A THEORY FOR NONDETERMINISM, PARALLELISM, COMMUNICATION, AND CONCURRENCY *

Manfred BROY


Communicated by M. Nivat
Received January 1984
Revised January 1986

Abstract. An applicative language is introduced for representing concurrent programs and communicating systems in the form of mutually recursive systems of nondeterministic equations for functions and streams. Mathematical semantics is defined by associating particular fixed points with such systems. These fixed points are chosen using a combination of several complete partial orderings. Operational semantics is described in the form of term rewriting rules, consistent with the mathematical semantics. It represents data-driven reduction semantics for usual expressions and data-driven data flow semantics in the case of recursive stream equations. So the language allows to treat the basic semantic notions of nondeterminism, parallelism, communication, and concurrency for multiprogramming in a completely formal, applicative framework. In particular, it provides a semantic theory for networks of loosely coupled, nondeterministic, communicating, stream processing functions. Finally, the relationship of the presented language to partial recursive functions and nonconventional computational models such as data flow and reduction machines is shown.

Contents

1. Introduction ................................................................. 2
2. The language .................................................................. 6
   2.1. The domain .......................................................... 7
   2.2. Syntax of the language AMPL ................................. 8
   2.3. Examples ............................................................. 9
      2.3.1. The ambiguity operator .................................. 10
      2.3.2. Merging ......................................................... 11
3. Mathematical semantics .................................................. 12
   3.1. Basic definitions ..................................................... 12
   3.2. Powerdomains revisited .......................................... 13
   3.3. Mathematical semantics of recursion-free expressions 16
4.4. Nondeterminate functions as fixed points ..................... 17
   4.4.1. Semantics of recursive systems .......................... 17
   4.4.2. Properties of the semantic definitions ................ 19
   4.4.3 A short discussion of the ambiguity operator ......... 21
   3.5. Streams as fixed points of systems of nondeterministic equations .......................... 22
   3.6. Semantics of programs .......................................... 27
4. Operational semantics .................................................... 27

* This work was partially sponsored by Sonderforschungsbereich 49, Programmiertechnik, Munich.
1. Introduction

The last two decades of computer science are characterized by an enormous progress in the formal foundations of programming languages and their semantics. In sequential, deterministic programming most of the remaining questions are of quantitative (questions of programming in the large) rather than of qualitative nature. In the field of nondeterministic, parallel, communicating, concurrent programs the state of the art is less satisfactory. Although drastic efforts have been undertaken to investigate this field leading to a considerable amount of knowledge, we are far from having extensive, widely accepted theories for concurrent programming. Most existing theories do not cover all important aspects and/or they are too complex and complicated.

Nevertheless, numerous papers have been published which suggest language constructs for concurrent programming. These papers have had a considerable impact on the field of concurrent programming, and in many cases helped in developing a better understanding. However, the lack of proper formal definitions of the semantics of such languages must be considered as a severe drawback. On the one hand, it seems impossible to enlarge on a programming methodology for the construction of concurrent software without having well-explored theoretical foundations. On the other hand, a properly designed programming language presumes a complete understanding of the underlying concepts, which is also impossible without having a formal theory. And last but not least, mathematical foundations are an indispensable requirement for teaching concurrent programming. Therefore, I strongly believe that it is necessary to investigate the concepts of concurrent programming in a joint consideration of both mathematical (denotational) semantics and its corresponding operational semantics. Here, fixed-point theory seems to be the most adequate framework.

So in the sequel we shall try to develop a strictly fixed-point oriented approach to the semantics of applicative multiprogramming. It should be noted that the restriction to applicative languages is not a profound one. It only helps in concentrating on the central issues. The approach is based on a simple nondeterministic
A theory for nondeterminism, parallelism, communication, and concurrency

programming language. For this language both mathematical and operational semantics are given. The operational semantics consists of a set of computation rules which model the behaviour of a simple reduction machine. Based on a thorough discussion, the language is stepwise extended to allow more general patterns of communication leading to systems of communicating expressions.

Before going deeper into the theory of applicative multiprogramming, it seems useful to recall some of the most important notions, namely: nondeterminism, parallelism, communication, and concurrency.

Considering these notions isolated from each other (as far as this is possible) already causes some problems. However, combining these notions into the concepts of one language multiplies the difficulties. For example there are different concepts (cf. [55, 23]) which, taken for themselves, can be treated quite satisfactorily. In concurrent programming, however, some of these concepts are used side by side. For instance, the scheduling of simple communication actions may be mapped onto straightforward-choice nondeterminism ('erratic' nondeterminism), while disjunctive ('multiple') waiting has to be mapped onto some kind of nondeterminism which delays the choice until one of the possibilities yields a defined way of resuming (local 'angelic' nondeterminism).

So nondeterministic, parallel, communicating, concurrent systems raise a number of severe theoretical and practical questions which have to be answered before a proper methodology for the construction of concurrent software and distributed hardware systems can be envisaged. In particular, a number of key questions has to be tackled whose solutions may give proper formal foundations for software and hardware architectures for 'computers of the fifth generation' (cf. [50]). In the sequel, a brief overview will be given on some of these questions and solutions suggested in this paper.

The domain problem: Multiprograms abstract from time and schedulers and hence, describe rather a (possibly infinite) class of (determinate) programs than one particular program. This is modelled by introducing nondeterminism. The combinatorial complexity of the class of programs described by a multiprogram makes it practically impossible to reason about the single courses of computation separately. So one tries to consider a multiprogram as a unit and to reason about it in a way such that the results hold for all feasible courses of computation. Hence, one fixed point is associated with a multiprogram, rather than a set of fixed points, containing all feasible courses of computation. However, this way of proceeding bears the risk of unwanted identifications and confusions between operationally separated courses of computations, especially, if nonflat domains including finite and infinite elements have to be considered in connection with communications. If the possible sequences of communicated values are taken as defining the meaning of a process, then, obviously, a process cannot be considered as a nondeterministic function over flat domains. However, on nonflat domains the classical powerdomain construction does not work (cf. [75, 79]), because there even the 'Egli-Milner ordering' represents only a quasi-ordering.
Nevertheless, the semantics of a nondeterministic computation over a nonflat domain can be precisely described by a fixed point. The central problem is how to characterize this fixed point. In contrast to classical fixed-point theory, where least (in one partial ordering) fixed points are taken, this problem is solved by combining two basic orderings: the Egli-Milner ordering, modelling the progress of the computation, and set inclusion, modelling the progress of decisions, the choices. A specific technique has to be applied for the ambiguity operator which is not monotonic in the Egli-Milner ordering and in some sense represents the ultimate nonsequential function (cf. [17, 18, 19]).

*Systems of expressions communicating by streams:* In a system of communicating agents, communication can be modelled by considering sequences (‘streams’) of communication actions. Formally, such a system of communicating expressions is defined by a system of mutually recursive fixed-point equations for streams with the respective stream processing functions on the right-hand side. In order not to obtain always trivially the least element undefined as fixed points of such systems, one has to use nonstrict constructor functions for streams and additionally specific computation rules. This includes infinite objects in a straightforward manner.

For solving the so-called ‘merge anomaly’ in cases of nondeterministic systems, which is the result of confusing different well-separated courses of computations, nondeterministic recursive equations for streams have to be considered as sets of deterministic equations for streams rather than as equations for sets of streams.

Accordingly, a system of communicating expressions can be considered as a purely applicative description of a network (a directed graph) of stream-processing functions (in the nodes) and streams of communications between them (as the arcs).

*Decision systems:* As already mentioned, a system of concurrent, communicating agents is generally nondeterministic for modelling the different options of executions, in particular, abstracting from concrete time and schedulers. So the evaluation of the respective nondeterministic programs generally requires decisions to obtain one concrete computation. This leads to the important problem at which particular ‘situation’ a decision is taken. Since time is replaced by ‘causality flow’, which is formally represented by the approximation principle in fixed-point theory, this is equivalent to the question how good the approximations for the input have to be for carrying enough information to take a decision consistently. So appropriate choice operators have to be selected very carefully.

In order to model a really distributed system, all these decisions should be made locally (i.e., within one agent) without regarding the global state of the systems or any of the states of other agents. This requirement is fulfilled by the nondeterministic systems of communicating expressions.

In such systems decisions are taken in time (*consistently*, modelled by the sufficiently good approximations) and place (*locally*, modelled by the ‘context-independence’ of decisions).

*Concurrency:* In order to introduce real concurrency of competing computation: into applicative languages, McCarthy's ambiguity operator is included. Such an operator is necessary for defining a nonstrict, nonsequential merge function fo
streams such that \( \text{merge}(\bot, s) = \text{merge}(s, \bot) = s \) and

\[
\text{merge}(x_1 \& s_1, x_2 \& s_2) = ((x_1 \& \text{merge}(s_1, x_2 \& s_2)) \sqcap (x_2 \& \text{merge}(x_1 \& s_1, s_2)))
\]

which is an inevitable prerequisite for many networks of communicating agents.

Thus, concurrency does not only require a free straightforward choice between concurrent computations, but it requires a choice depending on particular termination properties of the concurrent computations. This brings all the problems of unbounded nondeterminism, its noncontinuity as found in the fairness discussion (cf. [74, 17, 1]), and even worse problems concerning monotonicity. For solving these problems a fixed point for recursive equations containing the ambiguity operator is again characterized by combining two partial orderings. In the first step only an approximation (in the sense of partial correctness) is defined, and based on these approximations the precise semantics is given specifying a second fixed point (based on inclusion ordering) as a subset of the approximation.

**Computability:** One of the most important questions when switching from 'sequential' programming to multiprogramming concerns the expressive power: Can we define certain functions by multiprograms which cannot be expressed by sequential programs? Or more specific: Are there functions which are not partially recursive, but are associated with a concurrent system? This question is not only of theoretical interest, but also of high practical importance since it helps to answer the question whether the methods of specification, verification, and modelling for sequential programs may suffice also for concurrent programs.

The switch to concurrent communicating programs does not only include a necessity to consider general nonstrict functions and even nonsequential functions (in the technical sense of [84, p. 55]), which all can be mapped into the domain of partial recursive functions, but it also requires the consideration of the aforementioned ambiguity operator leading to unbounded nondeterminism and hence, to functions where the sets of arguments for which nonterminating computations exist are \( \Sigma_1 \)-complete.

**Computational models:** It is the very nature of notions like 'parallelism' that they not only correspond to abstract functional (input/output-)behaviours of program systems, but also characterize how a program is evaluated. So a close relationship to operational semantics is to be established. In this paper an operational semantics is defined in form of computation rules (term rewriting rules). In addition to classical operational semantics the following five aspects are of major importance when dealing with multiprograms:

1. Nondeterminism implies *nonconfluent term rewriting systems* (cf. 'decision systems' and 'concurrency' above).
2. Evaluations of communicating processes generally have to start *before* all information about the input is available; so one has to cope with computations with incomplete information leading to the concept of *partial evaluation* or *mixed computation.*
A communicating process consumes its input piecewise and produces its output piecewise. So respective computation rules for communication have to be used.

A communicating process that does not terminate may produce an infinite stream of output and thus compute an infinite object. So techniques of lazy or enforced evaluation have to be used.

The possibilities of inherent parallelism and of compulsory parallelism have to be expressed by parallel evaluation rules.

In a function application all actual arguments can be evaluated in parallel and even the body expression of the function can be partially evaluated by mixed computation techniques in parallel. The basic idea is to split the substitution step for an application of an n-ary function. Conventionally, the function identifier is substituted by the respective expressions, and all formal parameters are replaced by the respective actual parameter in one indivisible action. This action can be split into up to n + 1 separated substitution ('communication') steps such that the arguments can be computed and substituted independently. Thus a data-driven reduction semantics is defined.

Reduction, data flow, and networks of distributed agents: In contrast to the very general concept of data-driven reduction and data flow the classical Von-Neumann computer architecture is essentially based on sequential control. This is why all attempts to extend it to an architecture for parallel computations lead to extremely complicated hardware and software structures. So people try to suggest non-Von-Neumann architectures such as functional machines, data flow machines, reduction machines, cellular processors, reconfigurable ('programmable') hardware structures, etc.

