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: {wille, pieter}@cs.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 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. Measurements indicate performance gains in practice both with sequential and parallel evaluation. Communication lifting is a worthwhile optimisation to be included in a compiler for a lazy functional language.

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), Esorid (Berry and Coquand, 1984), Signal (Guéhéneuc et al., 1987) and Lustre (Agerfalk 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. This is a calculation that is sus-

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 and 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 organised 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 original stream elements are accessible to the functions that used to operate on stream elements as the elements of one large tuple.

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 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 com-
Communications 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 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 fiba and fibb 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 \( \text{map}_n \) 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.

\[
\begin{align*}
\tilde{a} & = 0 : \tilde{b} & \text{ (fib.a)} \\
\tilde{b} & = 1 : \text{map}_2 (+) \tilde{a} \tilde{b} & \text{ (fib.b)}
\end{align*}
\]

Using the function \( \text{fix} \) for the fixed point operator and the standard domain construction, the solution of these equations is given by:

\[
\begin{align*}
\text{fix} \ (\tilde{a}, \tilde{b}) & = \text{fix} \ \text{fib.b} \\
\text{fib.b} \ (\tilde{a}, \tilde{b}) & = (0 : \tilde{b}, 1 : \text{map}_2 (+) \tilde{a} \tilde{b}) & \text{ (fib.def)}
\end{align*}
\]

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

\[
\begin{align*}
\tilde{x} & = \text{zip}_2 \ (\text{fix fib.b}) \\
& = \text{zip}_2 \ (\tilde{a}, \tilde{b}) & \text{(by fib.bfix)} \\
& = \text{zip}_2 \ (0 : \tilde{b}, 1 : \text{map}_2 (+) \tilde{a} \tilde{b}) & \text{(by fib.a and fib.b)} \\
& = \text{zip}_2 \ (0,1) : \text{map}_2 (+) \tilde{a} \tilde{b} & \text{(unfold zip)} \\
& = \text{map}_2 \ (\text{fst} \ x + \text{snd} \ x \ \text{and use the definition of \( \tilde{x} \)} & \text{def plus x, see law 3.35 in Jening (1992)} \\
& = \text{iterate}_0 \ \text{nextstate} \ \tilde{x} & \text{(property of iterate,0)} \\
& = \text{iterate}_0 \ \text{nextstate} \ (0,1) & \text{def nextstate a} = (\text{snd} \ a, \text{plus} \ a) \\
& = \text{iterate}_0 \ \text{nextstate} \ \tilde{x} & \text{(property of iterate,0)}
\end{align*}
\]

Hence (the \( \text{zip} \) of) the fixed point of a system of equations over streams can be calculated as follows: start with an initial state \((0,1)\) and \( \text{iterate}_0 \) over the state transformation function \( \text{nextstate} \). 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 \( \text{fib} \) example, the required output stream \( \tilde{b} \) must be recovered from the stream of pairs \( \tilde{x} \). This, and gathering the newly introduced definitions yields a complete program:

\[
\begin{align*}
\tilde{a} & = \text{map}_2 \ (\text{fst} \ \text{iterate}_0 \ \text{nextstate} (0,1)) \\
\text{nextstate} \ a & = (\text{snd} \ a, \text{plus} \ a) \\
\text{plus} \ a & = \text{fst} \ a + \text{snd} \ a
\end{align*}
\]

As a finishing touch, the definition of \( \text{nextstate} \) can be simplified by unfolding the definitions of \( \text{plus} \), \( \text{fst} \) and \( \text{snd} \) and by using pattern matching thus:

\[
\begin{align*}
\tilde{a} & = \text{map}_2 \ (\text{fst} \ \text{iterate}_0 \ \text{nextstate} (0,1)) \\
\text{nextstate} \ (\tilde{a}, \tilde{b}) & = (\tilde{b}, \tilde{c}) & \text{where} \\
\tilde{c} & = (+) \ a \ b
\end{align*}
\]

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 \( \text{map}_2 \) in the network one definition is generated under the \( \text{where} \) of the \( \text{nextstate} \) function, which captures basically the same calculation as the \( \text{map}_2 \). Also for each application of \( \text{fib} \) in the network, there is one element in the state tuple processed by the \( \text{nextstate} \) function.

The \( \text{iterate}_0 \) function controls the succession of generations of state tuples and
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 (0,1)` computes the sequence of changes in the fixed point approximation in the form of the sequence of state tuples:

\[(0,1), (1,1), (1,2), \ldots\]

This is a form of program synthesis as found in for instance Bird and Wadler (1988, Pages 134-132). 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 pipeline 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 \(\delta\) and \(\tilde{\beta}\) to the network, to produce a running total of the Fibonacci sequence in the stream \(\tilde{\beta}\):

\[
\begin{align*}
\delta &= 0 ; \tilde{\beta} \\
\tilde{\beta} &= \text{map}_\beta (+) \delta \tilde{\beta} \\
\tilde{\alpha} &= 0 ; \tilde{\beta} \\
\tilde{b} &= 1 ; \tilde{\alpha} \\
\tilde{c} &= \text{map}_\beta (+) \tilde{\alpha} \tilde{b}
\end{align*}
\]

The original sub-expression `\(\text{map}_\beta (+) \delta \tilde{\beta}\)` 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 \(\tilde{\alpha} = 0, 1, 2, 3, 5, 8, \ldots\) and then adds these numbers to produce the stream \(\tilde{\beta} = 0, 1, 2, 4, 7, 12, \ldots\).

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 `\(\text{map}_\beta (+)\)` may perform the additions with pipeline 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 pipeline to deliver speedups.

Communication lifting of the extended example produces the following program:

\[
\begin{align*}
\tilde{\alpha} &= \text{map}_\beta (\lambda \beta. \delta \tilde{\beta}) \\
\text{nextstate} (\alpha, \beta) &= (\beta, \beta + \alpha)
\end{align*}
\]

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 \((0, 0, 1)\), to which the master thus applies `nextstate`. This calls on two slaves to perform the two additions \(p = \alpha + \alpha\) and \(c = \alpha + b\). The slaves have both access to the current tuple where the values of \(\alpha, \beta\) and \(\beta\) 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 `\(\text{map}_\beta (\lambda \beta. \delta)\)`.

