle \tilde{x}_1, \ldots, \tilde{x}_n \rangle)!i \ = \ \langle (\tilde{x}_1!i), \ldots, (\tilde{x}_n!i) \rangle$$

This can be proved by induction on *i*.

## *The map-lemma*

Given $\forall \tilde{x}_1 \ldots \tilde{x}_n \in \tilde{D} \wedge f \in F$, then:

$$\forall i \in \mathbb{N} : (map\_n \ f \ \tilde{x}_1 \ldots \tilde{x}_n)!i \ = \ f \ (\tilde{x}_1!i) \ \ldots \ (\tilde{x}_n!i)$$

This can be proved by induction on *i*.

## *The iterate-lemma*

Given $\forall \tilde{x}_1 \ldots \tilde{x}_n \in \tilde{D} \wedge f \in F \wedge s \in D$, then:

$$\begin{aligned} \forall i \in \mathbb{N} : (iterate\_n \ f \ s \ \tilde{x}_1 \ldots \tilde{x}_n)!i \ = \ & \tilde{y}!i \\ & where \\ & \tilde{y} \ = \ s : map\_m \ f \ \tilde{y} \ \tilde{x}_1 \ldots \tilde{x}_n \\ & m \ = \ n+1 \end{aligned}$$

To prove this lemma, consider the special case that $n = 1$. Then given $\tilde{x} \in \tilde{D}$ and the binary function $f \in F$, the iterate-lemma is:

$$\begin{aligned} \forall i \in \mathbb{N} : (iterate\_1 \ f \ s \ \tilde{x})!i \ = \ & \tilde{y}!i \\ & where \\ & \tilde{y} \ = \ s : map\_2 \ f \ \tilde{y} \ \tilde{x} \end{aligned}$$

The proof is by induction on *i*. Base case:

$$\begin{aligned} & (iterate\_1 \ f \ s \ \tilde{x})!0 \\ = \ & hd \ (iterate\_1 \ f \ s \ \tilde{x}) && \text{(unfold !)} \\ = \ & hd \ (s : iterate\_1 \ f \ (f \ s \ (hd \ \tilde{x})) \ (tl \ \tilde{x})) && \text{(unfold } iterate\_1\text{)} \\ = \ & s && \text{(unfold } hd\text{)} \\ = \ & \tilde{y}!0 && \text{(new definition, fold !)} \\ & where \\ & \tilde{y} \ = \ s : map\_2 \ f \ \tilde{y} \ \tilde{x} \quad \square \end{aligned}$$

Induction step:

$$\begin{array}{ll}
(\mathit{iterate\_1}\ \ f\ \ s\ \ \tilde{x})!(i+1) & \\
= (\mathit{tl}\ \ (\mathit{iterate\_1}\ \ f\ \ s\ \ \tilde{x}))!i & \text{(unfold !)} \\
= (\mathit{tl}\ \ (s : \mathit{iterate\_1}\ \ f\ \ (f\ \ s\ \ (\mathit{hd}\ \ \tilde{x}))\ \ (\mathit{tl}\ \ \tilde{x})))!i & \text{(unfold } \mathit{iterate\_1}\text{)} \\
= (\mathit{iterate\_1}\ \ f\ \ (f\ \ s\ \ (\mathit{hd}\ \ \tilde{x}))\ \ (\mathit{tl}\ \ \tilde{x}))!i & \text{(unfold } \mathit{tl}\text{)} \\
= \tilde{y}!i & \text{(hypothesis)} \\
\quad \mathit{where} & \\
\quad \tilde{y}\ \ =\ \ f\ \ s\ (\mathit{hd}\ \ \tilde{x}) : \mathit{map\_2}\ \ f\ \ \tilde{y}\ \ (\mathit{tl}\ \ \tilde{x}) & \\
= \tilde{y}!i & \text{(fold } \mathit{map\_2}\text{)} \\
\quad \mathit{where} & \\
\quad \tilde{y}\ \ =\ \ \mathit{map\_2}\ \ f\ \ (s : \tilde{y})\ \ (\mathit{hd}\ \ \tilde{x} : \mathit{tl}\ \ \tilde{x}) & \\
= (s : \tilde{y})!(i+1) & \text{(fold !, } \mathit{hd} \text{ and } \mathit{tl}\text{)} \\
\quad \mathit{where} & \\
\quad (s : \tilde{y})\ \ =\ \ s : \mathit{map\_2}\ \ f\ \ (s : \tilde{y})\ \ \tilde{x} & \\
= \tilde{z}!(i+1) & \text{(define } \tilde{z} = s : \tilde{y}\text{)} \\
\quad \mathit{where} & \\
\quad \tilde{z}\ \ =\ \ s : \mathit{map\_2}\ \ f\ \ \tilde{z}\ \ \tilde{x} \quad \square &
\end{array}$$

A similar proof can be given of the general case for $n \geq 2$.