However, to overcome the basic problems of Von-Neumann machines, such innovative architectures should be based on a proper theory (concerning their logical structure, not their physical representation) which can also be taken as the basis for a software engineering discipline including the specification, development, and verification of software for such systems. It will be briefly discussed and outlined how the language defined in this paper can be taken as a step towards formal foundations for such concepts. In particular a formal definition for the semantics of a data flow language will be given. So data flow graphs can be specified in terms of the given language for applicative multiprogramming. Such graphs can be used to represent networks of communicating agents or machines as well as integrated switching circuits and even machine architectures. The Von-Neumann concept of a sequential stored program architectures appears just as an extreme case.

2. The language

In this section the domain of finite and infinite streams is introduced on which a simple first-order language is based that allows the recursive definition of nondeterministic functions and streams.
2.1. The domain

For the notions complete partial order, algebraic domain, monotonic and continuous, see e.g. [79] or Section 3.

As an important example for a nonflat, algebraic domain for multiprogramming the domain of streams is considered (cf. [61, 28, 33]).

Let $A^\perp$ be a countable, flat domain. Then the domain $\text{STREAM}(A)$ of streams over $A$ is defined by

$$\text{STREAM}(A) = A^* \cup (A^* \times \{\perp\}) \cup A^\infty.$$ 

Here, $A^*$ denotes the set of finite streams, i.e., finite sequences of atoms from $A$, and includes $\varepsilon$, the empty stream. $A^* \times \{\perp\}$ denotes the set of partial streams, i.e., finite sequences of atoms ending with $\perp$, and includes $\perp$, the totally undefined stream (note that, for convenience, the one element stream $\langle \perp \rangle$ simply consisting of $\perp$ is denoted by $\perp$, too). $A^\infty$ denotes the set of infinite streams, i.e., infinite sequences of atoms (which may also be represented by total functions $\mathbb{N} \to A$).

The following four functions are used on streams:

- $\text{ap} : A^\perp \times \text{STREAM}(A) \to \text{STREAM}(A)$,
- $\text{rest} : \text{STREAM}(A) \to \text{STREAM}(A)$,
- $\text{first} : \text{STREAM}(A) \to A^\perp$,
- $\text{isempty} : \text{STREAM}(A) \to \{\text{tt}, \text{ff}\}$,

defined by

$$\text{ap}(a, s) = \begin{cases} \langle a \rangle \circ s & \text{if } a \in A, s \in \text{STREAM}(A), \\ \perp & \text{otherwise.} \end{cases}$$

The one-element sequence is denoted by $\langle a \rangle$, and the usual concatenation of two sequences by $s \circ s'$. Of course, $\varepsilon \circ s = s = s \circ \varepsilon$ and if $s$ is infinite, i.e., $s \in A^\infty$, then $s \circ s' = s$ for all $s' \in \text{STREAM}(A)$. Note, however, that $\text{STREAM}(A)$ is not closed with respect to concatenation since, for $s \in A \times \{\perp\}$ and $s' \in A^\ast \setminus \{\varepsilon\}$, $s \circ s' \not\in \text{STREAM}(A)$.

Let $a \in A$, $s \in \text{STREAM}(A)$, $s' = \langle a \rangle \circ s$; then,

$$\text{rest}(s') = s, \quad \text{rest}(\varepsilon) = \text{rest}(\perp) = \perp,$$

$$\text{first}(s') = a, \quad \text{first}(\varepsilon) = \text{first}(\perp) = \perp,$$

$$\text{isempty}(s') = \text{ff}, \quad \text{isempty}(\varepsilon) = \text{tt}, \quad \text{empty}(\perp) = \perp.$$

To make $\text{STREAM}(A)$ into a domain, an ordering is needed. So we define for $s_1, s_2 \in \text{STREAM}(A)$

$$s_1 \sqsubseteq s_2 \text{ iff } s_1 = s_2 \text{ or } \exists s_3, s_4 \in \text{STREAM}(A): s_1 = s_2 \circ \langle \perp \rangle \text{ and } s_2 = s_3 \circ s_4.$$

Intuitively, $s_1 \sqsubseteq s_2$ holds, i.e., $s_1$ 'approximates' $s_2$ if $s_1 = s_2$ or if $s_1$ is a partial stream which is a prefix of $s_2$ if $\perp$ is dropped at the end of $s_1$. With this ordering $\text{STREAM}(A)$
forms a countable algebraic cpo. Note that $A^\perp$ can be viewed as a proper subdomain of $STREAM(A)$.

2.1. Lemma. The functions $ap$, $rest$, $first$, and $isempty$ are monotonic and continuous.

Streams can be considered as the most fundamental domain when dealing with systems of communicating processes. For procedural multiprograms with shared memory one may consider streams of states, for processes with explicit communication primitives one can think of streams of communication actions (cf. [49, 70]).

2.2. Syntax of the language AMPL

The syntax of the language AMPL (‘applicative multiprogramming language’) is close to $\lambda$-notation. However, only first-order functions are considered and the fixed-point operator is replaced by the possibility of defining a system of mutually recursive functions and streams.

\[
\text{(program)} ::= \{\text{funct} \ (\text{function identifier}) \equiv (\text{funct abstract}),\}^* \\
\text{stream} \ (\text{identifier}) \equiv (\text{expr}),\}^* (\text{expr})
\]

\[
\text{(expr)} ::= (\text{funct appl}) \ (\text{cond}) \ (\text{choice}) \ (\text{object}) \\
\text{rest} \ | \text{first} \ | \text{isempty} \ (\text{expr}) \ (\text{expr}) \ & \ (\text{expr})
\]

\[
\text{(funct appl)} ::= (\text{function})(\{ (\text{expr}) \ (, \ (\text{expr}))^* \})
\]

\[
\text{(cond)} ::= \text{if} (\text{expr}) \text{ then} (\text{expr}) \text{ else} (\text{expr}) \text{ fi}
\]

\[
\text{(choice)} ::= ((\text{expr}) \text{ o} (\text{expr})) \text{ | } ((\text{expr}) \text{ V} (\text{expr}))
\]

\[
\text{(object)} ::= (\text{primitive object}) \ (\text{identifier})
\]

\[
\text{(funct abstract)} ::= \lambda (\{ (\text{identifier}) \ (, (\text{identifier}))^* \} : (\text{expr})
\]

\[
\text{(function)} ::= (\text{funct abstract}) \ (\text{function identifier}) \ (\text{primitive function})
\]

Here we assume a flat domain $A^\perp$ of ‘atomic’ semantic values including $\mathtt{ff}$ and $\mathtt{tt}$ for the boolean values, and the natural numbers. Furthermore, we assume a set $P$ of primitive function symbols, where for every $g \in P$ an arity $n$ is given and an $n$-ary function $g': (A^\perp)^n \rightarrow A^\perp$ which is strict, and thus monotonic and continuous.

As domain we consider

$$DOM =_{\text{def}} A^\perp \cup STREAM(A).$$

The elements of $D =_{\text{def}} A \cup \{\varepsilon\}$ are called primitive objects.

Note that we do not explicitly give any context conditions, although we assume throughout the following section that all expressions are context correct, for instance, that at syntactic positions where expressions denoting streams or boolean values are expected only respective expressions occur.

The replacement of identifiers $x_1, \ldots, x_n$ by expressions $E_1, \ldots, E_n$ in an expression $E$ is denoted by $E[x_1/E_1, \ldots, x_n/E_n]$. An expression or program is called closed if no free identifiers occur in it.
The language AMPL is well-suited for defining applicative reduction programs along the lines of [54] as well as data flow programs following [32, 58]. A system of recursive definitions for streams

\[
\text{stream } s_1 = S_1, \ldots, \text{stream } s_m = S_m
\]

can immediately be seen as a nondeterministic network (cf. [51, 52]) or a data flow graph, where the streams \( s_i \) correspond to arcs and the expressions \( S_i \) correspond to nodes, where \( S_i \) has the output arc \( s_i \) and as input arcs those streams \( s_k \) that occur in \( S_i \) (for a complete treatment of this issue see Section 6).

2.3. Examples

Besides classical sequential, deterministic, applicative programs over flat domains, one may write programs in AMPL that generate infinite streams.

2.2. Example (Producer-consumer).

\[
\begin{align*}
\text{funet produce} &= A x : \text{if } x = 0 \text{ then } e \text{ else } \text{product}(x) \& \text{produce}(x - 1) \text{ fi}, \\
\text{funet consume} &= A s : \text{if isempty}(s) \text{ then } t \text{ else } g(\text{first } s, \text{consume}(\text{rest } s)) \text{ fi,} \\
&\text{consume}(\text{produce}(n))
\end{align*}
\]

For \( n \in \mathbb{N} \), for arbitrary functions ‘product’ and \( g \), and for \( t \in A \), the program ‘produce’ above produces finite streams. However, if ‘product’ is also well-defined for negative numbers, for \( n = -1 \) for example, the function ‘produce’ generates some infinite stream.

However, in contrast to classical applicative programming languages, where recursive definitions are just used for defining functions, in AMPL also streams may be defined by recursion.

2.3. Example (Hamming’s sequence). If a program is required which generates the infinite stream of all numbers \( > 1 \) of the form \( 2^i \times 3^j \times 5^k \) (cf. [35]), in ascending order, one may use three communicating streams:

\[
\begin{align*}
\text{funet streammult} &= \lambda n, s : (n \times \text{first } s) \& \text{streammult}(n, \text{rest } s), \\
\text{funet merge} &= \lambda s_1, s_2 : \text{if } \text{first } s_1 \leq \text{first } s_2 \\
&\text{then } \text{first } s_1 \& \text{merge}(\text{rest } s_1, s_2) \\
&\text{else } \text{first } s_2 \& \text{merge}(s_1, \text{rest } s_2) \text{ fi,} \\
\text{stream } s_1 &= \text{streammult}(5, 1 \& s_1), \\
\text{stream } s_2 &= \text{merge}(\text{streammult}(3, 1 \& s_2), s_1), \\
\text{stream } s_3 &= \text{merge}(\text{streammult}(2, 1 \& s_3), s_2), s_3
\end{align*}
\]

The correctness of this program may quite straightforwardly be proved using induction.

Apart from the deterministic constructs in AMPL, also nondeterministic operators are included for writing nondeterministic and concurrent programs. Concurrency is surely one of the most intricate issues in multiprogramming. Analogously to
everyday life one may talk of two (or more) concurrent candidates (processes, expressions, programs) if these two candidates both concurrently compete for something (for instance, to be served or to be elected). For resolving such a competition, a choice has to be made.

### 2.3.1. The ambiguity operator

AMPL includes two nondeterministic operators: the simple straightforward choice operator "\(D\)" of erratic nondeterminism and the ambiguity operator "\(V\)" of angelic nondeterminism. Erratic nondeterminism corresponds to the simple concept of free choice: one may consider an expression \((E_1 \sqcup E_2)\) as a competition of the expressions \(E_1\) and \(E_2\) for being chosen. However, in contrast to everyday life, this choice is performed in a totally arbitrary way without taking into account any of the particular properties of \(E_1\) or \(E_2\).

Now, let us consider the following program that can be read as a simple example for a system with two terminals (having input streams \(s_1\) and \(s_2\)) that is supposed to transmit the first input given at one of the terminals (note that if a terminal produces no input, then the input stream is \(\bot\)):

\[
[ \text{stream } s_1 = S_1, \text{stream } s_2 = S_2, \\
\text{if } C(s_1, s_2) \text{ then first } s_1 \text{ else first } s_2 \text{ fi} ]
\]

There is no way to formulate the predicate \(C\) in AMPL without using the ambiguity operator such that the first alternative is chosen only if \(\text{first } s_1 \neq \bot\) and the second one is chosen only if \(\text{first } s_2 \neq \bot\) (and one of them is chosen ambiguously if both are \(\neq \bot\)). If such a predicate would be definable, then functions \(g\) would be definable such as the 'parallel or' \(g(tt, \bot) = tt, g(\bot, tt) = tt, g(ff, ff) = ff\).

According to [47] such a function is not definable in our nondeterministic language if we forget about \(V\). Note that then all definable functions are sequential (cf. Section 5.1). But even the 'parallel or' would not solve the problem above.