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 `\(\text{map}_\beta (+)\)` may perform the additions with pipeline 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 pipeline to deliver speedups.

Communication lifting of the extended example produces the following program:

\[
\begin{align*}
\tilde{\alpha} &= \text{map}_\beta (\lambda \beta. \delta \tilde{\beta}) \\
\text{nextstate} (\alpha, \beta) &= (\beta, \beta + \alpha)
\end{align*}
\]

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 \((0, 0, 1)\), to which the master thus applies `nextstate`. This calls on two slaves to perform the two additions \(p = \alpha + \alpha\) and \(c = \alpha + b\). The slaves have both access to the current tuple where the values of \(\alpha, \beta\) and \(\beta\) 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 `\(\text{map}_\beta (\lambda \beta. \delta)\)`.

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.
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 (\(\_\)), \(\text{map}_n\), \(\text{iterate}_n\) or \(\text{tl}\). 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 \(\_\) 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 \(\_\) as the root. Equations of the form \(\_\) are not permitted, they can 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 pipeline parallelism, is more difficult to harness efficiently. A more rigid paradigm (master-slave) provides the implementation more scope for optimizations than a more lenient paradigm (pipeline). A more lenient paradigm offers the programmer better possibilities for clarity and conciseness. These claims are substantiated in Section 7.

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

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

p, d, e ::= d_1 \ldots d_n (equations of the form e = v are not permitted)

c ::= \theta_0 = e_0
| \text{map}_n f e_1 \ldots e_n
| \text{iterate}_n f \_ e_1 \ldots e_n
| \_ e

\(s \in V\) (arbitrary values)

\(f, g \in F\) (arbitrary functions on streams)

\(f, g \in F\) (arbitrary functions on stream elements)

\(0, 1, \ldots, n \in \mathbb{N}\) (natural numbers)

4 The definition of a synchronous process network

In this section the definitions of the auxiliary functions and operators are shown in Figure 4. According to the syntax, only \(\text{tl}(\_\_\_\_\_\_\_\), \(\text{map}_n\), and \(\text{iterate}_n\) 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 \textit{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 (\(\text{tl}\)) and a third function to perform an arbitrary computation on a stream element (\(\text{map}_n\)). A fourth function \(\text{iterate}_n\) has been
Communication lifting

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 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 70 of Figure 5. The notation employed is more or less standard (see, for example, Ferguson and Wattier (1988)). The transformation rules take a syntactic argument enclosed in emphasis 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 (if ...) =⇒ connecting the left and right hand side 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 $\cdot$ and that the next $m$ equations must contain an application of map-$. The remaining $l$ equations must use $\cdot$. At this stage applications of iterate $\cdot$, equations of the form $0 = \bar{w}$, 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 $\bar{x}$ and $\bar{y}$ in the left hand side of Clause 0a will be the same as some of the variables $\bar{z}$, $\bar{t}$, $\bar{u}$, or $\bar{v}$. The $\bar{x}$ and $\bar{y}$ that are not defined within the network act as the input streams to the network. These input streams are identified by the set $\{\bar{u}, ... \bar{u}\}$.

The guard in Clause 0a ensures that the streams $\bar{z}, ... \bar{t}$, and the states $\bar{s}, ... \bar{s}$, 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 $e_1 ... e_n$, 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 $\bar{v}, ... \bar{v}$ are empty. This is consistent with the view that streams are infinite (lists).

5 Communication lifting of a synchronous process network

and the functions $f_j, ... f_m$ are not dependent on any of the streams $\bar{v}, \bar{v}, ... \bar{v}$, and $\bar{w}, ... \bar{w}$. This is necessary because the transformation removes the definitions of the streams $\bar{v}$ and $\bar{w}$.

The best way to understand Rule 70 is to try it out on a simple example. The program produced by applying Rule 70 to the Fibonacci network of Section 3 yields:

$$\begin{align*}
\bar{p} &= s_1 : x_1 & | \bar{p} &= 0 : \bar{p} \\
\bar{v} &= s_2 : x_2 & | \bar{v} &= 0 : b \\
\bar{v} &= s_3 : x_3 & | \bar{v} &= b \cdot 1 : \bar{v} \\
\bar{w} &= map_2 f_1 \bar{y}_1 \bar{y}_2 & | \bar{w} &= map_2 (+) \bar{p} \bar{b} \\
\bar{w} &= map_2 f_2 \bar{y}_1 \bar{y}_2 & | \bar{w} &= map_2 (+) \bar{v} \bar{b}
\end{align*}$$

Here we have shown the correspondence between formal and actual identifiers of the Rule 70. The guard of Clause 0a is satisfied; because there are no equations of type $l = \bar{v}$, and because $s_1 = 0, s_2 = 0, s_3 = 1, f_1 = (+), f_2 = (+)$ do not depend on $\bar{N} = \{\bar{v}, \bar{v}, \bar{b}, \bar{p}, \bar{c}\}$. For the Fibonacci-sum program the set $\{\bar{u}, ...\bar{u}\}$ is
\[ \hat{\alpha} = \text{iterate}_{A} (\text{iterate}_{B} \text{nextstate} \emptyset 0 1) \]

\[ \text{nextstate} \alpha = \{p,b,c\} \]

\[ \text{where} \]

\[ (o,a,b) = s \]

\[ p = (+) a b \]

\[ c = (+) a b \]

\[ T_{1} \mid d_{1} \ldots d_{m} \mid \rightarrow T_{2} \mid v = \text{iterate}_{A} f \ldots (g_{i} \ldots g_{n}) \mid \quad \text{(1)} \]