## *Corollary of the iterate-lemma*

Given $\forall \tilde{x}_1 \ldots \tilde{x}_n \in \tilde{D} \wedge f \in F \wedge s \in D \wedge m = n + 1$, then:

$$\begin{array}{ll}
\qquad\qquad \tilde{y} = s : \mathit{map\_m}\ \ f\ \ \tilde{y}\ \ \tilde{x}_1 \ldots \tilde{x}_n & \\
\equiv & \text{(iterate-lemma)} \\
\forall i \in \mathbb{N} : \tilde{y}!i = (\mathit{iterate\_n}\ \ f\ \ s\ \ \tilde{x}_1 \ldots \tilde{x}_n)!i & \\
\equiv & \text{(element-lemma)} \\
\qquad\qquad \tilde{y} = \mathit{iterate\_n}\ \ f\ \ s\ \ \tilde{x}_1 \ldots \tilde{x}_n &
\end{array}$$

## *The element-lemma*

Given $\forall \tilde{x}, \tilde{y} \in \tilde{D}$ then:

$$\tilde{x} = \tilde{y} \equiv \forall i \in \mathbb{N} : \tilde{x}!i = \tilde{y}!i$$

To prove the element-lemma an auxiliary result is needed to give the correspondence between list comprehensions and the function *take*, so that the take-lemma (Bird and Wadler, 1988, pp. 182–183) can be used. Given $\tilde{x} \in \tilde{D}$, then:

$$\forall i \in \mathbb{N} : \mathit{take}\ \ i\ \ \tilde{x}\ \ =\ \ [\tilde{x}!k \mid k \leftarrow [0..i-1]]$$

The proof of the auxiliary lemma is by induction on *i*. Base case:

$$\begin{aligned}
&take\ 0\ \tilde{x} \\
&= [] && \text{(unfold } take\text{)} \\
&= [\tilde{x}!k \mid k \leftarrow []] && \text{(list comprehension)} \\
&= [\tilde{x}!k \mid k \leftarrow [0..-1]] \quad \square && \text{(arithmetic)}
\end{aligned}$$

Induction step:

$$\begin{aligned}
&take\ (i+1)\ \tilde{x} \\
&= hd\ \tilde{x} : take\ i\ (tl\ \tilde{x}) && \text{(unfold } take\text{)} \\
&= hd\ \tilde{x} : [(tl\ \tilde{x})!k \mid k \leftarrow [0..i-1]] && \text{(hypothesis)} \\
&= hd\ \tilde{x} : [(tl\ \tilde{x})!(h-1) \mid h \leftarrow [1..i]] && (k = h-1) \\
&= \tilde{x}!0 : [\tilde{x}!h \mid h \leftarrow [1..i]] && \text{(fold !)} \\
&= [\tilde{x}!h \mid h \leftarrow [0..i]] \quad \square && \text{(list comprehension)}
\end{aligned}$$

The unfold step is permitted because streams are infinite lists and not partial lists. The element-lemma can now be proved as follows:

$$\begin{aligned}
& && \tilde{x} = \tilde{y} \\
&\equiv && && \text{(take-lemma)} \\
&\forall i \in \mathbb{N} : && take\ i\ \tilde{x} = take\ i\ \tilde{y} \\
&\equiv && && \text{(auxiliary lemma)} \\
&\forall i \in \mathbb{N} : && [\tilde{x}!k \mid k \leftarrow [0..i-1]] = [\tilde{y}!k \mid k \leftarrow [0..i-1]] \\
&\equiv && && \text{(list comprehension)} \\
&\forall i \in \mathbb{N} : && \tilde{x}!i = \tilde{y}!i \quad \square
\end{aligned}$$

***The stream-lemma***

When given $\tilde{z}_1 \ldots \tilde{z}_h \in \tilde{D} \wedge z_1 \ldots z_h \in D \wedge A \in F \wedge \tilde{A} \in \tilde{F}$ and the following 4 conditions are met:

(i) $\tilde{A}\ \tilde{z}_1 \ldots \tilde{z}_h = \tilde{e}_0[\tilde{z}_1 \ldots \tilde{z}_h, \tilde{a}_1 \ldots \tilde{a}_m]$
*where*
$$\begin{aligned}
\tilde{a}_1 &= \tilde{e}_1[\tilde{z}_1 \ldots \tilde{z}_h, \tilde{a}_1 \ldots \tilde{a}_m] \\
&\vdots \\
\tilde{a}_m &= \tilde{e}_m[\tilde{z}_1 \ldots \tilde{z}_h, \tilde{a}_1 \ldots \tilde{a}_m]
\end{aligned}$$

(ii) $A\ z_1 \ldots z_h = e_0[z_1 \ldots z_h, a_1 \ldots a_m]$
*where*
$$\begin{aligned}
a_1 &= e_1[z_1 \ldots z_h, a_1 \ldots a_m] \\
&\vdots \\
a_m &= e_m[z_1 \ldots z_h, a_1 \ldots a_m]
\end{aligned}$$

(iii) there are no free occurrences of either $\tilde{A}$ or $A$ in $\tilde{e}_0 \ldots \tilde{e}_m$ or $e_0 \ldots e_m$