However, with \(V\) we use a more strongly defined choice operator such as McCarthy's ambiguity operator, the meaning of which is specified in [67] as follows:

"We define a basic ambiguity operator \(\text{amb}(x, y)\), whose possible values are \(x\) or \(y\) when both are defined; otherwise, whichever is defined."

Formally, the definition may be written:

\[
(x \nabla y) = \begin{cases} 
  x \sqcup y & \text{if } x \neq \bot, y \neq \bot \\
  x & \text{if } x \neq \bot, y = \bot, \\
  y & \text{if } x = \bot, y \neq \bot, \\
  \bot & \text{if } x = \bot, y = \bot, 
\end{cases}
\]

and \(C(s_1, s_2)\) in the program above may be expressed by

\(\neg \text{isempty } s_1 \nabla \text{isempty } s_2\).
This is a nonstrict extension (and so not a natural one) of the choice operator.

The difference between erratic and angelic choice becomes more clear if we consider the following simple example:

\[
\text{funct choose } = \lambda n : n \triangleright \text{choose}(n + 1), \\
\text{funct choose'} = \lambda n : n \triangleright \text{choose'}(n + 1).
\]

The application \text{choose}(n) may give all natural numbers greater than or equal to \(n\) as a result, but must not diverge; \text{choose'}(n), however, includes a diverging computation.

With the angelic choice we may program, for instance, nonsequential ('parallel') operators such as parallel "and":

\[
\text{funct parand } = \lambda b_1, b_2: \\
\quad \text{if } \neg b_1 \text{ then } \text{ff else } \text{fi } \triangleright \text{if } \neg b_2 \text{ then } \text{ff else } \text{fi } \triangleright (b_1 \land b_2)
\]

Another example where the differences between angelic and erratic choice become obvious is merging.

2.3.2. Merging

Besides the simple example of deterministic merging, one of the most important paradigms of multiprogramming is nondeterministic merging:

\[
\text{funct merge_}_1 = \lambda s_1, s_2: \\
\quad \text{if } \text{isempty } s_1 \land \text{isempty } s_2 \text{ then } \epsilon \\
\quad \text{else } (\text{first } s_1) \land \text{merge_}_1(\text{rest } s_1, s_2) \sqsupset \\
\quad \text{if } \text{isempty } s_1 \text{ then } (\text{first } s_2) \land \text{merge_}_1(s_1, \text{rest } s_2) \text{fi}.
\]

The function \text{merge} works perfectly for (finite or infinite) total streams. On partial streams, however, it fails. If one input stream is \(\perp\), then \text{merge_}_1(\perp, s) always gives \(\perp\) even if \(s\) is infinite. The alternative:

\[
\text{funct merge_}_1 = \lambda s_1, s_2: \\
\quad (\text{first } s_1) \land \text{merge_}_1(\text{rest } s_1, s_2) \sqsupset (\text{first } s_2) \land \text{merge_}_1(s_1, \text{rest } s_2)
\]

gives for \text{merge_}_1(\perp, s) both \(\perp\) and \(s\) as possible result. This is why we call \text{merge_}_1 a 'strict merge'. A merge that suppresses \(\perp\) can be programmed with the help of the ambiguity operator:

\[
\text{funct merge_}_2 = \lambda s_1, s_2: \\
\quad \text{if } \text{parand}(\text{isempty}(s_1), \text{isempty}(s_2)) \text{ then } \epsilon \\
\quad \text{else if } \neg \text{isempty } s_1 \lor \text{isempty } s_2 \\
\quad \text{then } (\text{first } s_1) \land \text{merge_}_2(\text{rest } s_1, s_2) \\
\quad \text{else if } \neg \text{isempty } s_2 \lor \text{isempty } s_1 \\
\quad \text{then } (\text{first } s_2) \land \text{merge_}_2(s_1, \text{rest } s_2) \\
\quad \text{else } \perp \text{fi} \text{ fi fi}.
\]

We call \text{merge_}_2 a 'nonstrict merge'.
Both merge₁ and merge₂ are nonfair: for infinite streams s₁ and s₂ the application merge₁(s₁, s₂) as well as merge₂(s₁, s₂) may produce s₁ (or s₂ respectively) as a result without taking into account the values of s₂ (or s₁ respectively).

But even a fair merge (cf. [74]) may be programmed with the help of the ambiguity operator (let choose be defined as above):

\[
\text{funct merge₃ = } \lambda s₁, s₂: \text{take}(s₁, s₂, \text{choose}(1)) \quad \square
\]
\[
\text{funct take = } \lambda s₁, s₂, n:
\quad \text{if } n > 0 \text{ then } (\text{first } s₁) & \text{take}(\text{rest } s₁, s₂, n - 1)
\quad \text{else } \text{take}(s₂, s₁, \text{choose}(1)) \fi
\]

Note that merge₃ is formulated using the function choose (see Section 2.3.1 above) and thus includes the possibilities of unbounded nondeterminism as given by the ambiguity operator (cf. [1]). The merge function defined by merge₃ is fair but strict. The definition of a 'nonstrict, fair merge' is impossible in AMPL. It would lead into intricate problems of monotonicity.

3. Mathematical semantics

Since the language of the preceding section includes nondeterministic choice operations, generally, a set of possible results exists for an expression. So mathematical semantics is given by defining a semantic function mapping expressions onto sets of semantic values.

With systems of recursive nondeterministic equations for functions, fixed points are associated that represent nondeterministic functions. They are characterized by an appropriate combination of the Egli–Milner ordering and set-inclusion ordering. In particular, the fixed point for a recursive function \( f = \tau[f] \) is defined in three steps: As a first approximation a least fixed point is associated with it in the powerdomain comprising only closed, convex, finitely approximable sets. Then a fixed point is looked for in the set of closed sets. In contrast to classical fixed-point theory, this fixed point is not a least one, but is characterized with the help of the other fixed point. For the ambiguity operator, which is not monotonic in the Egli–Milner ordering, a special treatment is needed.

With the systems of recursive nondeterministic equations for streams, we associate sets of fixed points representing determinate streams. Thus, an instance of a process (represented by a stream) has one unique determinate 'identity' for all its cooperating processes within an instantiation of a nondeterministic system. So the so-called merge anomaly (cf. [14, 53]) is avoided, too.

3.1. Basic definitions

A countable algebraic domain is a partially ordered set with least element, where
(1) every directed set has a least upper bound ("lub");
(2) the set of finite elements is countable;
(3) every object is the lub of a directed set of finite elements.

Here, a set $S$ is called directed if, for every pair of elements $x, y \in S$, there is an element $z \in S$ with $x \subseteq z$ and $y \subseteq z$. An element $x$ is called finite if, for every directed set $S$ with $x \subseteq \text{lub } S$, we have $x \subseteq z$ for some $z \in S$. An element $x$ is called total if it is maximal w.r.t. $\subseteq$, otherwise it is called partial.

3.2. Powerdomains revisited

In a nondeterministic computation, i.e., in the evaluation of a nondeterministic program we find two dimensions of progress of the computation: on the one hand, during the computation certain choices have to be made; on the other hand, better and better approximations of the intended result are to be obtained. The first dimension is captured by the inclusion ordering on sets, the second one by the 'less defined' ordering in domains. Powerdomains are an attempt to capture both dimensions in one ordering. This is generally only partially possible.

In this section we recapitulate the three powerdomain constructions over a domain $\text{DOM}$ based on the idea of ideal completions. We choose very particular concrete representations for the elements of these powerdomains by elements ('sets') from the powerset over $\text{DOM}$.

Let $\text{DOM}$ be a consistently complete, countably algebraic domain; let $\text{FDOM}$ denote the set of finite elements from $\text{DOM}$. For $S, S_1, S_2 \subseteq \text{DOM}$ we define
$$\text{CLOSE}(S) = \{x \in \text{DOM} \mid \forall a \in \text{FDOM}: a \subseteq x \Rightarrow \exists z \in S: a \subseteq z \subseteq x\},$$
$$\text{UPC}(S) = \{x \in \text{DOM} \mid \exists y \in S: y \subseteq x\},$$
$$\text{MIN}(S) = \{x \in S \mid \forall y \in S: y \subseteq x \Rightarrow x = y\},$$
$$\text{DOC}(S) = \{x \in \text{DOM} \mid \exists y \in S: x \subseteq y\},$$
$$\text{CONE}(S) = \{x \in \text{DOM} \mid \exists y, z \in S: y \subseteq x \subseteq z\}.$$

$S$ is called convex iff $\text{CONE}(S) = S$; then, $S$ is called closed iff $\text{CLOSE}(S) = S$. We have
$$\text{CONE}(S) = \text{UPC}(S) \cap \text{DOC}(S).$$

Trivially, all these functions and notions on sets can be extended to set-valued functions and to functionals over these functions by applying them element-wise.

The following three pre-orderings are used (cf. [75, 79]):

$S_1 \sqsubseteq_E S_2$ iff $\forall x \in S_1 \exists y \in S_2: x \subseteq y$,

$S_1 \sqsubseteq_M S_2$ iff $\forall y \in S_2 \exists x \in S_1: x \subseteq y$,

$S_1 \sqsubseteq_{EM} S_2$ iff $S_1 \sqsubseteq_E S_2 \land S_1 \sqsubseteq_M S_2$.

Over nonflat (nondiscrete) domains these relations just define pre-orderings. What sets are identified if we try to make these relations into orderings can be seen from the following lemma.
3.1. Lemma. For sets $S_1, S_2$ we have,

\begin{align*}
S_1 \subseteqM S_2 & \text{ iff } \text{UPC}(S_1) \subseteqM \text{UPC}(S_2), \\
S_1 \subseteqE S_2 & \text{ iff } \text{DOC}(S_1) \subseteqE \text{DOC}(S_2), \\
S_1 \subseteqEM S_2 & \text{ iff } \text{CONE}(S_1) \subseteqEM \text{CONE}(S_2), \\
S_1 \subseteqM S_2 & \text{ iff } \text{UPC}(S_2) \subseteq \text{UPC}(S_1), \\
S_1 \subseteqE S_2 & \text{ iff } \text{DOC}(S_1) \subseteq \text{DOC}(S_2).
\end{align*}

This lemma shows one pathological property of the powerdomains based on these 'orderings': in a powerdomain, particular distinct sets are considered as being equivalent, i.e., the powerdomain constructions can actually be considered as classes of equivalent sets. But in the powerdomain construction, sets may not only be equivalent because they cannot be distinguished by the orderings above. It is interesting to distinguish only between those sets that can be distinguished by the respective pre-ordering for a finite set of finite elements from the given domain.

Due to the principles of finite approximability and continuity, two sets are considered to be equivalent in a countably algebraic powerdomain based on some of the above orderings iff the classes of finite sets of finite elements that approximate these sets in the sense of these orderings are identical. Based on these principles an explicit powerdomain construction is now given. Here, we take a very concrete set-theoretic view of powerdomains. Their elements are just represented by elements from $\mathcal{P}(\text{DOM})$, i.e., by particular elements of the powerset over DOM. These representations are chosen in a very particular way which is most convenient for our semantic descriptions. For sets $S_1, S_2 \subseteq \text{DOM}$ we define equivalence-relations:

\begin{align*}
S_1 \simEM S_2 & \text{ iff } \forall S \subseteq \text{FDOM}, |S| < \infty : (S \subseteqEM S_1 \Leftrightarrow S \subseteqEM S_2), \\
S_1 \simM S_2 & \text{ iff } \forall S \subseteq \text{FDOM}, |S| < \infty : (S \subseteqM S_1 \Leftrightarrow S \subseteqM S_2), \\
S_1 \simE S_2 & \text{ iff } \forall S \subseteq \text{FDOM}, |S| < \infty : (S \subseteqE S_1 \Leftrightarrow S \subseteqE S_2).
\end{align*}

Based on these equivalence relations we define the following three subsets of $\mathcal{P}(\text{DOM})$ that are used as representations of the class of sets equivalent to $S$ in the powerdomains.

\begin{align*}
\mathcal{C}_{\text{EM}}(S) &= \bigcup \{ S_0 \subseteq \text{DOM} | S \simEM S_0 \}, \\
\mathcal{C}_{\text{M}}(S) &= \bigcup \{ S_0 \subseteq \text{DOM} | S \simM S_0 \}, \\
\mathcal{C}_{\text{E}}(S) &= \bigcup \{ S_0 \subseteq \text{DOM} | S \simE S_0 \}.
\end{align*}

Based on these definitions we now define three powerdomains as subsets of $\mathcal{P}(\text{DOM})$:

\begin{align*}
\mathcal{P}_{\text{EM}}(\text{DOM}) &= \text{def} \{ \mathcal{C}_{\text{EM}}(S) | S \in \mathcal{P}(\text{DOM}) \}, \\
\mathcal{P}_{\text{M}}(\text{DOM}) &= \text{def} \{ \mathcal{C}_{\text{M}}(S) | S \in \mathcal{P}(\text{DOM}) \}, \\
\mathcal{P}_{\text{E}}(\text{DOM}) &= \text{def} \{ \mathcal{C}_{\text{E}}(S) | S \in \mathcal{P}(\text{DOM}) \}.
\end{align*}
A theory for nondeterminism, parallelism, communication, and concurrency

\( \mathcal{P}_{\text{EM}}(\text{DOM}) \) is the power domain of \textit{erratic nondeterminism} (also called Plotkin powerdomain or Egli–Milner powerdomain). Note that we have chosen a \( \subseteq \)-maximal maximal representation for the elements of \( \mathcal{P}_{\text{EM}}(\text{DOM}) \), i.e., \( C_{\text{EM}}(S) \) is the \( \subseteq \)-maximal set in the class of sets that are \( \equiv_{\text{EM}} \)-equivalent w.r.t. \( \equiv_{\text{EM}} \)-approximations by finite sets of finite elements.