\[ T_{2} \mid v = f \ldots (g_{i} \ldots g_{n}) \mid \rightarrow T_{2} \mid v = f \ldots v' \ldots \mid \quad \text{(2a)} \]

\[ \text{where} \ w \text{is a new variable} \]

\[ T_{3} \mid v = c \mid \rightarrow v = c \quad \text{(2b)} \]

\[ T_{3} \mid d_{1} \ldots d_{m} \mid \rightarrow T_{4} \mid d_{1} \ldots d_{m} \mid \quad \text{(3)} \]

\[ T_{4} \mid v = \text{iterate}_{A} f s x_{1} \ldots x_{n} \mid \rightarrow v = s : w \]

\[ w = \text{map}_{n} f v z_{1} \ldots z_{n} \]

\[ \text{where} \ v \text{and} w \text{are new variables and} m = n + 1 \]

\[ T_{5} \mid d \mid \rightarrow d \quad \text{(4b)} \]

\[ T_{5} \mid \hat{\alpha} = \text{map}_{n} f z_{1} \ldots z_{n} \]

\[ d_{2} \ldots d_{n} \mid \rho \rightarrow T_{5} \mid \hat{\alpha} = \text{map}_{n} f z_{1} \ldots z_{n} \]

\[ v = \text{iterate}_{A} (\text{iterate}_{B} \text{nextstate} \emptyset 0 1) \]

\[ \text{where} \ v \text{and} w \text{are new variables} \]

\[ T_{6} \mid v = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \mid \rightarrow T_{6} \mid \hat{\alpha} = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \]

\[ v = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \]

\[ w = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \]

\[ \text{where} \ v \text{and} w \text{are new variables} \]

\[ T_{7} \mid v = \text{iterate}_{A} f s x_{1} \ldots x_{n} \mid \rightarrow v = s : w \]

\[ w = \text{map}_{n} f v z_{1} \ldots z_{n} \]

\[ \text{where} \ v \text{and} w \text{are new variables and} m = n + 1 \]

\[ T_{8} \mid \hat{\alpha} = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \]

\[ T_{9} \mid \hat{\alpha} = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \]

\[ v = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \]

\[ w = \text{iterate}_{A} f (h_{1} x_{1}) \ldots (h_{d} x_{d}) \]

\[ \text{where} \ v \text{and} w \text{are new variables} \]

\[ T_{10} \mid p \mid \rho \rightarrow p \quad \text{(otherwise)} \]

\[ \text{Fig. 6. Simplifying transformations:} \quad p' = T_{5} (T_{3} (T_{1} p)) \emptyset. \]
**Rule T3: removing calls to the function iterate_v**

The purpose of Rule T3 is to simplify the network by replacing all occurrences of the function iterate_v by applications of the simpler map_v 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_v is transformed into an equation with (.) and an equation with map_v. 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 \( \hat{a} \) by two new equations. The equations for \( \hat{a} \) and \( \hat{q} \) remain unchanged:

\[
\begin{align*}
\hat{o} &= 0 : \hat{p} \\
\hat{p} &= \text{map}_2(+) \hat{a} \hat{q} \\
\hat{q} &= \hat{a} : \hat{h} \\
\hat{h} &= 1 : \hat{c} \\
\hat{c} &= \text{map}_2(+) \hat{a} \hat{d} \\
\hat{d} &= \hat{i} \hat{a}
\end{align*}
\]

**Rule T5: removing redundant calls to ()**

The goal of the final simplifying Rule T5 is to remove as many applications of () 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 ()

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 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 \( \hat{i} = \hat{l} \hat{w} \) and the corresponding equation \( \hat{w} = s : \hat{x} \).

The remaining equations are named \( d_1 \ldots d_m \) 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 \( (=) \). This is necessary to ensure that Rule T5 will terminate on all inputs. The labels are used in the guard \( \text{if } [j,k] \neq \rho \Rightarrow \text{Clause 5d} \) to avoid this clause from looping on a pair of definitions such as \( \hat{i} = \hat{l} \hat{w} \hat{w} = \text{map}_f j \hat{v} \). The labelling scheme to 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_v once and introduces two new equations so that \( \hat{a} = \ldots \hat{a} \) as required.

Clause 5c resolves a combination of \( \hat{i} = \hat{l} \hat{w} \) and \( \hat{w} = s : \hat{x} \) by replacing every free occurrence of \( \hat{w} \) in the process network by \( \hat{x} \). This replacement is expressed as \( \ldots \hat{x} \hat{i} \hat{t} \). The equation for \( \hat{v} \) is then removed. The \( \hat{t} \) is thus cancelled against the \( \hat{i} \). A combination of \( \hat{i} = \hat{l} \hat{w} \) and \( \hat{w} = \text{map}_v \ldots \) (Clause 5d) can be resolved in a similar way, after unfolding the definition of map_v once. Then the \( \hat{t} \) can be cancelled against the \( \hat{i} \).

Clause 5b is a special case of Clause 5c. Both clauses cancel an application of () against an application of () but the output stream of the network \( \hat{a} \) must be treated specially. If Clause 5b were omitted, Clause 5c applied to \( \hat{i} = \hat{l} \hat{w} \) would remove the equation for \( \hat{a} \) 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 \( \hat{i} = \hat{l} \hat{w} \) with \( \hat{w} = \hat{x} \hat{w} \), because a combination of two or more () applications is resolved by cancellation of the last () against either map_v or (), followed by cancellation of the penultimate () etc. A combination of equations involving applications of () can not be removed by cancellation if the last () 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 () 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.

\[
\begin{align*}
\hat{o} &= 0 : \hat{p} \\
\hat{p} &= \text{map}_2(+) \hat{a} \hat{q} \\
\hat{q} &= \hat{a} : \hat{h} \\
\hat{h} &= 1 : \hat{c} \\
\hat{c} &= \text{map}_2(+) \hat{a} \hat{d} \\
\hat{d} &= \hat{i} \hat{a}
\end{align*}
\]

