On the Power of Recursion in Dataflow Schemes

Peter Bachmann

Cottbus University of Technology, Institute of Computer Science
P.O. Box 10 13 44, D-03013 Cottbus
e-mail: pb@informatik.tu-cottbus.de

Abstract. Dataflow Schemes are formalizations of dataflow languages. We consider a class of schemes where the meaning of a node can be defined by a scheme, i.e. recursion is allowed. Data streams may flow through such a scheme in accordance with the dataflow relation defined in the scheme. Well-formed schemes allow only a downward dataflow unless iterations which include only one node. It is shown that every scheme can be transformed in a well-formed one. The main result concerns the power of iteration. It turns out that, on the basis of a certain class of stream functions, more functions can be implemented by recursion than without using iteration.

1 Introduction

Theoretical Computer Science, especially the theory of program schemes has been influenced strongly by russian scientists. Without to review the whole history, we only mention that A.P.Ershov was one of them who contributed a lot of ideas. One of the earliest papers in this direction is [EL67], a good overview is given in [E71] and [E77].

An aim of the theory of program schemes is to clarify the expressive power of several classes of schemes and to answer the question whether a scheme of a certain class can always be transformed into an "equivalent" scheme of another class.

We consider here the class of *dataflow schemes* which are formalizations of dataflow languages. A dataflow language allows to describe in a very natural way programs which can be carried out in parallel ([D74], [A82]). Schemes of parallel programs, especially dataflow schemes has been investigated in several papers, in particular in [K73], [DFL74]). A certain problem is to get a consistent semantics, in case recursion is allowed. This is due to the inherent nondeterministic behaviour of dataflow programs. If several data streams flow together into one "port" then these streams are merged nondeterministically which, for instance, causes nondeterminism.

In this paper, we consider dataflow schemes which consist, roughly spoken, of a set of *nodes* and a dataflow relation . Each node may possess some input *ports*, each of them is able to store a stream and works like a queue. The dataflow relation *into* describes the flow of data throughout the scheme. An element

$(n, (m, i))$ of *into* says that from node n to port i of node m data are transmitted. Such an element works like a channel but, the transmission of data does not take any time. The dataflow relation *into* is not restricted in any way.

The nodes of a system are labelled with two sorts of symbols: *function symbols* and *scheme symbols.*

A function symbol represents a *stream function*, which can be assigned by an *interpretation.* Such a stream function $f : \Sigma^n \to \wp(\Sigma \times \mathcal{F})$ accepts as input n-tuples of streams (words) from Σ and produces as output a set of pairs. Every element of the output consists of a stream and a stream function from $\mathcal{F}$ and is choosen nondeterministically. In this way, every node has a twofold internal state: the streams waiting at the input ports of the node for consuming and the current stream function of the node. This stream function may change at the moment when a certain input is consumed. The *meaning* of a scheme is a stream function too. Therefore, the meaning includes the internal state but not only the input-output relation between streams. This concept guarantees a consistent semantics and avoids anomalies. In [S89], stream functions were investigated in detail and it was shown in which way stream functions can be implemented by automata.

A scheme symbol represents a dataflow scheme again. The meaning of a node labelled with a scheme symbol is the meaning of the corresponding scheme. To overcome the problem of recursion, a mixture of operational and fixed-point semantics is used. For a given meaning of all scheme symbols and function symbols, the operational semantics describes a meaning of each scheme symbol again. So, on the basis of the meanings of function symbols, we get a function which transforms meanings of scheme symbols into meanings of scheme symbols. The least fixed point of this function is finally taken as the meaning of the whole dataflow scheme.

Since we did not restrict the dataflow relation *into*, the structure of a dataflow scheme may be very irregular. We show that every scheme can be transformed into a well-formed one where data flow only downward apart from loops which include exactly one node. The objective of this result is twofold. Firstly, we have an analogous result to the classical theory of program schemes which says that for every scheme there exists an equivalent scheme which is built only by sequences, alternatives and loops. Secondly, and more important here, by well-formed schemes we can easily show that iteration can be replaced by recursion. Of course, this is not a surprise.

However, the converse result does not hold. More exactly, there exists a class of functions such that only on the basis of these functions some functions can be implemented by recursion but not by nonrecursive schemes.