\( \mathcal{P}_{\text{M}}(\text{DOM}) \) is the powerdomain of \textit{demonic nondeterminism} (also called Smyth powerdomain). Note that we have chosen a \( \subseteq \)-maximal representation for \( \mathcal{P}_{\text{M}}(\text{DOM}) \), i.e., \( C_{\text{M}}(S) \) is the \( \subseteq \)-maximal set in the class of closed, finitely approximable sets that are \( \equiv_{\text{M}} \)-equivalent w.r.t. \( \equiv_{\text{M}} \)-approximations by finite sets of finite elements.

\( \mathcal{P}_{\text{E}}(\text{DOM}) \) is the powerdomain of \textit{angelic nondeterminism} (also called Hoare power-domain). Note that we have chosen a \( \subseteq \)-maximal representation for \( \mathcal{P}_{\text{E}}(\text{DOM}) \), i.e., \( C_{\text{E}}(S) \) is the \( \subseteq \)-maximal set in the class of closed, finitely approximable sets that are \( \equiv_{\text{E}} \)-equivalent w.r.t. \( \equiv_{\text{E}} \)-approximations by finite sets of finite elements.

Basically, these powerdomains contain just those subsets from \( \text{DOM} \) for which the respective relations form partial orderings and which can be approximated by finite sets of finite elements. The concept of finite observability over algebraic domains simply means that two objects are equal iff their classes of finite approximations are identical.

The above constructions show the shortcomings of the three orderings on which powerdomains are based. Therefore, it is natural to look for a more appropriate ordering. Inclusion-ordering of course seems the natural ordering for dealing with set-valued functions.

Unfortunately, the simple powerset without the empty set ordered by inclusion-ordering does not form a domain. For very obvious reasons we do not accept the empty set as an element (i.e., as possible set of results of a program) since the set of possible computations of a nondeterministic program can never be empty. However, \( (\mathcal{P}(\text{DOM}) \setminus \{\emptyset\}, \subseteq) \) forms a predomain, i.e., it has all properties of a domain besides the existence of a least element. We restrict ourselves to closed sets, i.e., to sets \( S \) where, with every directed set in \( S \), its least upper bound is also in \( S \). This is motivated by the concept of finite observability. Every object should be determined by its finite approximations.

Accordingly, the \textit{power predomain of closed sets} is defined as follows:

\[
\mathcal{P}_{\text{C}}(\text{DOM}) = \{ S \subseteq \text{DOM} \mid S = \text{CLOSE}(S) \}.
\]

A function \( f : \mathcal{P}(\text{DOM}) \rightarrow \mathcal{P}(\text{DOM}) \) is called \textit{closely union-continuous} iff, for every \( \subseteq \)-directed set \( X \subseteq \mathcal{P}(\text{DOM}) \),

\[
f(\text{CLOSE}(\bigcup X)) = \text{CLOSE}(\bigcup \{ f(x) \mid x \in X \}).
\]

The three different powerdomain constructions reflect three different viewpoints on nondeterminism. These different viewpoints are also reflected in distinct views of operational semantics of nondeterminism.
Now we define the following sets of functions:
\[ \text{NDF}^m = \text{DOM}^m \rightarrow \text{PC}(\text{DOM}), \quad \text{NDF}^m_{\text{EM}} = \text{DOM}^m \rightarrow \mathcal{P}_{\text{EM}}(\text{DOM}), \]
\[ \text{NDF} = \{ f \in \text{NDF}^m | m \in \mathbb{N} \}. \]

We write for \( f_1, f_2, f_i \in \text{NDF}^m \)
\[ f_1 \subseteq^* f_2 \text{ iff } \forall e_1, \ldots, e_m \in \text{DOM}: f_1(e_1, \ldots, e_m) \subseteq f_2(e_1, \ldots, e_m), \]
\[ (\bigcup^* f_i)(e_1, \ldots, e_m) = \text{def } \bigcup f_i(e_1, \ldots, e_m), \]
where \( f_i \subseteq^* f_{i+1} \), and similarly for \( f_1, f_2, f_i \in \text{NDF}_{\text{EM}}^m \)
\[ f_1 \subseteq^* \text{EM} f_2 \text{ iff } \forall e_1, \ldots, e_m \in \text{DOM}: \]
\[ f_1(e_1, \ldots, e_m) \subseteq_{\text{EM}} f_2(e_1, \ldots, e_m), \]
\[ (\subseteq_{\text{EM}}^* \text{ub} f_i)(e_1, \ldots, e_m) = \text{def } \subseteq_{\text{EM}}^* \text{ub} f_i(e_1, \ldots, e_m), \]
where \( f_i \subseteq^* \text{EM} f_{i+1} \). Let \( \tau, \tau' \) be functionals, \( \tau, \tau': \text{NDF}^m \rightarrow \text{NDF}^m \). We write
\[ \tau \subseteq^* \tau' \text{ iff } \forall f \in \text{NDF}^m: \tau[f] \subseteq^* \tau'[f]. \]
\( \tau \) is called inclusion-monotonic if, for \( f_1, f_2 \in \text{NDF}^m \),
\[ f_1 \subseteq^* f_2 \Rightarrow \tau[f_1] \subseteq^* \tau[f_2]. \]
\( \tau \) is called closely union-continuous if, for \( f_i \in \text{NDF}^m \) with \( f_i \subseteq^* f_{i+1} \),
\[ \tau[\text{CLOSE}(\bigcup^* f_i)] = \text{CLOSE}(\bigcup^* \tau[f_i]). \]

Now we have introduced all the mathematical notions we need for giving a denotational meaning to AMPL-programs.

3.3. Mathematical semantics of recursion-free expressions

As usual in denotational semantics, we use the idea of environments.
\[ \text{ENV} = \text{def } [(\text{FI} \cup \text{OI}) \rightarrow (\text{NDF} \cup \text{DOM})], \]
where \( \text{FI} \) denotes the set of function identifiers and \( \text{OI} \) the set of object identifiers and, of course, for every \( \sigma \in \text{ENV} \), we assume in the sequel that we have \( \sigma(f) \in \text{NDF} \) for \( f \in \text{FI} \) and \( \sigma(x) \in \text{DOM} \) for \( x \in \text{OI} \). The updating of an environment is denoted by \( \sigma[a/y] \), where
\[ \sigma[a/y](z) = \begin{cases} 
\sigma(z) & \text{for } z \neq y, \\
\sigma(a) & \text{otherwise}. 
\end{cases} \]
Now we can define the semantic function $B : \langle \text{expr} \rangle \rightarrow \text{ENV} \rightarrow \text{DOM}$. We write $B_\sigma[E]$ for $B(E)(\sigma)$.

\[
B_\sigma[\text{if } C \text{ then } E_1 \text{ else } E_2 \text{ fi}] = \bigcup_{c \in B_\sigma(C)} \begin{cases} B_\sigma[E_1] & \text{if } c = \text{tt}, \\ B_\sigma[E_2] & \text{if } c = \text{ff}, \\ \{\bot\} & \text{otherwise}; \end{cases}
\]

\[
B_\sigma[E_1 \sqcap E_2] = B_\sigma[E_1] \cup B_\sigma[E_2];
\]

\[
B_\sigma[E_1 \sqcup E_2] = (B_\sigma[E_1] \cup B_\sigma[E_2]) \cup \{\bot\} \cup \{\emptyset\}
\]

\[
B_\sigma[(\lambda x_1, \ldots, x_n : E)(E_1, \ldots, E_n)] = \bigcup_{e_i \in B_\sigma(E_i)} \bigcup_{e_n \in B_\sigma(E_n)} B_\sigma[e_i/x_1, \ldots, e_n/x_n][E];
\]

\[
B_\sigma[g(E_1, \ldots, E_n)] = \{g'(e_1, \ldots, e_n) | e_i \in B_\sigma[E_i]\}
\]

for primitive functions $g$;

\[
B_\sigma[f(E_1, \ldots, E_n)] = \{e \in \sigma[f](e_1, \ldots, e_n) | e_i \in B_\sigma[E_i]\} \quad \text{for } f \in \text{FI};
\]

\[
B_\sigma[x] = \{\sigma(x)\} \quad \text{for } x \in \text{OI};
\]

\[
B_\sigma[E \& S] = \{\text{ap}(e, s) | e \in B_\sigma[E] \land s \in B_\sigma[S]\};
\]

\[
B_\sigma[\text{first } S] = \{\text{first}(s) | s \in B_\sigma[S]\};
\]

\[
B_\sigma[\text{rest } S] = \{\text{rest}(s) | s \in B_\sigma[S]\};
\]

\[
B_\sigma[\text{isempty } S] = \{\text{isempty}(s) | s \in B_\sigma[S]\};
\]

\[
B_\sigma[e] = \{e\} \quad \text{for } e \in \text{DOM}.
\]

Note that the definition of function application corresponds to what is termed 'call-time-choice' in [46] (for an extended discussion, see Section 4).

### 3.4. Nondeterminate functions as fixed points

Now we give a semantic definition for systems of recursively defined functions.

#### 3.4.1. Semantics of recursive systems

For a given environment $\sigma_0$, we consider the system

\[
\text{funct } f_1 = F_1, \ldots, \text{funct } f_n = F_n
\]

of function definitions. Assume that $F_i$ stands for $\lambda x_1, \ldots, x_{n_i} : E_i$. Now we define

\[
\tau : \text{NDF}^a_1 \times \cdots \times \text{NDF}^a_n \rightarrow \text{NDF}^a_1 \times \cdots \times \text{NDF}^a_n
\]
by
\[ \tau[h_1, \ldots, h_n](e_1, \ldots, e_a) = \text{CLOSE}(B_\sigma[E_i]), \]
where \( \sigma = \sigma_0[h_1/f_1, \ldots, h_n/f_n, e_1/x_1, \ldots, e_a/x_a]. \) Note that our definition is independent from the choice of \( \sigma_0 \) since we assume that in the \( E_i \)s only the \( f_i \)s occur as free identifiers.