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 (Burtsell and Darlington, 1971) and uses algebraic properties of stream functions such as map_v and zip_v. 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 important performance issues can be distinguished. The first is the gain or loss in
## Communication lifting

Show the sequential execution synchronous process networks with an indication of their size and 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 3 streams before the transformation and still 2 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 4 CPUs, 4 instruction caches and 4 coherent data caches and 64Mb 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

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

<table>
<thead>
<tr>
<th>System program</th>
<th>Source lines</th>
<th>Sequential orig. time (ms)</th>
<th>Sequential max. time (ms)</th>
<th>Master-slave pipe orig. time (ms)</th>
<th>Master-slave pipe max. time (ms)</th>
<th>Tuple size</th>
<th>Stream length</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibsum</td>
<td>6</td>
<td>627</td>
<td>493</td>
<td>477</td>
<td>596</td>
<td>3</td>
<td>3000</td>
</tr>
<tr>
<td>flipflop</td>
<td>28</td>
<td>2434</td>
<td>700</td>
<td>581</td>
<td>974</td>
<td>13</td>
<td>1600</td>
</tr>
<tr>
<td>fft</td>
<td>190</td>
<td>3045</td>
<td>2506</td>
<td>1364</td>
<td>1504</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>wave4</td>
<td>304</td>
<td>2941</td>
<td>2831</td>
<td>1111</td>
<td>1161</td>
<td>3</td>
<td>200</td>
</tr>
</tbody>
</table>

*a* Sums the first 3000 Fibonacci numbers.

*b* Simulation of a D-Flipflop over 1000 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 x 4 grid points of the North Sea over 200 time steps (Vree, 1989).

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

† Miranda is a trademark of Research Software Ltd.
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
\text{pforce}.

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

\[
\begin{align*}
\bar{\tau} &= \text{map}_\mathcal{L} \{ \text{sel}_\mathcal{L} \bar{1} \} \{ \text{iterate}_\mathcal{L} \text{nextstate}_\mathcal{L} \{ 0, 0, 1 \} \} \\
\text{nextstate}_\mathcal{L} a &= \text{pforce} \{ p, b, c \}
\end{align*}
\]

where

\[
\begin{align*}
\langle a, b, c \rangle &= a \\
p &= (\langle + \rangle) a \ b \\
c &= (\langle + \rangle) a \ b
\end{align*}
\]

The two additions \((+ \ b) \ a\) and \((+ \ a) \ b\) are thus evaluated in parallel. The first
tuple ever created by the program contains three numbers, all of which are normal
forms: \((0, 0, 1)\). Because of the strict semantics of \text{pforce} as required by the master-
slave style parallel evaluation, each subsequent tuple will be fully normalised when
created. The sharing between computations is maintained as usual. In the example,
both additions use the variable \(a\), 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 \text{nextstate}. Because
of the completely strict semantics of \text{pforce}, all state tuple elements are fully nor-
malised before \text{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 in this 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 \text{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 these programs is improved by parallel execution, that of the \text{fibsum}
program is not affected.

The grammatical nature of the \text{fibsum} program is only one addition, just about suf-
cient to compensate the overhead of process creation by \text{pforce}. But the \text{flipflop}
program, still quite fine-grained, is already faster than the sequential version. This
shows that the overhead of \text{pforce} is indeed low.

The \text{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

![Fig. 7. Two processors performing pipe-line parallel graph reduction in a shared memory system. Processor 1 computes the Fibonacci-sum stream \(\bar{\tau}\) and Processor 2 computes the Fibonacci stream \(\bar{\sigma}\). The boxes represent suspended computations, all other nodes represent data.](image-url)
Communicating lifting

comparisons, all other nodes represent data. Applications of \texttt{bit} and \texttt{tl} (see the
definitions of \texttt{map} and \texttt{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 \texttt{map,tl} \((+\ldots)\). 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 pipeline 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
subgraph. The locking mechanism uses the \texttt{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 ac-
cessed 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
straightforward, 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 coarse grained
parallel execution. The best results are shown in the column marked \texttt{pipe/orig.}
of Table 1. We do not expect a dynamically scheduled pipeline to be much better,
because measurements have shown that the statically scheduled pipeline 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 pos-
sible 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 sequential-
ly on the master slave system offer some additional parallelism for the pipeline
system. This effect is particularly strong on the \texttt{FFX} 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 synchroni-

\texttt{communication lifting} essen
tially extracts the synchronisation mechanism at compile time, resulting in the iteration of the function \texttt{nextstate},

This function contains the knowledge of the communication pattern, while \texttt{iterate}
provides the synchronisation in the most efficient way.

8 Related work

Many languages have been developed to support programming with process net-
works, which gives an indication of the importance of work in this area. Luc-
id (Ashcroft and Wedge, 1977) is one of the first languages based on the notion
that variables represent not a single value, but a potentially infinite history of val-
ues. 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 conven-
tional in the sense that it uses a special purpose compiler. Our synchronous pro-
cess 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 per-
formance 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 syn-
chronous 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 \texttt{pre}, \texttt{→}, \texttt{when} and \texttt{current} using only the four stream functions \((:),\texttt{tl},\texttt{map},\texttt{and} \texttt{iterate} as defined in Figure 4.

Each Lustre sequence represents a stream of values, with which a clock is asso-
ciated. A clock can be thought of as a stream of boolean values. The \texttt{zip} of the
stream of values and the associated clock is a sensible representation of a Lustre sequence:

\[
\text{clock } \hat{x} = \text{map}_{\ldots} c \hat{x} \quad \text{where} \quad c \hat{x} = (\text{True}, \hat{x})
\]

The Lustre sequence operators \texttt{pre} and \texttt{→} correspond to \((:)\) and \texttt{tl} respectively:

\[
\text{pre } \hat{x} = (\text{True}, \ldots) : \hat{x} \\
\hat{x} → \bar{y} = \text{hd } \hat{x} : \text{tl } \bar{y}
\]

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

W. G. Vree and P. H. Hartel

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 pipeline fashion, shows that communication lifting outperforms a pipeline 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 Gravel, Koos Langendoen, Henk Muller and the two referees for their comments on a draft version of the paper. Andy Gravel suggested the derivation of the communication lifting transformation in section 2.