2 Dataflow Schemes

2.1 Basic Notions

A *signature* is a set F (of symbols) equipped with a function $ari : F \to I\!N$ which assigns to each symbol $f \in F$ an *arity* $ari(f) \in I\!N$ ($I\!N$ is the set of natural

numbers). For any sets, say A, B, the power A^B denotes the set of all functions $f : B \to A$. And, for a function $f : B \to A$, we may construct a function $f\left\langle {a \atop b} \right\rangle : B \to A$ with

$$f\left\langle {a \atop b} \right\rangle(x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise.} \end{cases}$$

If V is any set (of values) then $\Sigma := V^*$ denotes the set of all finite sequences (words) on V. We will also call an element $\alpha \in \Sigma$ a *stream*, ε denotes the empty stream. The concatenation of the streams α and β is simply denoted by $\alpha\beta$ and the usual prefix ordering is defined as $\alpha \sqsubseteq \beta$ iff there exists an γ such that $\alpha\gamma = \beta$. If $\alpha \sqsubseteq \beta$ then the *difference* $\beta - \alpha$ is defined as $\alpha(\beta - \alpha) = \beta$.

By Σ^n we denote the set of all n-tuples of streams. Sometimes, in order to distinct explicitely, we emphasize n-tuples of streams by an underline. If $[n] = \{1, \ldots, n\}$ then $\forall i \in [n] : \underline{\varepsilon}(i) = \varepsilon$. Concatenation, prefix-ordering, and difference can be extended on Σ^n.

By $\mathcal{F}_n$ we denote the set of all n-ary *stream functions* which is defined to be the largest set which fulfils the following three conditions:

1. $\mathcal{F}_n = \wp(\Sigma \times \mathcal{F}_n)^{\Sigma^n}$,
2. $\forall f \in \mathcal{F}_n : (\varepsilon, f) \in f(\underline{\varepsilon})$,
3. $\forall \underline{\alpha}, \underline{\beta} \in \Sigma^n :$
   $f(\underline{\alpha}) \neq \emptyset \wedge f(\underline{\alpha\beta}) = \{(\gamma\delta, h) \ : \ \exists g \in \mathcal{F}_n : (\gamma, g) \in f(\underline{\alpha}) \wedge (\delta, h) \in g(\underline{\beta})\}$.

By $\mathcal{F} := \bigcup\{\mathcal{F}_n \ : \ n \in I\!N\}$ we collect all the sets $\mathcal{F}_n$.

Every stream function $f \in \mathcal{F}_n$ induces a function $f^* : \Sigma^n \to \wp(\Sigma)$ by

$$f^*(\underline{\alpha}) := \{\beta \ : \ \exists g \in \mathcal{F}_n : (\beta, g) \in f(\underline{\alpha})\}.$$

A stream function, say f, is called *deterministic* if forall $\underline{\alpha} \in \Sigma^{ari(f)}$ the set $f^*(\underline{\alpha})$ is totally ordered wrt. the prefix ordering $\sqsubseteq$. We call two stream functions f, g *equivalent* if $f^* = g^*$.

If $f \in \mathcal{F}_n$ then we define $\langle f \rangle := \{g \ : \ \exists \underline{\alpha} \in \Sigma^n \exists \beta \in \Sigma : (\beta, g) \in f(\underline{\alpha})\}$.

An *embedding* of a stream function $f \in \mathcal{F}_n$ into a stream function $g \in \mathcal{F}_n$ is a function $emb : \langle f \rangle \to \langle g \rangle$ with

1. $emb(f) = g$, and
2. $\forall h, h' \in \langle f \rangle : (\beta, h') \in h(\underline{\alpha}) \Rightarrow (\beta, emb(h')) \in emb(h)(\underline{\alpha})$.

On the basis of embeddings we define a relation $\trianglelefteq$ on stream functions:

$$f \trianglelefteq g \quad \text{iff} \quad \text{there exists an embedding } emb : \langle f \rangle \to \langle g \rangle$$

The relation $\trianglelefteq$ is reflexive and transitive, but, in general, not an antisymmetric one, i.e. it is a quasi-order relation. If, however, $f \trianglelefteq g$ and $g \trianglelefteq f$ then $f^* = g^*$. Therefore, by factorization, we can make $\trianglelefteq$ into an order relation on the equivalence classes of stream functions. We identify a stream function with its equivalence class and use the symbol $\trianglelefteq$ for this order relation too. If $f \trianglelefteq g$ then we say that stream function f is *estimated* by g.

Let us consider any countable chain of stream functions $f_0 \trianglelefteq f_1 \trianglelefteq f_2 \ldots$ and let $emb_i : \langle f_i \rangle \to \langle f_{i+1} \rangle$ be an embedding between f_i and f_{i+1}. The relation $\bigcup \{ emb_i \ : \ i \in \mathbb{N} \}$ induces an equivalence on all stream functions $\bigcup \{ \langle f_i \rangle \ : \ i \in \mathbb{N} \}$. Let $[f]$ be the equivalence class of a stream function $f \in \langle f_i \rangle$ for a certain i. To each such equivalence class we assign a stream function by

$$[f](\underline{\alpha}) := \{ (\beta, [h]) \ : \ \exists g \in [f] : \ (\beta, h) \in g(\underline{\alpha}) \}.$$

By a straightforward proof one can show that this defines indeed a stream function. Moreover, this function is the least upper bound of our chain. That means: we got a cpo. The minimum is the function $nil = \{ (\underline{\varepsilon}, nil) \}$.