Now we consider two particular other functionals
\[ \tau_{EM} : \text{NDF}^{a_1}_{EM} \times \cdots \times \text{NDF}^{a_n}_{EM} \to \text{NDF}^{a_1}_{EM} \times \cdots \times \text{NDF}^{a_n}_{EM}, \]
\[ \tau_s : \text{NDF}^{a_1} \times \cdots \times \text{NDF}^{a_n} \to \text{NDF}^{a_1} \times \cdots \times \text{NDF}^{a_n}, \]
defined by
\[ \tau_{EM}[h_1, \ldots, h_n](e_1, \ldots, e_a) = C_{EM}(B_\sigma[E_0]), \]
\[ \tau_s[h_1, \ldots, h_n](e_1, \ldots, e_a) = \text{CLOSE}(B_\sigma[E_0]), \]
where always \( \sigma = \sigma_0[h_1/f_1, \ldots, h_n/f_n, e_1/x_1, \ldots, e_a/x_a] \) is assumed, and \( E_0 \) is obtained from \( E_i \) by replacing all occurrences of the operator \( \forall \) by \( \Box. \)

3.2. Theorem. \( \tau \) and \( \tau_s \) are inclusion-monotonic and closely union-continuous.

Proof. According to our semantic definitions, all program constructs besides \( \forall \) correspond to additive functionals of the form
\[ A[E_1, \ldots, E_m] = \{ h(e_1, \ldots, e_m) | e_i \in B_\sigma[E_i] \}, \]
or to a finite union of such functionals where \( h \) is independent of \( \sigma \) (standing for a continuous function) or \( h = \sigma_0[f_j] \) for some \( j \). So all such language constructs are inclusion-monotonic. The inclusion-monotonicity of \( \forall \) is easily checked. The composition of inclusion-monotonic functionals leads to inclusion-monotonic functionals. The closely union-continuity follows from the fact that all results of the functions are, by definition, closed sets. \( \Box. \)

3.3. Theorem. \( \tau_{EM} \) is \( \preceq_{EM} \)-monotonic and \( \preceq_{EM} \)-continuous.

Proof. According to our semantic definition, all language constructs (besides \( \forall \) that does not occur in \( E_0 \)) are of the the form
\[ A[E_1, \ldots, E_m] = \{ h_k(e_1, \ldots, e_m) | e_i \in B_\sigma[E_i] \land 1 \leq k \leq k_0 \}, \]
where \( h_k \) is a constant (standing for a continuous function) or \( h_k = \sigma_0[f_j] \) for some \( j \) such that \( h_k \in \text{NDF}^{a_n}_{EM}. \) The operator \( C_{EM} \) distributes:
\[ C_{EM}(A[E_1, \ldots, E_n]) \]
\[ = C_{EM}((C_{EM}(h_k(e_1, \ldots, e_m))| e_i \in C_{EM}(B_\sigma[E_i]) \land 1 \leq k \leq k_0)). \]
This immediately gives the $\leq_{EM}$-monotonicity and $\leq_{EM}$-continuity. □

Trivially, we obtain, by the definition of $\tau_S$ and $\tau_{EM}$,

$$\tau \subseteq^* \tau_S \subseteq^* \tau_{EM}.$$ 

Now we consider fixed points for all of the three functionals $\tau_{EM}$, $\tau_S$, $\tau$. At first we define $f_i^{EM}$ to be the $\leq_{EM}$-least fixed point of the equation $[f_1, \ldots, f_n] = \tau_{EM}[f_1, \ldots, f_n]$. Trivially, such a fixed point exists and is well-defined due to the $\leq_{EM}$-monotonicity of $\tau_{EM}$. Now we can observe

$$\text{MIN}([f_1^{EM}, \ldots, f_n^{EM}]) = \text{MIN}(\tau_{EM}[f_1^{EM}, \ldots, f_n^{EM}])$$

$$\subseteq^* \text{MIN}(\tau_S[f_1^{EM}, \ldots, f_n^{EM}])$$

$$\subseteq^* \tau_S[\text{MIN}(f_1^{EM}), \ldots, \text{MIN}(f_n^{EM})].$$

Hence, we obtain $f'_i \subseteq^* f_i^{i+1}$, where the $f'_i$s are functions inductively defined by

$$f'_0 = \text{MIN}(f_1^{EM}), \quad [f'_1^{i+1}, \ldots, f'_n^{i+1}] = \tau_S[f'_1, \ldots, f'_n]$$

as a simple consequence of the union-monotonicity of $\tau_S$. Hence, we can now define

$$f'_n = \text{CLOSE}(\bigcup^* f'_i).$$

Due to the close union-continuity of $\tau_S$ we obtain

$$[f'_1, \ldots, f'_n] = \tau_S[f'_{EM}, \ldots, f'_n].$$

Now, define $f'_i$ by

$$[f'_1, \ldots, f'_n] = \bigcup^* \{ \tau[h_1, \ldots, h_n] \mid h_i \subseteq^* \tau[h_1, \ldots, h_n] \subseteq^* f_i^S \}.$$ 

According to the $\leq$-monotonicity and to the fact that we have $\tau[f'_1, \ldots, f'_n] \subseteq [f'_1, \ldots, f'_n]$, the theorem of Knaster/Tarski immediately gives that the $f'_i$s are least fixed points of the equation $[f'_1, \ldots, f'_n] = \tau[f'_1, \ldots, f'_n]$. The functions $f'_1, \ldots, f'_n$ are actually taken as the meaning of the system of recursively defined functions above.

3.4.2. Properties of the semantic definitions

For the functions $f'_i$ the set $f'_i(e_1, \ldots, e_n)$ is never empty since $f'_i(e_1, \ldots, e_n)$ can be obtained by an intersection

$$f'_i(e_1, \ldots, e_n) = \bigcap_{j \in \text{ORD}} g'_i(e_1, \ldots, e_n)$$

of closed sets $g'_i(e_1, \ldots, e_n)$, where $g'_i$ is defined by

$$g'_0 = f'_1, \quad g'_i+1 = \tau[g'_i, \ldots, g'_i],$$

where $j \in \text{ORD}$ and $\text{ORD}$ is the set of ordinal numbers. In particular, due to the $\leq_{EM}$-monotonicity of our language constructs we may prove

$$\text{MAX}(f'_i^S(e_1, \ldots, e_n)) \leq f'_i(e_1, \ldots, e_n).$$
It may not be clear from the semantic construction given above why these particular semantic definitions were chosen. Therefore, we now add some explanations. For simplicity, we consider a single (non-mutually) recursive definition

\[ \text{funct } f = \lambda x_1, \ldots, x_n : E. \]

Let all definitions be as in the construction above, but just drop the indices. Now, define a family of function abstractions

\[ L_i = \lambda x_1, \ldots, x_n : E_i \]

where \( E_0 = \text{def } E \) and \( E_{i+1} = \text{def } E[L_i/f] \). Since we assume that \( E \) is closed (apart from occurrences of \( x_1, \ldots, x_n \) and \( f \)), we may drop the environment \( \sigma \) in \( B_\sigma[L_i(e_1, \ldots, e_n)] \) and just write \( B[L_i(e_1, \ldots, e_n)] \), where we assume that we use \( f' \) for \( f \) in \( \sigma \), i.e., \( \sigma[f'] = f' \).

For simplicity, we assume that the ambiguity operator does not occur in \( E \). Now, define \( G = (\lambda x_1, \ldots, x_n : \bot) \) and the function \( \Omega \) by

\[ \Omega(e_1, \ldots, e_n) = B_\sigma[G(e_1, \ldots, e_n)]. \]

An expression \( E_i \) is called a deterministic descendant (d.d.) of an expression \( E_0 \) if \( E_i \) does not contain the operator \( \sqcup \) and can be obtained from \( E_0 \) by successively replacing all occurrences of subexpressions of the form \((C_1 \sqcup C_2)\) by either \( C_1 \) or by \( C_2 \).

A sequence of function abstractions \( D_i \) is called a choice sequence for \( L_i \) if

\[ D_0 = \lambda x_1, \ldots, x_n : C_0 \]

and \( C_0 \) is a d.d. of \( E_0 \);

\[ D_{i+1} = \lambda x_1, \ldots, x_n : C_{i+1} \]

and \( C_{i+1} \) is a d.d. of \( C_i[(\lambda x_1, \ldots, x_n : E)/f] \). We immediately obtain:

1. all \( C_i \) are d.d. of \( E_i \);
2. \( B_\sigma[D_i[G/f](e_1, \ldots, e_n)] \leq B_\sigma[L_i[G/f](e_1, \ldots, e_n)] \).

Now, let \( d_i \) be defined such that \( B_\sigma[D_i[G/f](e_1, \ldots, e_n)] = \{d_i\} \). Note that \( D_i[G/f] \) is deterministic! The \( \{d_i\}_{i \in \mathbb{N}} \) form a chain since the deterministic language constructs are monotonic. By DD we denote the set of all such chains of \( \{d_i\}_{i \in \mathbb{N}} \) that we can obtain by choice sequences for \( L_i \).

Now the appropriateness of our definition of the fixed point \( f' \) can be seen from

\[ f'(e_1, \ldots, e_n) = \text{CLOSE}([\text{lub } d_i | \{d_i\}_{i \in \mathbb{N}} \in \text{DD}]). \] (1)

We will prove (1): By the construction of \( f' \) and DD we have

\[ C_{\text{EM}}(f'(e_1, \ldots, e_n)) = C_{\text{EM}}(\text{CLOSE}([\text{lub } d_i | \{d_i\}_{i \in \mathbb{N}} \in \text{DD}])) \]

and therefore, in (1), \( f'(e_1, \ldots, e_n) \) must be included in the right-hand side since \( f' \) is \( \preceq^* \)-least in the set of \( \preceq_{\text{EM}}^* \)-equivalent functions. Moreover,

\[ \text{MIN}(f'(e_1, \ldots, e_n)) = \text{MIN}(\text{CLOSE}([\text{lub } d_i | \{d_i\}_{i \in \mathbb{N}} \in \text{DD}])) \]
A theory for nondeterminism, parallelism, communication, and concurrency

since \( \text{MIN}(\tau'[\Omega](e_1, \ldots, e_n)) = \text{MIN}({d_i \mid \{d_i\}_{i \in \mathbb{N}} \in \mathbb{D}}) \). Moreover, due to the monotonicity of our language constructs, we have

\[
\{d_i\} \sqsubseteq_{\text{EM}} B_{\sigma[\text{MIN}(f)/f]}[D_i(e_1, \ldots, e_n)]
\]

(2)

and vice versa (since \( \text{lub}\{d_i\} \) must be approximated by a minimal element):

\[
\text{MIN}((\subseteq_{\text{EM}} \cup \text{lub}) B_{\sigma[\text{MIN}(f)/f]}[D_i(e_1, \ldots, e_n)]) \sqsubseteq_{\text{M}} \{\text{lub}\{d_i\}\}.
\]

Hence, according to the definition of \( \sqsubseteq_{\text{M}} \), we have

\[
\exists x \in \text{MIN}((\subseteq_{\text{EM}} \cup \text{lub}) B_{\sigma[\text{MIN}(f)/f]}[D_i(e_1, \ldots, e_n)]): x \sqsubseteq \text{lub}\{d_i\}.
\]

Since (according to (2)) \( \text{lub}\{d_i\} \) is below all elements in the set that \( x \) is taken from, we have that there exists some \( x \in \text{MIN}((\subseteq_{\text{EM}} \cup \text{lub}) B_{\sigma[\text{MIN}(f)/f]}[D_i(e_1, \ldots, e_n)]) \) such that \( x \sqsubseteq \text{lub}\{d_i\} \sqsubseteq x \), so \( \text{lub}\{d_i\} \sqsubseteq f'(e_1, \ldots, e_n) \).

3.4.3. A short discussion of the ambiguity operator

The particular problems that are caused by the ambiguity operator in the semantic definitions can be explained by its strange properties:

- it allows writing 'noncontinuous' functions (w.r.t. \( \sqsubseteq_{\text{EM}} \)) such as functions based on functions like

\[
\text{funct } \psi_x = \lambda x : (x \lor f(x + 1)),
\]

where \( f(0) \) represents a set which is not finitely approximable in the Egli-Milner ordering, i.e., it is not a member of the powerdomain of erratic nondeterminism (cf [1, 25]);

- it is not monotonic in the Egli-Milner ordering since \( \{1\} \sqsubseteq_{\text{EM}} \{1\} \) and \( \{\bot\} \sqsubseteq_{\text{EM}} \{2\} \), but

\[
B[1 \lor \bot] = B[1 \lor 2] = \{1\} \lor \{1, 2\} \quad \text{but} \quad \neg(\{1\} \sqsubseteq_{\text{EM}} \{1, 2\});
\]

this is even worse since, in the case of noncontinuity but monotonicity (like in the fairness discussion), according to Knaster/Tarski one can still work with least fixed points (cf. [1]).