(iv) $\forall i \in \mathbb{N} \wedge 0 \le j \le m$ we have:
$$(\tilde{e}_j[\tilde{z}_1 \ldots \tilde{z}_h, \tilde{a}_1 \ldots \tilde{a}_m])!i = e_j[\tilde{z}_1!i/z_1 \ldots \tilde{z}_h!i/z_h, \tilde{a}_1!i/a_1 \ldots \tilde{a}_m!i/a_m]$$

Then the stream-lemma asserts that:

(v) $\tilde{A} \;=\; map\_h\;A$

In the stream-lemma and its proof some special notation will be used. For an expression $e$, in which the variables $a_1 \ldots a_m$ occur free we write $e[a_1 \ldots a_m]$ and the notation $e[b_1/a_1 \ldots b_m/a_m]$ is an expression $e$ in which the free variables $a_1 \ldots a_m$ are simultaneously replaced by respectively $b_1 \ldots b_m$. For brevity we use a superscript in the proof to denote projection on tuples rather than the function $sel\_m$:

$$\forall\; 1 \le k \le m : \; \langle \tilde{a}_1, \ldots, \tilde{a}_m \rangle^k \;=\; \tilde{a}_k$$

We begin the proof by associating a function $\tilde{f}_j$ with each of the expressions $\tilde{e}_j[\tilde{z}_1 \ldots \tilde{z}_h, \tilde{a}_1 \ldots \tilde{a}_m]$ in (i), such that:

$$\forall\; 0 \le j \le m : \tilde{f}_j = \lambda ut.\tilde{e}_j[u^1/\tilde{z}_1 \ldots u^h/\tilde{z}_h, t^1/\tilde{a}_1 \ldots t^m/\tilde{a}_m]$$

Thus $\tilde{f}_j\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle \;=\; \tilde{e}_j$. In the same way associate functions $f_0 \ldots f_m$ with the expressions $e_0 \ldots e_m$. As the second step rewrite (iv) in terms of the functions $\tilde{f}_0 \ldots \tilde{f}_m$ and $f_0 \ldots f_m$.

$\forall i \in \mathbb{N} \wedge 0 \le j \le m$ :
$(\tilde{e}_j[\tilde{z}_1 \ldots \tilde{z}_h, \tilde{a}_1 \ldots \tilde{a}_m])!i \;=\; e_j[\tilde{z}_1!i/z_1 \ldots \tilde{z}_h!i/z_h, \tilde{a}_1!i/a_1 \ldots \tilde{a}_m!i/a_m]$
$\equiv$ (use the definitions of $f_0 \ldots f_m$ and $\tilde{f}_0 \ldots \tilde{f}_m$)
$(\tilde{f}_j\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle)!i \;=\; f_j\;\langle \tilde{z}_1!i, \ldots, \tilde{z}_h!i\rangle\;\langle \tilde{a}_1!i, \ldots, \tilde{a}_m!i\rangle$
$\equiv$ (substitute $z = \langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle$ and $a = \langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle$ and use the zip-lemma)
(vi) $(\tilde{f}_j\;z\;a)!i \;=\; f_j\;((zip\_h\;z)!i)\;((zip\_m\;a)!i)$

The third step is to reformulate the definition of $\tilde{A}$ from given (i):

(vii) $\tilde{A}\;\tilde{z}_1 \ldots \tilde{z}_h$
$= \tilde{f}_0\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle$ (given (i))
  *where*
  $\tilde{a}_1 \;= \tilde{f}_1\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle$
  $\vdots$
  $\tilde{a}_m \;= \tilde{f}_m\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle$
$= \tilde{f}_0\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle$ (tupling)
  *where*
  $\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle \;=\; \langle \tilde{f}_1\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle, \ldots, \tilde{f}_m\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;\langle \tilde{a}_1, \ldots, \tilde{a}_m\rangle\rangle$
$=$ (abstraction and fixed point)
  $\tilde{f}_0\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;(fix(\lambda a.\langle \tilde{f}_1\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;a, \ldots, \tilde{f}_m\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle\;a\rangle))$
$= \tilde{B}\;\langle \tilde{z}_1, \ldots, \tilde{z}_h\rangle$ (introduce $\tilde{B}$, $\tilde{E}$)
  *where*
  $\tilde{B} \;= \lambda z.\tilde{f}_0\;z\;\tilde{E}[z]$
  $\tilde{E}[z] = fix(\lambda a.\langle \tilde{f}_1\;z\;a, \ldots, \tilde{f}_m\;z\;a\rangle)$

Rewrite definition (ii) of $A$ in a similar way to yield:

(viii) $A\ z_1 \ldots z_h = B\ \langle z_1, \ldots, z_h \rangle$
$$\begin{aligned}&\textit{where}\\ &B \quad = \lambda z. f_0\ z\ E[z]\\ &E[z] = \mathit{fix}(\lambda a. \langle f_1\ z\ a, \ldots, f_m\ z\ a \rangle)\end{aligned}$$

Omitting the *where* expression of $B$ (viii) looks like this:

$A\ z_1 \ldots z_h \quad = B\ \langle z_1, \ldots, z_h \rangle$

$\equiv$ (Let $\tilde{z}_1 \ldots \tilde{z}_h \in \tilde{D}$ and use the element-lemma)

$\forall i \in \mathbb{N}$ :

$A\ (\tilde{z}_1 ! i) \ldots (\tilde{z}_h ! i) \quad = B\ \langle \tilde{z}_1 ! i, \ldots, \tilde{z}_h ! i \rangle$

$\equiv$ (zip-lemma)

$\forall i \in \mathbb{N}$ :

$A\ (\tilde{z}_1 ! i) \ldots (\tilde{z}_h ! i) \quad = B\ ((\mathit{zip\_h}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle) ! i)$

$\equiv$ (map- and element-lemma)

$\mathit{map\_h}\ A\ \tilde{z}_1 \ldots \tilde{z}_h = \mathit{map\_1}\ B\ (\mathit{zip\_h}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle)$

Compare this to (vii), which relates $\tilde{A}$ and $\tilde{B}$:

$\tilde{A}\ \tilde{z}_1 \ldots \tilde{z}_h \quad = \tilde{B}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle$

To prove the stream-lemma (v) it thus remains to show that:

$\tilde{B}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle \quad = \mathit{map\_1}\ B\ (\mathit{zip\_h}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle)$

$\equiv$ (element-lemma)

$\forall i \in \mathbb{N}$ :

$(\tilde{B}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle) ! i = (\mathit{map\_1}\ B\ (\mathit{zip\_h}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle)) ! i$

$\equiv$ (map-lemma)

$\forall i \in \mathbb{N}$ :

$(\tilde{B}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle) ! i = B\ ((\mathit{zip\_h}\ \langle \tilde{z}_1, \ldots, \tilde{z}_h \rangle) ! i)$

Using the definitions of $\tilde{B}$ and $B$ we can derive the following fixed point equation for $(\mathit{zip\_m}\ \tilde{E}[z]) ! i$:

$\forall i \in \mathbb{N}$ :

$(\mathit{zip\_m}\ \tilde{E}[z]) ! i$

$= (\mathit{zip\_m}\ (\mathit{fix}(\lambda a. \langle \tilde{f}_1\ z\ a, \ldots, \tilde{f}_m\ z\ a \rangle))) ! i$ (unfold $\tilde{E}$)

$= (\mathit{zip\_m}\ \langle \tilde{f}_1\ z\ \tilde{E}[z], \ldots, \tilde{f}_m\ z\ \tilde{E}[z] \rangle) ! i$ (unfold *fix*)

$= \langle (\tilde{f}_1\ z\ \tilde{E}[z]) ! i, \ldots, (\tilde{f}_m\ z\ \tilde{E}[z]) ! i \rangle$ (zip-lemma)

$=$ (by (vi))

$\langle f_1\ ((\mathit{zip\_h}\ z) ! i)\ ((\mathit{zip\_m}\ \tilde{E}[z]) ! i), \ldots, f_m\ ((\mathit{zip\_h}\ z) ! i)\ ((\mathit{zip\_m}\ \tilde{E}[z]) ! i) \rangle$

$=$ (abstraction)

$(\lambda a. \langle f_1\ ((\mathit{zip\_h}\ z) ! i)\ a, \ldots, f_m\ ((\mathit{zip\_h}\ z) ! i)\ a \rangle)\ ((\mathit{zip\_m}\ \tilde{E}[z]) ! i)$

$= \mathit{fix}(\lambda a. \langle f_1\ ((\mathit{zip\_h}\ z) ! i)\ a, \ldots, f_m\ ((\mathit{zip\_h}\ z) ! i)\ a \rangle)$ (fixed point)

$= E[(\mathit{zip\_h}\ z) ! i / z]$ (fold $E$)

So for all $h$-tuples $z$ we also have:

$$(ix)\ \forall i \in \mathbb{N} : (zip\_m\ \tilde{E}[z])!i \ = \ E[(zip\_h\ z)!i/z]$$

The proof of the stream-lemma can now be completed as follows:

$$\begin{array}{ll}
(\tilde{B}\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle)!i & \\
= ((\lambda z.\tilde{f}_0\ z\ \tilde{E}[z])\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle)!i & \text{(unfold } \tilde{B}) \\
= (\tilde{f}_0\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle\ \tilde{E}[\langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle])!i & \text{(reduction)} \\
= f_0\ ((zip\_h\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle)!i)\ ((zip\_m\ \tilde{E}[\langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle])!i) & \text{(by (vi))} \\
= f_0\ ((zip\_h\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle)!i)\ E[(zip\_h\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle)!i/z] & \text{(by (ix))} \\
= (\lambda z.f_0\ z\ E[z])\ ((zip\_h\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle)!i) & \text{(abstraction)} \\
= B\ ((zip\_h\ \langle \tilde{z}_1,\ldots,\tilde{z}_h\rangle)!i) \quad \square & \text{(fold } B)
\end{array}$$

## *Correctness of the simplifying transformations*

The correctness of each of the Rules *T1*, *T3* and *T5* will now be proved. The correctness of the composition of these transformations then follows, because each represents a total function ($p \rightarrow p$). The partial correctness of the clauses in Figure 6 will be proved first. Then the termination of the transformations will be established.