Additionally, we consider the following family of *generating functions* $gen_\alpha \in \mathcal{F}_0$ for all $\alpha \in \Sigma$ where

$$gen_\alpha(\emptyset) = \{ (\beta, gen_{\alpha - \beta}) \ : \ \beta \sqsubseteq \alpha \}.$$

It follows that $gen_\varepsilon(\emptyset) = \{ (\varepsilon, gen_\varepsilon) \}$. These functions are needed to define the semantics of dataflow schemes.

2.2 Syntax of Dataflow Schemes

Let $\mathbb{F}$ and $\mathbb{E}$ be two signatures. The elements of $\mathbb{F}$ are called *function symbols*, the elements of $\mathbb{E}$ *scheme symbols*.

A *dataflow scheme S over $\mathbb{F}, \mathbb{E}$* is defined as $S = (N, In, out, \lambda, into)$ where

- N is a finite set of "nodes".
- $In = \{ in_1, \ldots, in_k \} \subseteq N$ is a finite set of "input nodes", k is called the *arity* of the scheme S, and we set $ari(S) := k$.
- $\lambda : (N - In) \to (\mathbb{F} \cup \mathbb{E})$ is a labelling function which assigns to each node, except the input nodes, an element from $\mathbb{F}$ or $\mathbb{E}$. We set

$$ari(n) := \begin{cases} ari(\lambda(n)) & \text{if} \quad n \in N - In \\ 0 & \text{if} \quad n \in In \end{cases}$$

  and build a set $P := \{ (n, i) \ : \ n \in N \wedge i \in [ari(n)] \} \cup \{ out \}$. An element $(n, i) \in P$ is called a *port*, input nodes do not have any ports.
- $into \subseteq N \times P$ is the so-called dataflow relation.

A *dataflow system $\mathcal{S}$* over $\mathbb{F}, \mathbb{E}$ is a function which assigns to each scheme symbol $A \in \mathbb{E}$ a dataflow scheme $\mathcal{S}_A = (N_A, In_A, out_A, \lambda_A, into_A)$ such that $ari(A) = ari(\mathcal{S}_A)$.

If there is no danger of misunderstanding then we identify the scheme $\mathcal{S}_A$ with the scheme symbol A.

A dataflow system induces a relation

$$uses \subseteq \mathbb{E} \times \mathbb{E} \ \text{ by } \ uses := \{ (A, B) \ : \ \exists n \in N_A : \lambda_A(n) = B \}.$$

$uses^+$ denotes the transitive closure of $uses$.

2.3 Semantics

Let us consider any dataflow scheme $(N, In, out, \lambda, into)$, where P is the set of ports. We introduce three auxiliary operations.

For $\xi \in \Sigma^P$ and $n \in N$, it is $(\xi \downarrow n) \in \Sigma^{ari(n)}$ with

$$(\xi \downarrow n)(i) := \xi(n, i) \quad \text{for all } i \in [ari(n)].$$

$\xi \downarrow n$ describes the $ari(n)$-tuple of streams at the ports of node n. If $ari(n) = 0$, i.e. node n does not have any port, then $\Sigma^0 = \{\emptyset\}$ and, therefore, $\xi \downarrow n = \emptyset$.
Next, we define an operation witch describes the flow of streams which are produced by the nodes of a scheme. Since several streams may flow into one port, these streams are merged nondeterministically. This merge operation is denoted by $\| : \wp(\Sigma) \to \wp(\Sigma)$ and defined as

$$\|\emptyset := \{\varepsilon\}, \quad \|(\mathcal{A} \cup \{\varepsilon\}) := \|\mathcal{A},$$

$$\|\mathcal{A} := \bigcup \{a \circ \|((\mathcal{A} - \{a\alpha\}) \cup \{\alpha\}) \ : \ a\alpha \in \mathcal{A}\} \text{ where } a \in V, \alpha \in \Sigma.$$

For $\pi \in \Sigma^N$, it is $\pi \Uparrow \subseteq \Sigma^P$ where

$$\pi \Uparrow := \{\xi \ : \ \xi(p) \in \|\{\pi(n) \ : \ (n, p) \in into\}\}.$$

Finally, if $\underline{\alpha} \in \Sigma^k$ (k is the number of input nodes of the considered scheme), $\eta \in \mathcal{F}^N$ then $\eta \oplus \underline{\alpha} \in \mathcal{F}^N$ where

$$(\eta \oplus \underline{\alpha})(n) := \begin{cases} gen_{\gamma\gamma'} & \text{if} \quad n = in_j, \ \eta(n) = gen_\gamma, \ \gamma' = \underline{\alpha}(j) \\ \eta(n) & \text{otherwise} \end{cases}$$

$\eta \oplus \underline{\alpha}$ describes the transmission of the n-tuple $\underline{\alpha}$ of streams to the input nodes of the scheme which are then able to generate these streams.