Note that the ambiguity operator \( \lor \) really brings a new important concept into our language since it is not monotonic in the Egli-Milner ordering (cf. [17, 18, 19]). It allows to program unbounded nondeterminism, i.e., functions with sets of possible results that are not finitely approximable (cf. [17, 18, 19]), and therefore, in combination with the choice operator, one can define functions \( f \) such that the set of arguments \( \{x \in \text{DOM} \mid \bot \in f(x)\} \) is neither recursively enumerable nor corecursively enumerable, but \( \Sigma_1 \) (cf. [30]). In particular, the \( \lor \)-operator allows to write all kinds of nonstrict functions and nonsequential functions in the sense of [84] (cf. the generalization to nondeterministic functions in Section 5).

In particular, the \( f_i \)'s provide a fully abstract (in the sense of [68]) fixed-point semantics for recursive definitions of nondeterministic functions in AMPL, as far as no recursive definitions for streams occur; i.e., for a program \( P \):

\[
[\text{funct } f_1 = F_1, \ldots, \text{funct } f_n = F_n, E]
\]

(3)
the semantics is defined by $B[P] = B[E[f'_1/f_1, \ldots, f'_n/f_n]]$, where the $f'_i$s are the fixed points defined above. The semantics is fully abstract (cf. [68]) since, for a recursive definition:

$$\text{funct } f_1 \equiv G_1, \ldots, \text{funct } f_n \equiv G_n,$$

(4)

the fixed points associated with (3) are identical to the fixed points associated with (4) iff, for every expression $E$, definition (4) can be used in $P$ without changing its meaning.

3.5. Streams as fixed points of systems of nondeterministic equations

Now, consider a system $CS$ of recursively defined streams and functions:

$$CS = (\text{funct } f_1 \equiv F_1, \ldots, \text{funct } f_n \equiv f_n, \text{stream } s_1 \equiv S_1, \ldots, \text{stream } s_m \equiv S_m),$$

where we assume that the $s_i$ do not occur in $F_j$. Before we start giving meaning to recursive stream equations, we discuss a technical problem. Basically, there are two possibilities to interpret recursive equations $f = \tau[f]$ with nondeterministic right-hand side. We may consider it as a recursive equation for the set of possible results, i.e., for a set-valued function. Or we may interpret it as a set of equations for deterministic functions. In the first alternative, in distinct applications distinct choices can be made. In the second alternative, the same deterministic function is used in all applications.

A system of expressions communicating by streams corresponds to a mutually recursive definition of streams. This coincides with the result of [16], where it is shown that every tail-recursive system of concurrent processes working on shared variables within conditional critical regions can be transformed into a system of mutually recursive nondeterministic procedures. If the right-hand side $E$ of a stream equation consists of a deterministic expression, functionally, no difference can be found between the stream equation $\text{stream } s \equiv E$ and the recursive definition of a nullary function $\text{funct } s \equiv \lambda : E$ (cf. the proposal of [51, 52]).

In the presence of nondeterminism, recursive stream definitions cannot be simply understood by nullary functions. For instance, the system $S_1$

$$[\text{funct } f \equiv \lambda : (1 \downarrow 2) \& \text{empty}, \text{first } f( ) = \text{first } f( )]$$

should be different from the system $S_2$

$$[\text{stream } s = (1 \downarrow 2) \& \text{empty}, \text{first } s = \text{first } s]$$

since functions are treated by simple substitutions (‘call-by-name’-like with respect to the body of the function), while streams are to be substituted (element-wise) only after evaluation (‘call-by-value’ or, more precisely, ‘call-time-choice’) such that

$$B[S_1] = \{tt, ff\}, \quad \text{while} \quad B[S_2] = \{tt\}.$$  

Note that this question is equivalent to the question whether the choice operator distributes over the fixed-point operator. To avoid mixing alternative possibilities
of behaviour (cf. the 'merge anomaly' described in [53, 14]), we have to consider recursive stream equations as *sets of fixed-point equations* rather than one fixed-point equation for a set-valued ('nondeterministic') nullary function. To explain the reasons let us consider the following example.

### 3.4. Example (The merge anomaly [53])

We consider the program

```
[ stream x = 0 & (0 & ε),
  stream y = merge2(x, z),
  stream z = g(y) ]
```

where merge2 is defined as in Section 2.3.2 and g is given by

```
funct g = λ s:
  if first s = 0 then 1 & rest s
  else 0 & ε fi.
```

This corresponds to the data flow graph shown in Fig. 1.

![Data flow graph](image)

**Fig. 1.**

Operationally, one would expect that first 0 is taken from the left input of merge2 (z is still ⊥). So the first element of y has to be 0 and thus, the first element of z has to be 1. However, if we associated sets of values with x, y and z, then the fact that 1 is the first element of z would allow to conclude from the definition of merge (since y = merge2(x, z)) that 1 is a possible first element of y and thus that 0 is a possible first element of z, which is counter-intuitive since 1 is only available in response to taking 0 as first element of y and thus 1 as first element of z. It looks as if 0 would overhaul 1.

More precisely, if one liked to define the semantics of the three recursively defined streams by associating the sets of streams X, Y, and Z with the stream-identifiers x, y, and z respectively as the least fixed point of

\[
(X, Y, Z) = \bigcup_{(x,y,z) \in (X,Y,Z)} (B[0 \& 0 \& ε], B[merge2(x, z)], B[g(y)]),
\]

then if there is a \( y \in Y \) with first(y) = 0, then there is a \( z \in Z \) with first(z) = 1, and thus there is a \( y' \in B[merge2(x, z)] \subseteq Y \) with first(y') = 1 and thus there exists a
$z' \in B[g(y')]$ with $\text{first}(z') = 1$. This fictitious, amazing contradiction can be made understandable if we apply definition (5) to the data flow graph of Fig. 1. Following the fixed-point idea we would be allowed to 'unfold' the loop leading to the flow graph of Fig. 2 or, representing this data flow program by equations, to:

$$\text{stream } x = 0 \& (0 \& e),$$

$$\text{stream } z = g(\text{merge}_2(x, g(\text{merge}_2(x, z))),).$$

But now, operationally, 1 may be the first of $z$ in this data flow graph: the two data flow graphs are obviously not equivalent w.r.t. input/output. A fixed-point definition like the one above cannot work! Here, 0 actually has the possibility to overhaul 1!

However, this does not mean that fixed-point theory is the wrong technique here. It just has to be applied in the right way: Let $\text{MERGE}$ be the set of continuous functions $\text{merge} : \text{STREAM}(A)^2 \rightarrow \text{STREAM}(A)$ with

$$\forall s_1, s_2 \in \text{STREAM}: \exists d \in \{1, 2\}^\infty : \text{merge}(s_1, s_2) = \text{schedule}(s_1, s_2, d),$$

where

$$\text{schedule}(s_1, s_2, 1 \& d) = \text{first}(s_1) \& \text{schedule}(\text{rest}(s_1), s_2, \text{rest}(d)),$$

$$\text{schedule}(s_1, s_2, 2 \& d) = \text{first}(s_2) \& \text{schedule}(s_1, \text{rest}(s_2), \text{rest}(d)).$$

Now we define the sets $X, Y, Z$ by

$$(X, Y, Z) = \bigcup_{\text{merge} \in \text{MERGE}} \{ \text{FIX } \lambda (x, y, z) : (0 \& 0 \& e, \text{merge}(x, z), g(y)) \}.$$

Here, $\text{FIX}$ denotes the least-fixed-point operator for stream processing functions. Such fixed points exist since all functions are monotonic. A short analysis shows that now our mathematical semantics coincides with the operational one that we want to associate with the data flow program.

So, in the sequel, definitions of the semantics of recursively defined streams will follow this idea. In particular, a semantics can be given in the classical style of
A theory for nondeterminism, parallelism, communication, and concurrency

fixed-point theory without switching to particular, more complicated stream representations like 'scenarios' (cf. [14]) or 'tagged streams' (cf. [60]).

An analysis of Keller's Example 3.4 clearly shows that the merge anomaly is the result of taking the wrong interpretation for the fixed-point equations for streams. The value of \( y \) has to be produced by one consistent (i.e., determinate) instance of the function merge. Every stream identifier has to denote one unique stream in one particular instance of the nondeterminate system. So for avoiding the merge anomaly, a nondeterministic recursive stream has to be considered as a set of recursive deterministic equations and not as a recursive definition of a nondeterministic stream (i.e., a set of streams). Therefore, we introduce

\[
SDF^n = \text{DOM}^n \rightarrow \text{PC}([\text{DOM}^n \rightarrow \text{DOM}]), \quad SDF = \{ f \in SDF^n | n \in \mathbb{N} \}
\]

and environments

\[
\text{NENV} = (\text{FI} \cup \text{OI}) \rightarrow (\text{SDF} \cup \text{DOM}).
\]

Now the meaning of an expression is defined by

\[
\text{BF}: (\text{expr}) \rightarrow \text{NENV} \rightarrow \text{PC}([\text{NENV} \rightarrow \text{DOM}]).
\]

We write \( \text{BF}_\sigma[E] \) for \( \text{BF}[E](\sigma) \).

\[
\text{BF}_\sigma[\text{if } C \text{ then } E_1 \text{ else } E_2 \text{ fi}] = \{ \lambda \sigma : \text{IF}(g(\sigma), h_1(\sigma), h_2(\sigma)) | g \in \text{BF}_\sigma[C] \land h_1 \in \text{BF}_\sigma[e_1] \land h_2 \in \text{BF}_\sigma[e_2] \},
\]

where

\[
\text{IF}(c, e_1, e_2) = \begin{cases} e_1 & \text{if } c = \text{tt}, \\ e_2 & \text{if } c = \text{ff}, \\ \perp & \text{otherwise}; \end{cases}
\]

\[
\text{BF}_\sigma[E_1 \sqcup E_2] = \text{BF}_\sigma[E_1] \cup \text{BF}_\sigma[E_2];
\]

\[
\text{BF}_\sigma[E_1 \sqcap E_2] = \{ \text{AMB}(h_1, h_2)(\sigma) | h_1 \in \text{BF}_\sigma[E_1] \land h_2 \in \text{BF}_\sigma[E_2] \},
\]

where

\[
\text{AMB}(h_1, h_2)(\sigma) = \begin{cases} \{ h_1 \} & \text{if } h_1(\sigma) \neq \perp \land h_2(\sigma) = \perp, \\ \{ h_2 \} & \text{if } h_1(\sigma) = \perp \land h_2(\sigma) \neq \perp, \\ \{ h_1, h_2 \} & \text{otherwise}; \end{cases}
\]

\[
\text{BF}_\sigma(x) = \{ \lambda \sigma_1 : \sigma_1(x) \};
\]

\[
\text{BF}_\sigma[\lambda x_1, \ldots, x_k : E](E_1, \ldots, E_n) = \{ h \in \text{BF}_\sigma[E] | h_i \in B_\sigma[E_i] \},
\]
where \( \sigma_1 = \sigma[h_1(\sigma)/x_1, \ldots, h_n(\sigma)/x_n]; \)
\[
\begin{align*}
\text{BF}_\sigma[g(E_1, \ldots, E_n)] &= \{ \lambda \sigma_1 : g'(h_1(\sigma_1), \ldots, h_n(\sigma_1)) \mid h_1 \in \text{BF}_\sigma[E_1] \} \land h_1 \in \sigma(g)(h_1(\sigma), \ldots, h_n(\sigma_1)); \\
\text{BF}_\sigma[f(E_1, \ldots, E_n)] &= \{ \lambda \sigma_1 : h(h_1(\sigma_1), \ldots, h_n(\sigma_1)) \mid h_1 \in \text{BF}_\sigma[E_1] \land h \in \sigma(f)(h_1(\sigma), \ldots, h_n(\sigma)) \}; \\
\text{BF}_\sigma[e] &= \{ \lambda \sigma_1 : e \} \text{ for } e \in \text{DOM}; \\
\text{BF}_\sigma[\text{first } E] &= \{ \lambda \sigma_1 : \text{first}(h(\sigma_1)) \mid h \in \text{BF}_\sigma[E] \}; \\
\text{BF}_\sigma[\text{rest } E] &= \{ \lambda \sigma_1 : \text{rest}(h(\sigma_1)) \mid h \in \text{BF}_\sigma[E] \}; \\
\text{BF}_\sigma[\text{isempty } E] &= \{ \lambda \sigma_1 : \text{isempty}(h(\sigma_1)) \mid h \in \text{BF}_\sigma[E] \}; \\
\text{BF}_\sigma[E_1 \& E_2] &= \{ \lambda \sigma_1 : \text{ap}(h_1(\sigma_1), h_2(\sigma_1)) \mid h_1 \in \text{BF}_\sigma[E_1] \land h_2 \in \text{BF}_\sigma[E_2] \}.
\end{align*}
\]

Note that \( \text{BF}_\sigma[E] \) is independent of \( \sigma \) if the ambiguity operator does not occur in \( E \). With the recursively defined function identifiers we associate functions \( F'_i \) defined in the same style as the functions \( f'_i \) above using the semantic definition of BF instead of B.