### *Partial correctness of the simplifying transformations*

Using the auxiliary lemmas, the correctness of Clause 4a can be proved as follows. Define $\forall n \in \mathbb{N} \wedge f \in F \wedge s \in D \wedge x_1 \ldots x_n \in \tilde{D}$:

$$\begin{array}{ll}
\tilde{y} & = s : map\_m\ f\ \tilde{y}\ \tilde{x}_1\ \ldots\ \tilde{x}_n \\
m & = n + 1
\end{array}$$

Then:

$$\begin{array}{ll}
\quad \tilde{v} = iterate\_n\ f\ s\ \tilde{x}_1\ \ldots\ \tilde{x}_n & \\
\equiv & \text{(element-lemma)} \\
\forall i \in \mathbb{N} : \tilde{v}!i = (iterate\_n\ f\ s\ \tilde{x}_1\ \ldots\ \tilde{x}_n)!i & \\
\equiv & \text{(iterate-lemma)} \\
\forall i \in \mathbb{N} : \tilde{v}!i = f\ (\tilde{y}!i)\ (\tilde{x}_1!i)\ \ldots\ (\tilde{x}_n!i) & \\
\equiv & \text{(map-lemma)} \\
\forall i \in \mathbb{N} : \tilde{v}!i = (map\_m\ f\ \tilde{y}\ \tilde{x}_1\ \ldots\ \tilde{x}_n)!i & \\
\equiv & \text{(element-lemma)} \\
\quad \tilde{v} = map\_m\ f\ \tilde{y}\ \tilde{x}_1\ \ldots\ \tilde{x}_n &
\end{array}$$

The fact that for all $n, f, s, \tilde{x}_1 \ldots \tilde{x}_n$ the stream $\tilde{v}$ prior to transformation is equal to that after transformation establishes the correctness of Clause 4a.

The proofs for the remaining clauses present no difficulties. The Clauses 1, 2b, 3, 4b and 5e by themselves make no changes. Clause 2a introduces a new equation, which is equivalence preserving. An unfold in recursive equations such as those under consideration here is always equivalence preserving (Manna *et al.*, 1973). Rule *T5* uses unfolds and introduces new definitions: Clause 5a unfolds the definition of *map_n* as given in Figure 4 and introduces new equations; Clauses 5b and 5c unfold the definitions of $\tilde{w}$ and *tl*. For Clause 5b this gives the equation $\tilde{o} = \tilde{x}$ and for (5c) we have $\tilde{v} = \tilde{x}$. These equations can be eliminated by renaming $\tilde{x}$ to $\tilde{o}$ and $\tilde{v}$ to $\tilde{x}$ respectively. Clause 5d unfolds the definitions of *map_n*, $\tilde{w}$ and *tl* and introduces new equations. Rule *T5* is thus equivalence preserving, which concludes the proofs of the partial correctness of the simplifying transformations.

## *Termination of the simplifying transformations*

All simplifying transformations terminate because a bound can be given for the number of times each individual clause is applied.

Clauses 1, 3 and 5e are each applied once. The number of function applications in the original process network is an upper bound for number of times Clauses 2a, 2b, 4a or 4b is applied. Clause 5a will be applied at most once. The only clause of Rule *T5* that matches an equation of the form $\tilde{o} = \ldots : \ldots$ is (5c), which will never replace that equation by one of the form that can be matched again by (5a).

To derive an upper bound on the number of times Clauses 5b, 5c or 5d may be called, let the number of applications of *tl* and *map_n* be *t* respectively *m*. The way the labelling of the equations is created and maintained guarantees that Clause 5d will be applied at most $m \times t$ times. The upper bound on the number of times Clauses 5b or 5c are applied is given by the number of *tl* applications, which are either present originally, or introduced by Clause 5a or 5d. This number is bounded because there is an upper bound on the number of times (5d) is applied.

Summarizing, we have now established the fact that the simplifying transformations preserve equivalence and terminate.

## ***Correctness of the communication lifting transformation***

The termination proof of Rule *T0* (Figure 5) is immediate. To prove the partial correctness of Rule *T0*, the (:) and *map_n* equations on the left-hand side of Rule *T0* are tupled; the *tl* equations are left unchanged. The guard in Clause 0a ensures that the *tl* equations are independent of the remaining equations. Therefore, it is correct to consider the network without the *tl* equations. From now on let $\tilde{v}_1 = \tilde{o}$. Then:

$$\left\{\begin{array}{lcl} \tilde{v}_1 & = & s_1 : \tilde{x}_1 \\ & \vdots & \\ \tilde{v}_k & = & s_k : \tilde{x}_k \\ \tilde{w}_1 & = & map\_n_1\ f_1\ \tilde{y}_{11} \ldots \tilde{y}_{1n_1} \\ & \vdots & \\ \tilde{w}_m & = & map\_n_m\ f_m\ \tilde{y}_{m1} \ldots \tilde{y}_{mn_m} \end{array}\right\} \equiv \begin{array}{l} \langle \tilde{v}_1, \ldots, \tilde{v}_k \rangle \\ = \langle s_1 : \tilde{x}_1, \ldots, s_k : \tilde{x}_k \rangle \\ \textit{where} \\ \quad \tilde{w}_1 = map\_n_1\ f_1\ \tilde{y}_{11} \ldots \tilde{y}_{1n_1} \\ \qquad \vdots \\ \quad \tilde{w}_m = map\_n_m\ f_m\ \tilde{y}_{m1} \ldots \tilde{y}_{mn_m} \end{array}$$

The function composition *unzip_k* ∘ *zip_k* is the identity function on a tuple of streams, thus the left hand-side of Clause 0a is equivalent to:

$$\begin{aligned}
&\langle \tilde{v}_1,\ldots,\tilde{v}_k\rangle = unzip\_k\ (zip\_k\ \langle s_1 : \tilde{x}_1,\ldots,s_k : \tilde{x}_k\rangle)\dagger \\
&\equiv \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(unfold } zip\_k\text{)} \\
&\langle \tilde{v}_1,\ldots,\tilde{v}_k\rangle = unzip\_k\ (\langle s_1,\ldots,s_k\rangle : (zip\_k\ \langle \tilde{x}_1,\ldots,\tilde{x}_k\rangle))\dagger
\end{aligned}$$

Next a new equation $\tilde{s}$ for the stream of $k$-tuples is introduced, and the free variables of the network $\tilde{u}_1\ldots\tilde{u}_h$ are made explicit. This yields the following set of equations, again equivalent to the left-hand side of Clause 0a:

$$\begin{aligned}
&\langle \tilde{v}_1,\ldots,\tilde{v}_k\rangle && = unzip\_k\ \tilde{s} \\
&\tilde{s} && = \langle s_1,\ldots,s_k\rangle : (zip\_k\ \langle \tilde{x}_1,\ldots,\tilde{x}_k\rangle)\dagger \\
&\equiv && \qquad\qquad\qquad\qquad\qquad \text{(make } unzip\_k \text{ equation local)} \\
&\tilde{s} && = \langle s_1,\ldots,s_k\rangle : (zip\_k\ \langle \tilde{x}_1,\ldots,\tilde{x}_k\rangle)\dagger \\
& && \quad where \\
& && \quad \langle \tilde{v}_1,\ldots,\tilde{v}_k\rangle = unzip\_k\ \tilde{s} \\
&\equiv && \qquad\qquad \text{(introduce function } \tilde{A} \text{ and make free variables explicit)} \\
&\tilde{s} && = \langle s_1,\ldots,s_k\rangle : (\tilde{A}\ \tilde{s}\ \tilde{u}_1\ldots\tilde{u}_h) \\
&\tilde{A}\ \tilde{s}\ \tilde{u}_1\ldots\tilde{u}_h && = zip\_k\ \langle \tilde{x}_1,\ldots,\tilde{x}_k\rangle\dagger \\
& && \quad where \\
& && \quad \langle \tilde{v}_1,\ldots,\tilde{v}_k\rangle = unzip\_k\ \tilde{s}
\end{aligned}$$

In the last step of this derivation, we have used the fact that the states $s_1\ldots s_k$ are independent of the stream variables $\tilde{v}_1\ldots\tilde{v}_k$ and $\tilde{w}_1\ldots\tilde{w}_m$ (see Figure 5).

As shown in Figure 8, the final step brings the system of equations in a form that fits the structural requirements of the stream-lemma. The function $\tilde{A}$ as derived from the equations on the left-hand side of Clause 0a is shown to the left, while the corresponding elements of the definitions for the stream-lemma are shown to the right.

The stream-lemma states that $\tilde{A} = map\_h\ \ A$, provided stream-lemma condition (iii) holds $\forall i \in \mathbb{N} \wedge 0 \le j \le k+m$. This is verified as follows:

For $j = 0$ we have that $\tilde{e}_0 \equiv zip\_k\ \langle \tilde{x}_1,\ldots,\tilde{x}_k\rangle$ and also that $e_0 \equiv \langle x_1,\ldots,x_k\rangle$. This can be proved as follows:

$$\begin{aligned}
&\forall i \in \mathbb{N} : \\
&(zip\_k\ \langle \tilde{x}_1,\ldots,\tilde{x}_k\rangle)!i \\
&= \langle \tilde{x}_1!i,\ldots,\tilde{x}_k!i\rangle \qquad\qquad\qquad\qquad \text{(zip-lemma)} \\
&= \langle x_1,\ldots,x_k\rangle[\tilde{x}_1!i/x_1,\ldots,\tilde{x}_k!i/x_k] \quad \square
\end{aligned}$$

† The *where* equations for $w_1\ldots w_m$ have been omitted.