A *configuration* is a pair $(\xi, \eta) \in \Sigma^P \times \mathcal{F}^N$ such that $\forall n \in N : \eta(n) \in \mathcal{F}_{ari(n)}$, and $\forall n \in In : \eta(n) = gen_\alpha$ for some stream $\alpha \in \Sigma$. It characterizes the internal state of a scheme.
A *transition* describes the stepwise transformation of configurations:

$$(\xi, \eta) \to (\xi', \eta')$$

if and only if

$$\exists \underline{\alpha}, \underline{\beta} \in \Sigma^P \ \exists \pi \in \Sigma^N \ \forall n \in N :$$
$$\underline{\alpha} \sqsubseteq \xi \ \wedge \ (\pi(n), \eta'(n)) \in \eta(n)(\underline{\alpha} \downarrow n) \ \wedge \ \underline{\beta} \in (\pi \Uparrow) \ \wedge \ \xi' = (\xi - \underline{\alpha})\underline{\beta}.$$

In this definition, $\underline{\alpha}(p)$ is a certain prefix of the stream waiting at port p for consuming, $\eta(n)$ is the current meaning (stream function) assigned to node n, $\pi(n)$ is the stream produced by node n if $\underline{\alpha} \downarrow n$ is consumed, $\eta'(n)$ is the new meaning of node n after this action is curried out, and $\underline{\beta}$ describes one possible result after flowing and merging all produced streams into the ports in accordance with the dataflow relation *into*.

The reflexive and transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

Let A be any scheme of arity k. For the configuration (ξ, η) we define $[\![A, \xi, \eta]\!] \in \mathcal{F}_k$, the *meaning of A under* (ξ, η), as

$$[\![A, \xi, \eta]\!](\underline{\alpha}) := \{(\beta, [\![A, \xi' - \underline{\varepsilon}\left\langle \begin{matrix} out \\ \beta \end{matrix} \right\rangle, \eta']\!]) \ : \ (\xi, \eta \oplus \underline{\alpha}) \rightarrow^* (\xi', \eta') \ \wedge \ \beta \sqsubseteq \xi'(out)\}.$$

Here, the stream β is assumed to be produced by the stream function $[\![A, \xi, \eta]\!]$ if $\underline{\alpha}$ is consumed. The stream $\xi'(out)$ at the output port is reduced by the produced stream β in order to get the final configuration.

By an *interpretation* I we mean a function $I \ : \ \mathbb{F} \cup \mathbb{E} \ \rightarrow \ \mathcal{F}$ with $\forall s \in (\mathbb{F} \cup \mathbb{E}) \ : \ \ I(s) \in \mathcal{F}_{ari(s)}$. We divide an interpretation I into the two parts $I_F : \mathbb{F} \rightarrow \mathcal{F}$ and $I_E : \mathbb{E} \rightarrow \mathcal{F}$ such that $I = I_F \cup I_E$. The meaning of a scheme A under the interpretation I is defined as

$$[\![A, I]\!] := [\![A, \underline{\varepsilon}, \eta_I]\!] \quad \text{with} \quad \eta_I(n) := \begin{cases} I(\lambda(n)) & \text{if} \quad n \in (N - In) \\ gen_\varepsilon & \text{if} \quad n \in In \end{cases}$$

If the whole interpretation I is given then the meaning $[\![A, I]\!]$ is defined in an operational way. However, in general, only the part I_F of an interpretation is known. The part I_E is then defined by a fixed point method.

For a given I_F, we define a function $\mathbf{F}_{I_F} : \mathcal{F}^{\mathbb{E}} \rightarrow \mathcal{F}^{\mathbb{E}}$ by

$$\mathbf{F}_{I_F}(I_E)(A) := [\![A, I]\!].$$

Lemma 1. *Function $\mathbf{F}_{I_F}$ is continuous.*

Proof: is straightforward and omitted.