References

P. Caspi, D. Plass, N. Hallowell, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In 14th Principles of programming languages, pages 176-188, Munich, West Germany, Jan 1987. ACM.
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 $\bot$.

The zip-lemma
Given $\forall n \in \mathbb{N} \cdot \bar{x}_1, \ldots, \bar{x}_n \in \mathbb{D}$, then:
\[
\forall i \in \mathbb{N} : (\text{zip}_n \, [\bar{x}_1, \ldots, \bar{x}_n])_i = ([\bar{x}_1]_i, \ldots, [\bar{x}_n]_i)
\]
This can be proved by induction on $i$.

The map-lemma
Given $\forall \bar{x}_1, \ldots, \bar{x}_n \in \mathbb{D} \Rightarrow f \in \mathbb{F}$, then:
\[
\forall i \in \mathbb{N} : (\text{map}_n \, f \, \bar{x}_1 \ldots \bar{x}_n)_i = f \, ([\bar{x}_1]_i \ldots [\bar{x}_n]_i)
\]
This can be proved by induction on $i$.

The iterate-lemma
Given $\forall \bar{x}_1, \ldots, \bar{x}_n \in \mathbb{D} \Rightarrow f \in \mathbb{F} \Rightarrow s \in \mathbb{D}$, then:
\[
\forall i \in \mathbb{N} : (\text{iterate}_n \, f \, s \, \bar{x}_1 \ldots \bar{x}_n)_i = \hat{y}_i
\]
where
\[
\hat{y}_i = s : \text{map}_m \, f \, \hat{y}_0 \hat{y}_1 \ldots \hat{y}_{m-1}
\]
\[m = n + 1\]

To prove this lemma, consider the special case that $n = 1$. Then given $\hat{x} \in \mathbb{D}$ and the binary function $f \in \mathbb{F}$, the iterate-lemma is:
\[
\forall i \in \mathbb{N} : (\text{iterate}_1 \, f \, s \, \hat{x})_i = \hat{y}_i
\]
where
\[
\hat{y}_i = s : \text{map}_m \, f \, \hat{y}_0 \hat{y}_1 \ldots \hat{y}_{m-1}
\]
\[m = n + 1\]

The proof is by induction on $i$. Base case:
\[
(\text{iterate}_1 \, f \, s \, \hat{x})_0 = \hat{y}_0 = \text{fold} \, (\text{iterate}_1 \, f \, s \, \hat{x}) (\text{fold} \, (\text{iterate}_1 \, f \, s \, \hat{x})) (\text{new definition, fold} !)
\]
\[= s (\text{fold} (\text{iterate}_1 \, f \, s \, \hat{x})) \quad \text{(unfold fold !)}
\]
\[= s \quad \text{(unfold iterate, base case !)}
\]
\[= \hat{y}_0 \quad \text{(definition, fold !)}
\]
\[= \hat{y}_i \quad \text{(definition, fold !)}
\]
Induction step:

\[
\text{iterate}_f \cdot s \cdot x! (i + 1) = (\text{iterate}_f \cdot s \cdot \text{hd} \cdot x! (i + 1)) \quad \text{(unfold)}
\]

\[
= (\text{iterate}_f \cdot s \cdot \text{map} \_f \cdot \text{map} \_g \cdot (\text{iterate}_f \cdot s \cdot (\text{hd} \cdot x! (i + 1)))) \quad \text{(unfold iterate \_f)}
\]

\[
= \text{iterate}_f \cdot s \cdot \text{map} \_f \cdot \text{map} \_g \cdot (\text{iterate}_f \cdot s \cdot (\text{hd} \cdot x! (i + 1))) \quad \text{(unfold iterate \_f)}
\]

\[
= \text{iterate}_f \cdot s \cdot \text{map} \_f \cdot \text{map} \_g \cdot (\text{iterate}_f \cdot s \cdot \text{hd} \cdot x! (i + 1)) \quad \text{(iterate \_f \_lemma)}
\]

\[
= \text{iterate}_f \cdot s \cdot \text{map} \_f \cdot \text{map} \_g \cdot \text{map} \_h \cdot (\text{iterate}_f \cdot s \cdot \text{hd} \cdot x! (i + 1)) \quad \text{(iterate \_f \_lemma)}
\]

\[
\text{where}
\]

\[
\text{iterate}_f \cdot s \cdot \text{map} \_f \cdot \text{map} \_g \cdot \text{map} \_h \cdot (\text{iterate}_f \cdot s \cdot \text{hd} \cdot x! (i + 1))
\]

\[
= \text{iterate}_f \cdot s \cdot \text{map} \_f \cdot \text{map} \_g \cdot \text{map} \_h \cdot (\text{iterate}_f \cdot s \cdot \text{hd} \cdot x! (i + 1)) \quad \text{(iterate \_f \_lemma)}
\]

\[
\text{where}
\]

\[
\text{iterate}_f \cdot s \cdot \text{map} \_f \cdot \text{map} \_g \cdot \text{map} \_h \cdot (\text{iterate}_f \cdot s \cdot \text{hd} \cdot x! (i + 1))
\]

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

**Corollary of the iterate-lemma**

Given \( \forall \bar{x}_1 \ldots \bar{x}_n \in \bar{D} \land f \in F \land s \in D \land m = n + 1 \), then:

\[
\bar{y} = s : \text{map}_m f \cdot \bar{y} \cdot \bar{x}_1 \ldots \bar{x}_n
\]

\[
\forall i \in \mathbb{N} : \bar{y} \cdot i = (\text{iterate}_f \cdot s \cdot \bar{x}_1 \ldots \bar{x}_n) \cdot i
\]

\[
= \text{iterate}_f \cdot s \cdot \bar{x}_1 \ldots \bar{x}_n
\]