Given the environment \( \sigma_\perp \) with \( \sigma_\perp(f)(e_1, \ldots, e_n) = \{ \lambda x_1, \ldots, x_n : \bot \} \) and \( \sigma_\perp(x) = \bot \) for all \( f \in F_I, x \in O_I \), we associate with the recursive definitions of \( f_1, \ldots, f_n \) the environment \( \sigma \) defined by

\[
\sigma = \sigma_\perp[F'_1/f_1, \ldots, F'_n/f_n].
\]

3.5. Lemma. All \( f \in F'(e_1, \ldots, e_n) \) with \( F' \in \text{BF}_\sigma[E](\sigma) \) are monotonic and continuous.

Proof. All language constructs (apart from the choice operators) are monotonic and continuous when restricted to deterministic expressions. The \( f \in \text{BF}[E](t) \) are generated just by composition of such constructs and are thus continuous. \( \square \)

Now, for the system CS introduced at the beginning of this section, we define its meaning by associating with the recursive stream equations the set of all tuples of streams that are least fixed points of the following equation:

\[
[s_1, \ldots, s_n] = [H_1(\sigma'), \ldots, H_n(\sigma')],
\]

where \( \sigma' = \sigma[s'_1/s_1, \ldots, s'_n/s_n] \) and \( H_i \in \text{BF}_\sigma[S_i] \). This definition is mathematically sound since all \( H_i \in \text{BF}_\sigma[S_i] \) are continuous.

Now, by \( \Sigma \) we denote the closure of the set of all \( \sigma 's \) from the definition above. The precise definition of \( \Sigma \) reads

\[
\Sigma = \text{CLOSE} \left( \{ Y(\sigma) \mid H_i \in \text{BF}_{Y(\sigma)}[S] \} \right),
\]
where \( Y(\sigma) \) is an abbreviation of

\[
\text{FIX } \lambda \sigma' : \sigma[H_1(\sigma')/s_1, \ldots, H_n(\sigma')/s_n].
\]

\text{FIX} again denotes the fixed-point operator. Note that the monotonicity properties of the involved constructs guarantee that, for each \( \sigma \in H \), such a \( Y(\sigma) \) exists.
3.6. Semantics of programs

Now the semantics of a closed AMPL-program P

\[
\begin{align*}
\begin{array}{c}
\text{funct } f_1 = F_1, \ldots, \text{funct } f_n = F_n, \\
\text{stream } s_1 = S_1, \ldots, \text{stream } s_k = S_k, E
\end{array}
\end{align*}
\]

is defined by

\[
B[P] = \bigcup_{\sigma \in \Sigma} \{ BF_{\sigma}[E](\sigma') \},
\]

where \( \Sigma \) is defined as in Section 3.5.

A system of mutually recursively defined streams can be seen as a nondeterministic Kahn network. The nodes of the graph are associated with sets of deterministic, continuous functions. In operational terms the conceptual function of the ambiguity operator can be explained as follows: if no ambiguity operator occurs in the system, then a node simply corresponds to a set of deterministic networks, each of which can be freely chosen for some computational instantiation; if the ambiguity operator occurs, however, then this choice has to be done dynamically during the computation depending on the semantic properties of the input and on the values of approximations.

4. Operational semantics

Operational semantics is defined in the form of conditional term rewriting rules. Since our language includes nondeterminism, these rules cannot be confluent.

Simple rules for the evaluation of the conditional, choice, and ambiguity operator are

\[
\begin{align*}
\text{if } \text{true} \text{ then } E_1 \text{ else } E_2 \text{ fi } &\rightarrow E_1, \\
\text{if } \text{false} \text{ then } E_1 \text{ else } E_2 \text{ fi } &\rightarrow E_2, \\
C_1 \Rightarrow C_2 &\Rightarrow \text{if } C_1 \text{ then } E_1 \text{ else } E_2 \text{ fi } &\rightarrow \text{if } C_2 \text{ then } E_1 \text{ else } E_2 \text{ fi}, \\
E_1 \sqcap E_2 &\rightarrow E_1, \quad E_1 \sqcup E_2 \rightarrow E_2, \\
e \in D &\Rightarrow (E \sqcap e \rightarrow e) \land (e \sqcup E \rightarrow e), \\
E_1 \rightarrow E_1' \land E_2 \rightarrow E_2' &\Rightarrow (E_1 \sqcap E_2) \rightarrow (E_1' \sqcap E_2').
\end{align*}
\]

For identifiers \( x \) and the bottom element \( \bot \), two simple rules are used:

\[
\bot \rightarrow \bot, \quad x \rightarrow x.
\]

The rules so far work for expressions having atoms as values as well as for expressions having streams as values. For atoms \( a \) the term rewriting idea is simple: if \( a \) is a possible value of \( E \), then \( E \rightarrow \cdots \rightarrow a \) should hold. For streams the situation is less clear. For finite total streams we could use the same idea. But what about partial or even infinite streams? Of course, one could use the idea of defining a computation sequence \( E \rightarrow E_1, \ldots \) such that the \( E_i \)'s form better and better approximations for the resulting stream (cf. [20]). The other possibility is term rewriting rules with output: if \( E = e \& S \) with \( e \in D \), then \( E \rightarrow^* S \) means \( E \) is rewritten into
S (forming the rest of the resulting stream or, more precisely, the expression denoting it) while giving the object e as output (which, of course, forms the first element of the resulting stream). S could be called 'resumption' (cf. [75]), e the output of E.

For the function application the following rules are used for propagating computed primitive objects:

\[
E_i \in D \Rightarrow (\lambda x_1, \ldots, x_n : E)(E_1, \ldots, E_n) \\
= (\lambda x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n : E[E_i/x_i])(E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n),
\]

\[
e \in D \Rightarrow (\lambda x_1, \ldots, x_n : e)(E_1, \ldots, E_n) \Rightarrow e,
\]

\[
e \in D \Rightarrow (\lambda x_1, \ldots, x_n : e \& S)(E_1, \ldots, E_n) \\
= (\lambda x_1, \ldots, x_n : S)(E_1, \ldots, E_n),
\]

\[\forall i, 1 \leq i \leq n: e_i \in D \Rightarrow g(e_1, \ldots, e_n) \Rightarrow g'(e_1, \ldots, e_n).
\]

If no primitive objects can be propagated, then parallel evaluation is required. (6/8) define the parallel evaluation of the arguments of primitive functions g, (7/9) define the unfold of recursively defined functions with the simultaneous parallel evaluation of the arguments.)

\[
(\forall i, 1 \leq i \leq n: E_i \Rightarrow E_i' \lor (E_i = E_i' \land E_i \in D)) \\
\Rightarrow \left\{ \begin{array}{l}
\exists i, 1 \leq i \leq n: E_i \Rightarrow E_i' \Rightarrow g(E_1, \ldots, E_n) \Rightarrow g(E_1', \ldots, E_n'), \\
E_i = E_i' \land E_i = e_i \land S_i \land e_i \in D
\end{array} \right.
\]

For the language constructs manipulating streams, the following rules are required:

\[e \in D \Rightarrow e \& S \Rightarrow S,\]

\[S \Rightarrow S' \Rightarrow \text{first } S \Rightarrow e,\]

\[S \Rightarrow S' \Rightarrow \text{rest } S \Rightarrow S',\]

\[S \Rightarrow S' \Rightarrow \text{isempty } S \Rightarrow \text{ff},\]

\[E \Rightarrow E' \Rightarrow E \& S \Rightarrow E' \& S,\]

\[S \Rightarrow S' \Rightarrow \text{first } S \Rightarrow \text{first } S',\]

\[S \Rightarrow S' \Rightarrow \text{rest } S \Rightarrow \text{rest } S',\]

\[S \Rightarrow S' \Rightarrow \text{isempty } S \Rightarrow \text{isempty } S',\]

\[\text{isempty } e \Rightarrow \text{tt}, \quad \text{first } e \Rightarrow \bot, \quad \text{rest } e \Rightarrow \bot,\]
A theory for nondeterminism, parallelism, communication, and concurrency

Now, let CS denote the system \( \text{stream } s_1 = S_1, \ldots, \text{stream } s_m = S_m, E \) and CS' denote the system \( \text{stream } s'_1 = S'_1, \ldots, \text{stream } s'_m = S'_m, E' \). The rules for evaluating such systems read:

\[
S \rightarrow S' \Rightarrow ((S \nabla S') \rightarrow S') \land ((S' \nabla S) \rightarrow S').
\]

Now, let CS denote the system \( \text{stream } s_1 = S_1, \ldots, \text{stream } s_m = S_m, E \) and CS' denote the system \( \text{stream } s'_1 = S'_1, \ldots, \text{stream } s'_m = S'_m, E' \). The rules for evaluating such systems read:

\[
S_i = \varepsilon \Rightarrow \text{CS} \rightarrow (\text{stream } s_1 = S_1[\varepsilon/ s_i], \ldots, \text{stream } s_{i-1} = S_{i-1}[\varepsilon/ s_i])
\]

\[
\text{stream } s_{i+1} = S_{i+1}[\varepsilon/ s_i], \ldots, \text{stream } s_m = S_m[\varepsilon/ s_i],
\]

\[
E[\varepsilon/ s_i]),
\]

\[E \in D \Rightarrow \text{CS} \rightarrow E,
\]

\[
\forall i, 1 \leq i \leq n: ((S_i \rightarrow S'_i \land s'_i = s_i) \lor (S_i \nrightarrow S'_i \land s'_i = e_i \& s_i))
\]

\[
\Rightarrow \left\{ \begin{array}{l}
(E \rightarrow E" \land E' = E"[s'_i/s_1, \ldots, s'_m/s_m]) \Rightarrow \text{CS} \rightarrow \text{CS}' \\
(E \nrightarrow E" \land E' = E"[s'_i/s_1, \ldots, s'_m/s_m]) \Rightarrow \text{CS} \nrightarrow \text{CS}'
\end{array} \right.
\]

The computation rule above defines an evaluation strategy for expressions which may be called \textit{data-driven reduction} (cf. [31]); only conditionals are evaluated in a demand-driven mode. For systems of recursive equations it models data-driven data flow semantics. So such a system can operationally be seen as a loosely coupled data flow network where the stream processing functions in its nodes are evaluated by data-driven reduction (cf. asynchronous systems in [71]).

The main difference between the computation rule "\( \rightarrow \)" and the rules given in [66] is found in the different substitution mechanisms. In [66], all substitutions are UNFOLDINGs replacing an identifier \( f_i \) for a recursively defined function in one indivisible action (where \( F_i = \lambda x_1, \ldots, x_n : E \)):

\[
f_i(E_1, \ldots, E_n) \rightarrow E[E_1/x_1, \ldots, E_n/x_n],
\]

where the \( E_1, \ldots, E_n \) all are deterministic (since in [66] only deterministic expressions are considered), while in the rule "\( \rightarrow \)" this unfolding action is divided in upto \( n+1 \) independent substitution steps.

4.1. Remark (\textit{Call-by-value versus call-by-name revisited}). As is well-known, for recursively defined functions call-by-value and call-by-name rules may produce different results if expressions with undefined values occur as arguments. Then the function corresponding to call-by-value may be strictly less defined than the function defined by call-by-name. In particular, one can give proper fixed-point theories for each of these rules ("smash product" of domain \((A^n)^*\) versus "cartesian product" \((A^n)^\times\), cf. [11]).