By this lemma we know, that the least fixed point, $fix\, \mathbf{F}_{I_F}$, exists (and equals to the least upper bound of an infinite iteration of $\mathbf{F}_{I_F}$). Therefore, we can define the meaning $[\![\mathcal{S}]\!] \in \mathcal{F}^{\mathbb{E}}$ of a dataflow system $\mathcal{S}$ as

$$[\![\mathcal{S}]\!] \ := \ fix\, \mathbf{F}_{I_F}.$$

3 Well-formed Schemes

If $into_A$ is the dataflow relation of any given scheme A then we build the relation $into_A_n \subseteq N_A \times N_A$ by

$$into_A_n := \{(m, n) \ : \ \exists i : (m, (n, i)) \in into_A\}$$

A scheme A is called *well-formed* if $into_A_n^*$ is an antisymmetric relation, i.e. if $(m, n) \in into_A_n^*$ and $(n, m) \in into_A_n^*$ then $m = n$. A dataflow system $\mathcal{S}$ is well-formed if each of its schemes is well-formed.

In the following, we show that each dataflow system can be transformed into a

well-formed one. To do this we introduce an operation *condense* which takes a certain set M of nodes of a scheme A and build one scheme of it. As a auxiliary set we build

$$N_M := \{n \ : \ n \in N_A - M \wedge \exists m \in M : (n,m) \in into_A_n\}$$

and we consider any bijection

$$\varphi : N_M \to [l].$$

Here, $l \in I\!N$ is the cardinality of set N_M.

If $\mathcal{S}$ is a dataflow system over $I\!F$ and $I\!E$, $\mathcal{S}_A = (N_A, In_A, out_A, \lambda_A, into_A)$ is any scheme of $\mathcal{S}$ and $M \subseteq N_A - In_A$ is a set of nodes, $c \in M$ is a special node of M then $\mathcal{S}' = \mathcal{S}\langle A, M, c]$ is the condensed system over $I\!F, I\!E'$ where

- $I\!E' = I\!E \cup \{C\}$ with $C \notin I\!E$,
  $\mathcal{S}'_A = (N_{A'}, In_{A'}, out_{A'}, \lambda_{A'}, into_{A'})$, $\mathcal{S}'_C = (N_C, In_C, out_C, \lambda_C, into_C)$,
- $N_{A'} := N_A \cup In_A,\ In_{A'} := In_A$,
- $ari(c) := ari(C) := l + 1$,
- $\lambda_{A'}(n) := \begin{cases} C & \text{if } n = c \\ \lambda_A(n) & \text{otherwise} \end{cases}$,
- $\begin{aligned} into_{A'} := &\{(n,(m,i)) \ : \ (n,(m,i)) \in into_A \wedge m \neq c\} \quad \cup \\ &\{(n,(c,\varphi(n))) \ : \ n \in N_M\} \quad \cup \\ &\{(c,(c,l+1))\} \end{aligned}$,
- $N_C := M \cup In_C,\ In_C := \{i_1, \ldots, i_{l+1}\},\ \lambda_C := \lambda_A$,
- $\begin{aligned} into_C := &\{(n,(m,i)) \ : \ (n,(m,i)) \in into_A \wedge n,m \in M \wedge n \neq c\} \quad \cup \\ &\{(in_{\varphi(n)},(m,i)) \ : \ (n,(m,i)) \in into_A \wedge n \in N_M \wedge m \in M\} \quad \cup \\ &\{(in_{l+1},(m,i)) \ : \ (c,(m,i)) \in into_A \wedge m \in M\} \quad \cup \\ &\{(c, out)\}. \end{aligned}$.

Lemma 2. *If* $\mathcal{S}' = \mathcal{S}\langle A, M, c]$ *then* $\forall B \in I\!E : [\![\mathcal{S}]\!](B) \trianglelefteq [\![\mathcal{S}']\!](B)$.

Proof: The proof is very technical. One has to show that ever the node c of scheme A is involved in a computation of the original system this can be done analogously by involving the scheme C of the condensed system.This holds for any given interpretation I and therefore, it holds for the least fixed point too.

Due to node splitting, the condense operation generates a more general scheme than the original one. However, if the meanings of all nodes are deterministic stream functions, and there is no merge of data streams, i.e. the implication $(n,p) \in into \wedge (m,p) \in into \Rightarrow m = n$ holds - we call such a dataflow system deterministic too - then we get a stronger version of the last lemma.

Lemma 3. *If* $\mathcal{S}$ *is a deterministic dataflow system and* $\mathcal{S}' = \mathcal{S}\langle A, M, c]$ *then* $\forall B \in I\!E : [\![\mathcal{S}]\!](B) = [\![\mathcal{S}']\!](B)$.