**The element-lemma**

Given \( \forall \bar{x}, \bar{y} \in \bar{D} \) then:

\[
\bar{x} = \bar{y} = \forall i \in \mathbb{N} : \bar{x} \cdot i = \bar{y} \cdot i
\]

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

\[
\forall i \in \mathbb{N} : \text{take} \cdot i \cdot \bar{x} = [\bar{x} \cdot i | k \rightarrow 0 \ldots i - 1]
\]

\[
\forall i \in \mathbb{N} : \text{take} \cdot i \cdot \bar{y} = [\bar{y} \cdot i | k \rightarrow 0 \ldots i - 1]
\]

The proof of the auxiliary lemma is by induction on \( i \). Base case:

\[
\text{take} \cdot 0 \cdot \bar{x} = [\bar{x} \cdot 0 | k \rightarrow 0]
\]

\[
\text{take} \cdot 0 \cdot \bar{y} = [\bar{y} \cdot 0 | k \rightarrow 0]
\]

Induction step:

\[
\text{take} \cdot (i + 1) \cdot \bar{x} = [\bar{x} \cdot (i + 1) | k \rightarrow 0 \ldots i - 1]
\]

\[
\text{take} \cdot (i + 1) \cdot \bar{y} = [\bar{y} \cdot (i + 1) | k \rightarrow 0 \ldots i - 1]
\]

The unfold step is permitted because streams are infinite lists and not partial lists.

The element-lemma can now be proved as follows:

\[
\bar{x} = \bar{y}
\]

\[
\forall i \in \mathbb{N} : \text{take} \cdot i \cdot \bar{x} = \text{take} \cdot i \cdot \bar{y}
\]

\[
\forall i \in \mathbb{N} : \text{take} \cdot i \cdot \bar{y} = \text{take} \cdot i \cdot \bar{y}
\]

\[
\forall i \in \mathbb{N} : \bar{y} \cdot i = \bar{y} \cdot i
\]
The stream-lemma

When given $z_1, \ldots, z_n \in \bar{D} \land z_1, \ldots, z_n \in D \land A \in F \land \tilde{A} \in \hat{F}$ and the following 4 conditions are met:

(i) $\tilde{A}(z_1, \ldots, z_n) = \tilde{e}_i[z_1, \ldots, z_n, \tilde{a}_1 \ldots \tilde{a}_m]$
   where
   
   \[ \tilde{a}_1 = e_i[z_1, \ldots, z_n, a_1 \ldots a_m] \]
   
   \[ \vdots \]
   
   \[ \tilde{a}_m = e_m[z_1, \ldots, z_n, a_1 \ldots a_m] \]

(ii) $A(z_1, \ldots, z_n) = e_0[z_1, \ldots, z_n, a_1 \ldots a_m]$
   where
   
   \[ a_1 = e_1[z_1, \ldots, z_n, a_1 \ldots a_m] \]
   
   \[ \vdots \]
   
   \[ a_m = e_m[z_1, \ldots, z_n, a_1 \ldots a_m] \]

(iii) there are no free occurrences of either $\tilde{A}$ or $A$ in $a_0, \ldots, a_m$ or $e_0, \ldots, e_m$

(iv) $\forall i \in N \land 0 \leq j \leq m$ we have:
   
   \[ (e_j[z_1, \ldots, z_n, a_1 \ldots a_m])\hat{B} = e_j[z_1, \ldots, z_n, a_1 \ldots a_m, b_0, \ldots, b_n] / a_0, \ldots, a_m] / a_0, \ldots, a_m]

Then the stream-lemma asserts that:

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

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 $e[b_1, a_1 \ldots a_m]/a_0, \ldots, a_m]$ is an expression $e$ in which the free variables $a_1 \ldots a_m$ are simultaneously replaced by respectively $b_1, \ldots, a_m$. For brevity we use a superscript in the proof to denote projection on tuples rather than the function $\text{sel}_m$:

\[ \forall 1 \leq k \leq m : (a_1, \ldots, a_m)^k = a_k \]

We begin the proof by associating a function $\tilde{f}_j$ with each of the expressions $e_j[z_1, \ldots, z_n, a_1 \ldots a_m]$ in (i), such that:

\[ \forall 0 \leq j \leq m : \tilde{f}_j = \lambda x.e_j[x]^0 / z_1, \ldots, x^n / z_1, \ldots, z_n, a_1, a_2, \ldots, a_m] / a_0, \ldots, a_m] / a_0, \ldots, a_m]

Thus $\tilde{f}_j(z_1, \ldots, z_n)(a_1, \ldots, a_m) = e_j$. In the same way associate functions $f_0, \ldots, f_m$ with the expressions $e_0, \ldots, e_m$.

As the second step rewrite (v) in terms of the functions $f_0, \ldots, f_m$.

\[ \forall i \in N \land 0 \leq j \leq m : (e_j[z_1, \ldots, z_n, a_1 \ldots a_m])\hat{B} = f_j[z_1, \ldots, z_n, b_0, \ldots, b_n, a_1, a_2, \ldots, a_m] / a_0, \ldots, a_m] / a_0, \ldots, a_m]

\[ (f_j[z_1, \ldots, z_n] a_1, \ldots, a_m])\hat{B} = f_j[z_1, \ldots, z_n, b_0, \ldots, b_n, a_1, a_2, \ldots, a_m] / a_0, \ldots, a_m] / a_0, \ldots, a_m]

\[ (f_j a)\hat{B} = f_j((\text{zip}_h z) a)] / (\text{zip}_h a)] / (\text{zip}_h a)]

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

(vii) $\tilde{A}(z_1, \ldots, z_n) = \tilde{f}_0(z_1, \ldots, z_n)(a_1, \ldots, a_m)$

where

\[ a_1 = f_1(z_1, \ldots, z_n)(a_1, \ldots, a_m) \]

\[ \vdots \]

\[ a_m = f_m(z_1, \ldots, z_n)(a_1, \ldots, a_m) \]