| left- and right-hand side of Clause 0a | stream-lemma functions $\tilde{A}$ and $A$ |
|---|---|
| $\tilde{A}\ \tilde{s}\ \tilde{u}_1\ldots\tilde{u}_h$ | $\tilde{A}\ \tilde{z}_1\ldots\tilde{z}_h$ |
| $= \mathit{zip\_k}\ \langle\tilde{x}_1,\ldots,\tilde{x}_k\rangle$ | $= \tilde{e}_0$ |
| *where* | *where* |
| $\tilde{v}_1 = \mathit{sel\_k}\ 1\ (\mathit{unzip\_k}\ \tilde{s})$ | $\tilde{a}_1 = \tilde{e}_1$ |
| $\vdots$ | $\vdots$ |
| $\tilde{v}_k = \mathit{sel\_k}\ k\ (\mathit{unzip\_k}\ \tilde{s})$ | $\tilde{a}_k = \tilde{e}_k$ |
| $\tilde{w}_1 = \mathit{map\_n_1}\ f_1\ \tilde{y}_{11}\ldots\tilde{y}_{1n_1}$ | $\tilde{a}_{k+1} = \tilde{e}_{k+1}$ |
| $\vdots$ | $\vdots$ |
| $\tilde{w}_m = \mathit{map\_n_m}\ f_m\ \tilde{y}_{m1}\ldots\tilde{y}_{mn_m}$ | $\tilde{a}_{k+m} = \tilde{e}_{k+m}$ |
| $A\ s\ u_1\ldots u_h$ | $A\ z_1\ldots z_h$ |
| $= \langle x_1,\ldots,x_k\rangle$ | $= e_0$ |
| *where* | *where* |
| $v_1 = \mathit{sel\_k}\ 1\ s$ | $a_1 = e_1$ |
| $\vdots$ | $\vdots$ |
| $v_k = \mathit{sel\_k}\ k\ s$ | $a_k = e_k$ |
| $w_1 = f_1\ y_{11}\ldots y_{1n_1}$ | $a_{k+1} = e_{k+1}$ |
| $\vdots$ | $\vdots$ |
| $w_m = f_m\ y_{m1}\ldots y_{mn_m}$ | $a_{k+m} = e_{k+m}$ |

Fig. 8. Structural correspondence between the stream-lemma and *T0*.

For $1 \le j \le k$ we have that $\tilde{e}_j \equiv \mathit{sel\_k}\ j\ (\mathit{unzip\_k}\ \tilde{s})$ and also $e_j \equiv \mathit{sel\_k}\ j\ s$ because:

$\forall i \in \mathbb{N}$ :

$(\mathit{sel\_k}\ j\ (\mathit{unzip\_k}\ \tilde{s}))!i$

$= (\mathit{sel\_k}\ j\ (\mathit{unzip\_k}\ (\mathit{zip\_k}\ \langle s_1 : \tilde{x}_1,\ldots,s_k : \tilde{x}_k\rangle)))!i$ (unfold $\tilde{s}$)

$= (\mathit{sel\_k}\ j\ \langle s_1 : \tilde{x}_1,\ldots,s_k : \tilde{x}_k\rangle)!i$ (identity)

$= (s_j : \tilde{x}_j)!i$ (unfold *sel_k*)

$= (\mathit{sel\_k}\ j\ \langle (s_1 : \tilde{x}_1)!i,\ldots,(s_k : \tilde{x}_k)!i\rangle)$ (fold *sel_k*)

$= \mathit{sel\_k}\ j\ ((\mathit{zip\_k}\ \langle s_1 : \tilde{x}_1,\ldots,s_k : \tilde{x}_k\rangle)!i)$ (zip-lemma)

$= \mathit{sel\_k}\ j\ (\tilde{s}!i)$ (fold $\tilde{s}$)

$= (\mathit{sel\_k}\ j\ s)[\tilde{s}!i/s]$ □

For $k+1 \le k+j \le k+m$ we have $\tilde{e}_{k+j} \equiv \mathit{map\_n_j}\ f_j\ \tilde{y}_{j1}\ldots\tilde{y}_{jn_j}$ and $e_{k+j} \equiv f_j\ y_{j1}\ldots y_{jn_j}$ since:

$\forall i \in \mathbb{N}$ :

$(\mathit{map\_n_j}\ f_j\ \tilde{y}_{j1}\ldots\tilde{y}_{jn_j})!i$

$= f_j\ (\tilde{y}_{j1}!i)\ldots(\tilde{y}_{jn_j}!i)$ (map-lemma)

$= (f_j\ y_{j1}\ldots y_{jn_j})[\tilde{y}_{j1}!i/y_{j1},\ldots,\tilde{y}_{jn_j}!i/y_{jn_j}]$ □

Given the definition of *nextstate* as in Figure 5, (which is the same as $A$ here) we have:

$\tilde{s}\ = \langle s_1,\ldots,s_k\rangle : (\tilde{A}\ \tilde{s}\ \tilde{u}_1\ldots\tilde{u}_h)$

$\equiv$ (stream-lemma)

$\tilde{s}\ = \langle s_1,\ldots,s_k\rangle : (map\_h\ \ A\ \tilde{s}\ \tilde{u}_1\ldots\tilde{u}_h)$

$\equiv$ ($A = nextstate$ and corollary of iterate-lemma)

$\tilde{s}\ = iterate\_h\ \ nextstate\ \langle s_1,\ldots,s_k\rangle\ \tilde{u}_1\ldots\tilde{u}_h$ □

This completes the correctness proof of transformation Rule *T0*.