Theorem 4. *Any dataflow system can be transformed into a well-formed one. The new system estimates the original system, and if the original system is determistic then both are equivalent.*

Sketch of the **Proof:**
Let $\mathcal{S}$ be any dataflow system over $\mathbb{F}, \mathbb{E}$. If all $A \in \mathbb{E}$ are well-formed then we are done. Otherwise we take any A which is not a well-formed scheme. Then, in A there exists a node c such that the set

$$M_c := \{n \ : \ (c, n) \in \mathit{into}_A_n^* \wedge (n, c) \in \mathit{into}_A_n^*\}$$

has at least two elements. We build $\mathcal{S}' = \mathcal{S}\langle A, M_c, c]$. In the new scheme A as well as in C, the sets M_c are singletons now. Therefore, after a finite number of condensations we will get a well-formed system with the same meaning as the original system. □

4 Iteration and Recursion

A dataflow system over $\mathbb{F}, \mathbb{E}$ is called *non-recursive* if the relation uses^+ is irreflexive and it is called *non-iterative* if for all $A \in \mathbb{E}$ the relation $\mathit{into}_A_n^+$ is irreflexive too.

At first, we show that iteration can be replaced, in a certain sense, by recursion.

Theorem 5. *For each dataflow system $\mathcal{S}$ there exists a non-iterative system $\mathcal{S}'$ which estimates the system $\mathcal{S}$.*

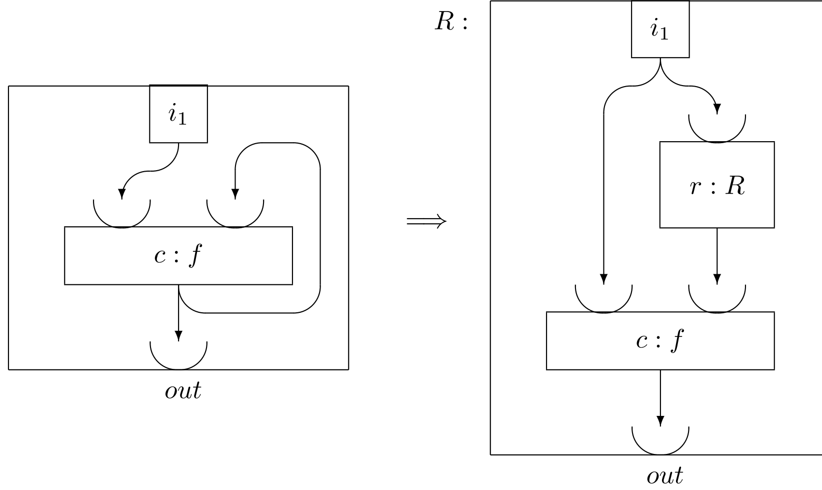
Sketch of the **Proof:**
In accordance with theorem 4 we transform the system $\mathcal{S}$ into a well-formed one. Let us consider any scheme A and node $c \in N_A$ with $(c, c) \in \mathit{into}_A_n$. The arity of c should be $l + 1$, i.e. $(c, (c, l + 1)) \in \mathit{into}_A$. Corresponding to the condense operation, there does not exist any dataflow $(n, (c, l + 1))$ apart from $n = c$. Therefore, we transform scheme A into A' such that only the following changes are done:

- $\lambda_{A'}(c) := R$ where R is a new scheme symbol of arity l,
- $\mathit{into}_{A'} := \mathit{into}_A - \{(c, (c, l + 1))\}$.

The new scheme R is built in the following way:

- $N_R := \{c, r\} \cup In_R,\ In_R := \{in_1, \ldots, in_l\}$,
- $\lambda_R(c) := \lambda_A(c),\ \lambda_R(r) := R$,
- $\mathit{into}_R := \{(in_j, (c, j)) \ : \ 1 \le j \le l\} \cup$
  $\{(in_j, (r, j)) \ : \ 1 \le j \le l\} \cup \{(r, (c, l + 1)), (c, \mathit{out})\}$.

The new scheme estimates to the old one. □



The used condense operations are responsible that the new system properly estimates the old one. If, the original system is already a well-formed system then we do not need any condense operation. Or, if the old system is deterministic then the condense operation generates equivalent systems. Therefore, we get:

Corollary 6. *For every determininistic or well-formed scheme S, an equivalent non-iterative system exists.*

However, the replacement of recursion by iteration is impossible. In order to show this we consider a special class of stream functions.