(viii) $B(z_1, \ldots, z_n) = B(z_1, \ldots, z_n)\hat{B}$

where

\[ B = \lambda x.f_0 z B[z] \]

\[ E[z] = f_0(\lambda x(f_1 z \ldots, f_m z \ldots)) \]

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

\[ A(z_1, \ldots, z_n) = B(z_1, \ldots, z_n) \]

\[ \forall i \in N : (zip_h z) a)] / (\text{zip}_h a)] / (\text{zip}_h a)]

\[ B(z_1, \ldots, z_n) = B((\text{zip}_h z) a)] / (\text{zip}_h a)] / (\text{zip}_h a)]

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

\[ A(z_1, \ldots, z_n) = \hat{B}(z_1, \ldots, z_n) \]
To prove the stream-lemma (vi) it thus remains to show that:
\[
\begin{align*}
\hat{B} (z_1, \ldots, z_h) &= \text{map}_I B (zip_J (z_1, \ldots, z_h)) \\
&= (\text{element-lemma})
\end{align*}
\]

\(\forall i \in \mathbb{N} : \quad (\hat{B} (z_1, \ldots, z_h))_i = (\text{map}_I B (zip_J (z_1, \ldots, z_h)))_i = (\text{map-lemma})
\]

Using the definitions of \(\hat{B}\) and \(B\) we can derive the following fixpoint equation for \((zip_m E[z])_i\):
\[
\begin{align*}
\forall i \in \mathbb{N} : \quad (zip_m E[z])_i &= (zip_m \text{map}[\lambda a. (f_1 z a, \ldots, f_m z a)]))_i \\
&= (zip_m (f_1 z E[z], \ldots, f_m z E[z]))_i \\
&= ((f_1 z E[z], \ldots, f_m z E[z])_i) \\
&= ((f_1 z E[z])_i, \ldots, (f_m z E[z])_i) \\
&= (by \ (vi)) \\
&= (\text{zip-lemma}) \\
&= (\text{by (vi))} \\
&= (\text{abstraction}) \\
&= (\text{fold } E) \\
&= (\text{fold } E)
\end{align*}
\]

So for all \(h\)-tuples \(z\) we also have:
\[
(\forall i \in \mathbb{N} : (zip_m E[z])_i = E[(zip_m E[z])_i])
\]

The proof of the stream-lemma can now be completed as follows:
\[
\begin{align*}
\hat{B} (z_1, \ldots, z_h) &\quad (\forall i \in \mathbb{N} : (zip_m E[z])_i = E[(zip_m E[z])_i]) \\
&= ((f_1 z E[z], \ldots, f_m z E[z])_i) \\
&= (by \ (vi)) \\
&= (\text{reduction}) \\
&= (by \ (vi)) \\
&= (\text{reduction}) \\
&= (\text{reduction})
\end{align*}
\]

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} : f \in F \land z \in D \land x_1, \ldots, x_m \in \hat{D}:
\]
\[
\begin{align*}
\hat{b} &= \sigma (map_m f \hat{y} \hat{x}_1, \ldots, \hat{x}_m) \\
n &= n + 1
\end{align*}
\]

Then:
\[
\hat{b} = \text{iterate}_n f \hat{x}_1, \ldots, \hat{x}_m
\]

The proofs of the remaining clauses present no difficulties. The Clauses 1, 2b, 3, 4b and 5b by themselves make no change. 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 \(map_a\) as given in Figure 4 and introduces new equations; Clauses 5b and 5c unfold the definitions of \(\hat{b}\) and \(n\). For Clause 5b this gives the equation \(\hat{b} = \hat{x}\) and for \(5c\), we have \(\hat{b} = \hat{x}\). These equations can be eliminated by renaming \(\hat{x}\) to \(a\) and \(\hat{b}\) to \(\hat{x}\) respectively. Clause 5d unfolds the definitions of \(\text{map}_m, \hat{w}\) and \(\hat{t}\) 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 5c 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 is applied at most once. The only clause of Rule T5 that matches an equation of the form \(\sigma = \cdots = \hat{x}\) 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 \( f \) and \( map_s \) 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 \( f \) 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.

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

---

left and right hand side of Clause 6a | stream-lemma functions \( \vec{A} \) and \( A \)
---|---
\( \vec{A} \triangleright a_1, \ldots, a_k \) | \( \vec{A} \triangleright z_1, \ldots, z_n \)
= \( zip(A_1, \ldots, A_k) \) | = \( e_0 \)
where \( v_1 = rel_1 k \triangleright (\text{unzip}_k s) \) | \( \vec{A} \triangleright a_1, f \)
\( \vdots \) | \( \vec{A} \triangleright a_k, f \)
\( w_i = map_s a_i f \triangleright y_{i1}, \ldots, y_{im_i} \) | \( \vec{A} \triangleright a_{k+i} \)
\( \vdots \) | \( \vec{A} \triangleright \)
\( w_m = map_s w_m f \triangleright y_{m1}, \ldots, y_{mm_m} \) | \( \vec{A} \triangleright a_{k+m} \)

Fig. 8. Structural correspondence between the stream-lemma and \( T0 \).

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 \( \vdash \) and \( map_s \) equations on the left hand side of Rule \( T0 \) are copied; the \( \vdash \) equations are left unchanged. The gamed in Clause 6a ensures that the \( \vdash \) equations are independent of the remaining equations. Therefore it is correct to consider the network without the \( \vdash \) equations. From now on let \( \bar{v}_1 = \bar{\delta} \).