## References

Allison, L. (1986) *A Practical Introduction to Denotational Semantics.* Cambridge University Press.

Ashcroft, E. A. and Wadge, W. W. (1977) Lucid, a non procedural language with iteration. *Communications of the ACM*, **20**(7):519–526, July.

Berry, G. and Cosserat, L. (1984) The ESTEREL synchronous programming language and its mathematical semantics. In: S. D. Brookes, A. W. Roscoe and G. Winksel, eds., *Seminar on concurrency. Lecture Notes in Computer Science 197*, pp. 389–448. Springer-Verlag.

Bird, R. S. and Wadler, P. L. (1988) *Introduction to Functional Programming.* Prentice Hall.

Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *Journal of the ACM*, **24**(1):44–67, January.

Caspi, P., Pilaud, D., Halbwachs, N. and Plaice, J. A. (1987) LUSTRE: A declarative language for programming synchronous systems. In: *14th Conference on the Principles of Programming Languages*, pp. 178–188. Munich, Germany, January.

Darlington, J., Field, A. J., Harrison, P. G., Harper, D., Jouret, G. K., Kelly, P. H. J., Sephton, K. M. and Sharp, D. W. (1991) Structured parallel functional programming. In: H. W. Glaser and P. H. Hartel, eds., *3rd Implementation of Functional Languages on Parallel Architectures*, pp. 31–51. Southampton, UK, June. (Also available as CSTR 91-07, Department of Electrical and Computer Science University of Southampton, UK.)

Ferguson, A. B. and Wadler, P. L. (1988) When will deforestation stop? In: C. Hall, R. J. M. Hughes and J. T. O'Donnell, eds., *Functional Programming*, pp. 39–56., Rothesay, Isle of Bute, Scotland, August. (Also Research report 89/R4, Department of Computer Science, University of Glasgow, Scotland.)

Gautier, T., le Gueric, P. and Besnard, L. (1987) SIGNAL: A declarative language for synchronous programming of real-time systems. In: G. Kahn, ed., *3rd Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 274*, pp. 257–277. Springer-Verlag.

Gill, A., Launchbury, J. and Peyton Jones, S. L. (1993) A short cut to deforestation. In: *6th Functional Programming Languages and Computer Architecture*, pp. 223–232. Copenhagen, Denmark, June.

Hartel, P. H., Glaser, H. W. and Wild, J. M. (1991) Compilation of functional languages using flow graph analysis. *Software – Practice and Experience*, **24**(2):127–173, February.

Hartel, P. H and Vree, W. G. (1992) Arrays in a lazy functional language – a case study: the fast Fourier transform. In: G. Hains and L. M. R. Mullin, eds., *2nd Arrays, Functional Languages, and Parallel Systems (ATABLE)*, pp. 52–66. Publication 841, Dept. d'informatique et de recherche opérationelle, Univ. de Montréal, Canada, June.

Jeuring, J. (1992) *Theories for algorithm calculation.* PhD thesis, Department of Computer Science, University of Utrecht, The Netherlands.

Kahn, G. (1974) The semantics of a simple language for parallel programming. In: J. L. Rosenfeld, ed., *Information processing*, pp. 471–475. Stockholm, Sweden, August.

Kelly, P. H. J. (1989) *Functional Programming for Loosely-coupled Multiprocessors*. Pitman.

Langendoen, K. G. and Hartel, P. H. (1992) FCG: a code generator for lazy functional languages. In: U. Kastens and P. Pfahler, eds., *Compiler Construction (CC): Lecture Notes in Computer Science 641*, pp. 278–296. Springer-Verlag.

Manna, Z., Ness, S. and Vuillemin, J. E. (1973) Inductive methods for proving properties of programs. *Communications of the ACM*, **16(8)**:491–502, August.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice Hall.

Turner, D. A. (1985) Miranda: A non-strict functional language with polymorphic types. In: J.-P. Jouannaud, ed., *2nd Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 201*, pp. 1–16. Springer-Verlag.

Vree, W. G. (1989) *Design considerations for a parallel reduction machine*. PhD thesis, Department of Computer Science, University of Amsterdam, December.

Vuillemin, J. E. (1973) *Proof techniques for recursive programs*. PhD thesis, Computer Science Department, Stanford University, October. Technical report STAN-CS-73-393.

Wadler, P. L. (1988) Deforestation: Transforming programs to eliminate trees. In: H. Ganzinger, ed., *European Symposium on Programming (ESOP 88). Lecture Notes in Computer Science 300*, pp. 344–358. Springer-Verlag.