A stream function f of arity n is called *bounded* if n monotonic functions $\varphi_i : \mathbb{N} \to \mathbb{N}$ exist such that

$$\forall \underline{\alpha}, \beta : \beta \in f^*(\underline{\alpha}) \longrightarrow \\ \Big(\exists \underline{\alpha}' \forall i \in [n] : \underline{\alpha}' \sqsubseteq \underline{\alpha} \wedge \beta \in f^*(\underline{\alpha}') \wedge length(\underline{\alpha}'(i)) \leq \varphi_i(length(\beta))\Big).$$

A bounded stream function allows to estimate the length of the needed input of every port in order to produce an output of a certain length.

A stream function f is called *undelayed* if

$$\forall \underline{\alpha} \in \Sigma^{ari(f)} : \ \{\gamma \ : \ \exists \beta : \beta \in f^*(\underline{\alpha}) \wedge \gamma \sqsubseteq \beta\} \subseteq f^*(\underline{\alpha}).$$

For an undelayed function, every prefix of an output can be produced as output too.

A bounded and undelayed function is called a *bu*-stream function.

Theorem 7. *If $\mathcal{S}$ is a non-recursive system and I is an interpretation which uses only bu-functions then the meaning of all schemes of $\mathcal{S}$ is also a bu-function.*

Sketch of the **Proof:**
We consider any system $\mathcal{S}$ and transform it into a well-formed system. It is not hard to see that the meaning of every well-formed scheme is a bu-function. $\square$

There are simple functions which are not bu-functions, e.g. the invertation of a part of a stream which needs the whole part in order to produce an output of length 1. Let us assume that ! is our value which separates a first part of a stream. This behaviour of invertation is described by the following stream functions inv_α and nil_α for $\alpha \in \Sigma$:

$$\begin{array}{rl} ! \notin \alpha\alpha' \Rightarrow inv_\alpha(\alpha') & := \{(\varepsilon, inv_{\alpha\alpha'})\} \\ ! \notin \alpha\alpha' \Rightarrow inv_\alpha(\alpha'!\beta) & := \{(\gamma, nil_\delta) \ : \ \gamma\delta = \alpha^{-1}!\} \\ nil_\alpha(\beta) & := \{(\gamma, nil_\delta) \ : \ \gamma\delta = \alpha\} \end{array}$$

Here, functions nil_α play the role to consume the suffix of the stream after the exclamation mark.

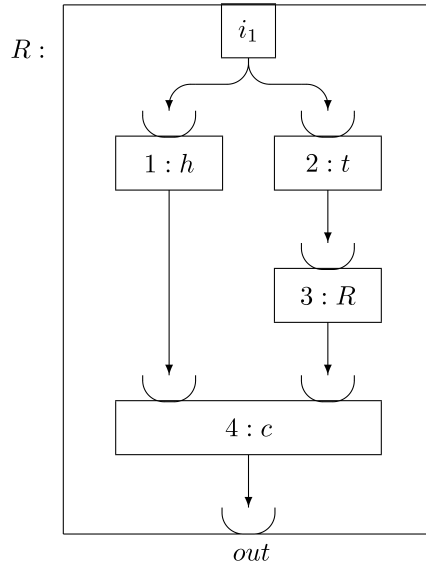
Clearly, inv is not a bu-function! Now, we introduce some bu-functions in order to implement inv. Since always $(\varepsilon, f) \in f(\underline{\varepsilon})$, we omit this in the following definitions.

$$\begin{array}{rll} & head(a\alpha) & := \{(\varepsilon, head_a), (a, nil_\varepsilon)\} \\ & head_a(\alpha) & := \{(\varepsilon, head_a), (a, nil_\varepsilon)\} \\ & tail(!\alpha) & := \{(\varepsilon, nil_\varepsilon)\} \\ a \neq ! & \Rightarrow tail(a\alpha) & := \{(\gamma, id_\delta) \ : \ \gamma\delta = \alpha\} \\ & id_\alpha(\beta) & := \{(\gamma, id_\delta) \ : \ \gamma\delta = \alpha\beta\} \\ & conc_{(\varepsilon,\alpha)}(\varepsilon, \beta) & := \{(\varepsilon, conc_{(\varepsilon,\alpha\beta)})\} \\ & conc_{(\varepsilon,\alpha)}(!, \beta) & := \{(\varepsilon, conc_!), (!, bnil_\varepsilon)\} \\ ! \notin a\alpha\gamma & \Rightarrow conc_{(\varepsilon,\alpha)}(a\beta, \gamma) & := \{(\delta, conc_{(a,\chi)}) \ : \ \delta\chi = \alpha\gamma\} \\ \alpha\gamma = \psi!\sigma \wedge ! \notin \psi & \Rightarrow conc_{(\varepsilon,\alpha)}(a\beta, \gamma) & := \{(\delta, bnil_\chi) \ : \ \delta\chi = \psi a!\} \\ & conc_!(\alpha, \beta) & := \{(\varepsilon, conc_!), (!, bnil_\varepsilon)\} \\ & bnil_\alpha(\beta, \gamma) & := \{(\delta, bnil_\chi) \ : \ \delta\chi = \alpha\} \\ ! \notin \gamma & \Rightarrow conc_{(a,\alpha)}(\beta, \gamma) & := \{(\delta, conc_{(a,\chi)}) \ : \ \delta\chi = \alpha\gamma\} \\ \alpha\gamma = \psi!\sigma \wedge ! \notin \psi & \Rightarrow conc_{(a,\alpha)}(\beta, \gamma) & := \{(\delta, bnil_\chi) \ : \ \delta\chi = \psi a!\} \end{array}$$