---

Then:

\[
\begin{align*}
\bar{v}_1 &= \{a_1 : \bar{\delta}_1\} \\
\vdots \\
\bar{w}_1 &= \{a_n : \bar{\delta}_k\} \\
\bar{w}_m &= \{w_m : \bar{\delta}_m\}
\end{align*}
\]

where \( \bar{w}_m = map_s w_m f \triangleright y_{m1}, \ldots, y_{mm_m} \)

The function composition \( unzip_k \circ zip_k \) is the identity function on a tuple of streams, thus the left hand side of Clause 6a is equivalent to:

\[
\begin{align*}
\bar{v}_1 &= unzip_k \circ (zip_k (a_1 : \bar{\delta}_1, \ldots, a_k : \bar{\delta}_k)) \\
\bar{w}_m &= (unzip_k \circ (zip_k (a_1 : \bar{\delta}_1, \ldots, a_k : \bar{\delta}_k)))
\end{align*}
\]

Next a new equation \( \bar{\alpha} \) for the stream of \( k \)-tuples is introduced, and the free variables of the network \( \bar{v}_1 \ldots \bar{v}_k \) are made explicit. This yields the following set of equations, again equivalent to the left hand side of Clause 6a:

\[
\begin{align*}
\bar{v}_1 &= unzip_k \circ (zip_k (a_1 : \bar{\delta}_1, \ldots, a_k : \bar{\delta}_k)) \\
\bar{w}_m &= (unzip_k \circ (zip_k (a_1 : \bar{\delta}_1, \ldots, a_k : \bar{\delta}_k)))
\end{align*}
\]

(2i) holds \( \forall \bar{v} \in \mathbb{N}^k \exists j \leq k \) \( \bar{\delta} \).

Then:

\[
\begin{align*}
\bar{v}_1 &= \{a_1 : \bar{\delta}_1\} \\
\vdots \\
\bar{w}_1 &= \{a_n : \bar{\delta}_k\} \\
\bar{w}_m &= \{w_m : \bar{\delta}_m\}
\end{align*}
\]

where \( \bar{w}_m = map_s w_m f \triangleright y_{m1}, \ldots, y_{mm_m} \)

In the last step of this derivation, we have used the fact that the states \( a_1 \ldots a_k \) are independent of the stream variables \( \bar{v}_1 \ldots \bar{v}_k \) and \( \bar{w}_1 \ldots \bar{w}_m \) (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 \( \bar{\delta} \) as derived from the equations on the left hand side of Clause 6a 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 \( \bar{\delta} = map_s \circ \bar{A} \), provided stream-lemma condition (2i) holds \( \forall \bar{v} \in \mathbb{N}^k \exists j \leq k + m \). This is verified as follows:

For \( j = 0 \) we have that \( \bar{v}_1 = unzip_k \circ (\bar{\delta}_1, \ldots, \bar{\delta}_k) \) and also that \( \bar{v}_1 = \{a_1 : \bar{\delta}_1, \ldots, a_k : \bar{\delta}_1\} \).

† The where equations for \( w_0, \ldots, w_m \) have been omitted.
can be proved as follows:

\[ \forall \alpha \in \text{IN} : \]
\[ (\pi \gamma_{\alpha} (\bar{\chi}, \ldots, \bar{\chi})) = (\bar{\chi}, \bar{\chi}, \ldots, \bar{\chi}) \quad \text{(zip-lemma)} \]
\[ = (\bar{\chi}, \bar{\chi}, \ldots, \bar{\chi}) \quad \text{(stream-lemma)} \]
\[ = \{ \bar{\chi}, \bar{\chi}, \ldots, \bar{\chi} \} \quad \text{(stream-lemma)} \]

For \( 1 \leq j \leq k \) we have that \( \epsilon_j = \text{sel}_{\alpha} \bar{\gamma}_{\alpha} (\text{unzip}_{\alpha} \bar{\gamma}) \) and also \( \epsilon_j = \text{sel}_{\alpha} \bar{\gamma} \) because:

\[ \forall \alpha \in \text{IN} : \]
\[ (\text{sel}_{\alpha} \bar{\gamma}_{\alpha} (\text{unzip}_{\alpha} \bar{\gamma})) = (\text{sel}_{\alpha} \bar{\gamma}_{\alpha} (\text{zip}_{\alpha} (\bar{\gamma}))) = (\text{sel}_{\alpha} \bar{\gamma}_{\alpha} (\bar{\gamma})) = (\text{sel}_{\alpha} \bar{\gamma}) = (\text{sel}_{\alpha} \bar{\gamma}) = (\text{sel}_{\alpha} \bar{\gamma}) \quad \text{(fold sel}_{\alpha} \bar{\gamma})) \]

For \( k + 1 \leq h \leq k + m \) we have \( \epsilon_{j+1} = \text{map}_{\alpha} f_j \bar{y}_{j+1} \ldots \bar{y}_{j+m} \) and \( \epsilon_{j+1} = f_j \bar{y}_{j+1} \ldots \bar{y}_{j+m} \) since:

\[ \forall \alpha \in \text{IN} : \]
\[ (\text{map}_{\alpha} f_j \bar{y}_{j+1} \ldots \bar{y}_{j+m}) = (f_j \bar{y}_{j+1} \ldots \bar{y}_{j+m}) \quad \text{(map-lemma)} \]

Given the definition of \textit{nextstate} as in Figure 5, (which is the same as \( A \) here) we have:

\[ \bar{\gamma} = (s_1, \ldots, s_k) : (A \bar{u} \bar{u}_1 \ldots \bar{u}_k) \]
\[ = \quad \text{(stream-lemma)} \]
\[ \bar{\gamma} = (s_1, \ldots, s_k) : (\text{map}_{\alpha} A \bar{u} \bar{u}_1 \ldots \bar{u}_k) \]
\[ = \quad (A = \textit{nextstate} \text{ and corollary of iterate-lemma}) \]
\[ \bar{\gamma} = \textit{iterate}_{\alpha} \textit{nextstate} (s_1, \ldots, s_k) \bar{u}_1 \ldots \bar{u}_k \quad \text{(iterate-lemma)} \]

This completes the correctness proof of transformation Rule T0.