The implementation of inv is now given by the scheme R over $\mathbb{F} = \{h, t, c\}$, $\mathbb{E} = \{R\}$ with

$$\begin{array}{c} N_R := \{1, 2, 3, 4\} \cup In_R, \ In_R := \{i_1\}, \ \lambda_R := \{(1, h), (2, t), (3, R), (4, c)\}, \\ into_R := \{(i_1, (1, 1)), (i_1, (2, 1)), (1, (4, 1)), (2, (3, 1)), (3, (4, 2)), (4, out)\} \end{array}$$

and the interpretation $I_F := \{(h, head), (t, tail), (c, conc_{(\varepsilon,\varepsilon)})\}$.



It follows:

Theorem 8. *There are stream functions which can be implemented by non-iterative dataflow schemes on the basis of bu-functions but not by non-recursive schemes.*

5 Conclusions

There is a similiar result to theorem 8 within the classical theory of program schemes: if any program scheme is given, it is in general impossible to construct an equivalent non-recursive one. But, there is an important difference. Namely, every partial-recursive function (in other words: computable function) can be implemented without recursion on the basis of the functions: assignment of zero, increment of one, decrement of one, and the test whether a certain variable has zero as value. It is not hard to show that every partial recursive function can be implemented by recursive dataflow schemes on the basis of bu-stream functions. A certain natural number n can, for instance, be represented as a stream of n digits "1" closed by a "0". But, in accordance with our last theorem, in such a way every partial-recursive function cannot be implemented by nonrecursive schemes.

There are a lot of open problems connected, in particular, with the practical use of recursive dataflow languages. Some of them are mentioned in [A82], [A86]. Another important problem is the estimation of the needed resources. The input ports are considered to be resources of infinite capacity. Of course, they must be restricted in implementations. Therefore, it is necessery to estimate the needed length of the data-streams in order to schedule the storage. A first approach to this problem is presented in [L91].

Finally, we believe that dataflow languages are suitable to build programs which can be carried out in parallel. The big advantage is, that the programmer can concentrate on the inherent data dependencies between computations and leave the other synchronisations to the compiler. This causes research to clarify the needed methods for compilation.

References

[A82] Ackerman,W.B.: Data Flow Languages. Computer 15,2, 1982.

[A86] Ashcroft,E.A.: Dataflow and Education: Data-driven and Demand-driven Distributed Computation. LNCS 224, 1-50, Springer, Berlin, 1986.

[D74] Dennis,J.B.: First version of a Data Flow Procedure Language. LNCS 19, 362-376, Springer, Berlin, 1974.

[DFL74] Dennis,J.B., Fosseen, J.B., Linderman, J.P.: Data Flow Schemes. LNCS 19, Springer, Berlin, 1974.

[EL67] Ershov,A.P., Ljapunov, A.A.: On the formalisation of the program notion. Kibernetika, Kiev, 1967.

[E71] Ershov,A.P.: Theory of program schemata. IFIP Congress 1971, North Holland Publishing Company, Amsterdam, 1971.

[E77] Ershov,A.P.: Introduction in the theory of programming(in Russian). Nauka, Moskow, 1977.

[G75] Greibach,S.A.: Theory of Program Structures: Schemes, Semantics, Verification. LNCS 36, Springer, Berlin, 1975.

[K73] Keller,R.M.: Parallel program schemata and maximal parallelism. Journal of ACM, 20,1973, 514-537, 696-710.

[K80] Kotov,V.E.: On Basic Parallel Language. IFIP Congress 1980, North Holland Publishing Company, Amsterdam, 1980.

[L91] Lau,S.: Ansätze und Methoden zur Ressourcenabschätzung in Datenflußsystemen. Ph.D. Thesis, Dresden University of Technology, 1991.

[S89] Son,T.C.: Realisierungen nichtdeterministischer Wortfunktionen. Master Thesis, Dresden University of Technology, 1989.

This article was processed using the LaTeX macro package with LLNCS style