For nondeterministic functions still another difference between call-by-value and call-by-name becomes apparent. Consider the examples

\[
(P_1) \quad [\text{funct } f_1 = \lambda x : x + x, f_1(0 \square 1)],
\]

\[
(P_2) \quad [\text{funct } f_2 = \lambda x : 2 \times x, f_2(0 \square 1)].
\]
In strict call-by-value, we obtain $B[P_1] = B[P_2] = \{0, 2\}$, while (as pointed out in [45]) in straightforward call-by-name semantics, we obtain $B[P_1] = \{0, 1, 2\}$ and $B[P_2] = \{0, 2\}$, although from the mathematical point of view the functions $f_1$ and $f_2$ are (in a deterministic environment) equivalent. So in straightforward call-by-name semantics, one is forced to consider functions as mappings $\mathcal{P}(S^1)^n \rightarrow \mathcal{P}(S^1)$, whereas in call-by-value it suffices to take $(S^n)^1 \rightarrow \mathcal{P}(S^1)$ (cf. [5, 12, 44]).

In our definition of mathematical semantics a mixture of call-by-value and call-by-name is used which is called call-time-choice in [46]. It can be understood as an extension of delayed evaluation (cf. [83]) or call-by-need (cf. [85]) to nondeterministic functions. It allows one to consider nondeterministic functions as elements from $(S^1)^n \rightarrow \mathcal{P}(S^1)$. The parallel evaluation rule (call-in-parallel) as defined in this section contrasts the implementation of call-time-choice (and thus of call-by-name in the deterministic case) by delayed evaluation (call-by-need) by a method which does not delay the evaluation of the arguments until they become decisive and are thus needed, but starts the evaluation of the body of the function and of the arguments in parallel, simply eliminating computations of arguments which are apparently no longer needed. So one might, in analogy, talk of enforced evaluation.

With the computation rule "\$\rightarrow\$" a function application

$$ (\lambda x_1, \ldots, x_n : E_{n+1})(E_1, \ldots, E_n) $$

is evaluated by $n$ independent processes evaluating $E_i$ which communicate their results under the identifier $x_i$ to the process $E_{n+1}$. If the evaluation of $E_{n+1}$ needs a value for some identifier $x_i$, then its evaluation stops and waits until the value is communicated. If the value is never communicated, i.e., if the evaluation of $E_i$ fails or does not terminate, then the process waits forever.

Here, an important difference between call-by-value and call-by-name (or more precisely call-time-choice) can be seen. In the case of call-by-name, the whole system of processes (of evaluations) terminates iff the process terminates which evaluates $E_{n+1}$ (which needs the termination of all processes evaluating the expressions $E_i$ for which $x_i$ is actually needed in $E_{n+1}$), while in the case of call-by-value, the whole system of processes terminates iff all processes terminate themselves. For parallel evaluation using call-by-value see [15].

The computation rule above defines the data-driven counterpart to demand-driven techniques such as delayed evaluation [23] or call-by-need [85], which are used for efficient implementations of call-by-name, as well as for lazy evaluation [43, 39], which is used for effective implementations of infinite structures (cf. [11]). But it even includes nondeterminism (for modelling the decision process of schedulers) and also allows the partial evaluation of expressions with free identifiers (cf. mixed computation in [37]) and thus copes with computations for expressions with incomplete information, which, in the end, communicating processes are. The evaluation of an expression with free identifiers always leads to infinite computation sequences, at least if one of the values for the identifiers 'becomes decisive', i.e., is actually
needed. So the partial evaluation should be stopped as soon as the computation sequence becomes constant. Furthermore, the rule genuinely includes the evaluation of nonsequential functions as, for instance, the ambiguity operator, which definitely require a (quasi-)parallel evaluation of (subsets of) their argument expressions and therefore, incorporate mathematically the concept of parallelism and concurrency.

Based on the relation "\( \rightarrow \)" as defined above we define the following relations:

\[
E_0 \rightarrow E_i \text{ iff } E_0 \rightarrow \cdots \rightarrow E_i,
\]

\[
E_0 \rightarrow^* E_i \text{ iff } E_0 \rightarrow \cdots \rightarrow E_k \rightarrow E_{k+1} \rightarrow \cdots \rightarrow E_i,
\]

\[
E_1 \rightarrow^* E_2 \text{ iff } \exists i \in \mathbb{N}: E_1 \rightarrow E_2,
\]

\[
E_0 \rightarrow^{e_1, \ldots, e_k} E_k \text{ iff } \forall i_1, \ldots, i_k \in \mathbb{N}: E_0 \xrightarrow{e_{i_1}} E_1 \rightarrow \cdots \rightarrow E_{k-1} \xrightarrow{e_{i_k}} E_k,
\]

\[
E_0 \rightarrow^{(e_1)} \text{ iff } E_0 \rightarrow^{e_1} E_1 \rightarrow^{e_2} E_2 \rightarrow \cdots.
\]

We say that an evaluation of the expression \( E \) may fail iff there is an infinite sequence \( \{E_i\}_{i \in \mathbb{N}} \) with \( E = E_0 \) and \( E_i \rightarrow E_{i+1} \). Note that no 'finite' errors (exceptions) may occur since, according to the rule \( \perp \rightarrow \perp \), 'finite' errors are turned into nonterminating computations.

Let

\[
P = \left[ \text{funct } f_1 = F_1, \ldots, \text{funct } f_n = F_n, S \right],
\]

\[
P' = \left[ \text{funct } f_1 = F_1, \ldots, \text{funct } f_n = F_n, S' \right]
\]

be closed programs, where

\[
S = (\text{stream } s_1 = S_1, \ldots, \text{stream } s_m = S_m, E),
\]

\[
S' = (\text{stream } s'_1 = S'_1, \ldots, \text{stream } s'_k = S'_k, E').
\]

A computation rule "\( \rightarrow \)" for the language AMPL is called:

- consistent iff, for all \( P, S \rightarrow S' \Rightarrow B[P] \subseteq B[P] \land S \rightarrow^* S' \Rightarrow \forall x \in B[P]: \text{ap}(e, x) \in B[P] \);
- complete iff, for all \( P \) and for all \( e \in B[P] \),
  1. \( e \in A \Rightarrow S \rightarrow^* e \),
  2. \( e = \langle e_1 \rangle \circ \cdots \circ \langle e_l \rangle \Rightarrow S \xrightarrow{e_1 \cdots e_l} e \),
  3. \( e = \langle e_1 \rangle \circ \cdots \circ \langle e_l \rangle \circ \langle \perp \rangle \Rightarrow \exists E : S \xrightarrow{e_1 \cdots e_l} S' \) and \( S' \) may fail,
  4. \( e = \langle e_1 \rangle \circ \cdots \Rightarrow \forall i \in N \exists S': S \xrightarrow{e_1 \cdots e_l} S' \);  
- finitary iff, for all \( P, \{S'|S \rightarrow S'\} \) is finite;
- effective iff, for all \( P \), the evaluation of \( S \) may fail iff \( \perp \in B[P] \) and \( \{S'|S \rightarrow S'\} \) is empty iff \( S \in D \) or \( B[P] = \perp \).
An operational semantics $\rightarrow$ is called \emph{tight} if for every program $P$ the rule $\rightarrow$ is consistent, complete, and effective. $\rightarrow$ is called \emph{loose} (cf. [74]) if $\rightarrow$ is consistent and effective.

Of course, for multiprograms $P$ a loose operational semantics is fully sufficient since we do not require for a particular system or implementation that all values $x \in B[P]$ are possibly produced; we are satisfied if at least one feasible computation is executed.

4.2. Lemma. \emph{There does not exist a finitary tight operational semantics for the language AMPL.}

\textbf{Proof.} If, for every expression $E$ of AMPL, there is only a finite number of expressions $E'$ such that $E \rightarrow E'$ holds, then, according to König’s Lemma, there must be an infinite computation path for choose(1), where choose is defined as in Section 2. So the computation rule is either not complete or not effective. \hfill $\square$

So one may give loose finitary, or tight but not finitary operational semantics for AMPL.

Now a sequence $R_i$ of rewriting rules is defined. Let $R_0$ be defined like $\rightarrow$. $R_{i+1} = R_i \cup W_i$, where the rule $W_i$ is defined by the rules:

\begin{align*}
E \rightarrow i e \land e \in A & \Rightarrow (E \lor E' \rightarrow e) \land (E' \lor E \rightarrow e), \\
S \rightarrow i S'' & \Rightarrow (S \lor S' \rightarrow S'') \land (S' \lor S \rightarrow S'').
\end{align*}

Now we define $R_\infty = \bigcup_{i \in \mathbb{N}} R_i$. $R^*$ is written for $\rightarrow^*$ and $R^*_i(e)$ for $\rightarrow^* i e$ (and $R^*_i$, $R^*_i(e)$ respectively). Each of the rules defines an operational semantics via

\begin{align*}
\text{OP}_i : \text{PROGRAM} & \rightarrow \mathcal{P}\left(\text{DOM}\right) \quad \text{for } i \in \mathbb{N} \cup \{\infty\},
\end{align*}

where

\begin{align*}
\text{OP}_i\left[\left[\text{funct } f_1 = F_1, \ldots, \text{funct } f_n = F_n, E \right] \right] \\
= \{e \in A \mid ER^*_i e\} \cup \begin{cases}
\{\bot\} & \text{if } E \text{ may fail,} \\
\emptyset & \text{otherwise}
\end{cases}
\end{align*}

for programs generating atoms, and for programs generating streams we have

\begin{align*}
\text{OP}_i\left[\left[\text{funct } f_1 = F_1, \ldots, \text{funct } f_n = F_n, S \right] \right] \\
= \{s \in \text{STREAM}(A) \mid SR^*_i(s) e\} \cup \{s \in A^\infty \mid SR^*_i(s)\} \\
\cup \{s \circ (\bot) \in \text{STREAM}(A) \mid SR^*_i(s) S' \text{ and } S' \text{ may fail}\}.
\end{align*}

4.3. Theorem. (1) For $i \in \mathbb{N}$ the operational semantics defined by $R_i$ are loose and finitary, but not complete.
(2) $\text{OP}_i \subseteq \text{OP}_{i+1}$.
(3) $(\bigcup_{i \in \mathbb{N}} \text{OP}_i) \subseteq \text{OP}_\infty$, but $(\bigcup_{i \in \mathbb{N}} \text{OP}_i) \neq \text{OP}_\infty$.
(4) $\text{CLOSE}(\text{OP}_\infty[P]) = B[P]$ for all closed programs $P$ where in $P$ the ambiguity operator is not applied to stream expressions, i.e., $R_\infty$ is tight, but of course $R_\infty$ is not finitary.

Proof. The proof of the consistency of our rules can be simply checked by comparison of the term rewriting rules with the definition of the mathematical semantics. The rules for unfolding recursive functions or stream definitions are trivially consistent because of the fixed-point properties of the recursively defined functions and streams.

The proof of effectiveness is first done only for programs without recursive stream definitions. Without loss of generality, let $P$ be of the form $\text{funet} \; f \equiv F; E \]$. If $E$ does not contain applications of recursive functions, then the rewriting process does not fail, provided $\bot \not\in B_\sigma[E]$. This can be shown by straightforward structural induction on the structure of the terms.

If the ambiguity operator does not occur in $P$, then $\bot \in B[P]$ iff $\bot \in B_\sigma[E[f'/f]]$ for all $i$, where $f'$ is defined as in Section 3.3. Hence, if $\bot \not\in B[P]$, there exists an $i$ such that $\bot \not\in B_\sigma[E^i]$ where $E^i = E[F^i/f]$ and

$$F^0 = \lambda x_1, \ldots, x_n : \bot, \quad F^{i+1} = F[F^i/f].$$

According to the argumentation above, the evaluation of $E^i$ does not fail. So the evaluation of $E$ does not fail either. According to König's Lemma, every computation sequence (without output) is bounded.

According to the considerations at the end of Section 3.3., every value that can be obtained by doing choices and unfoldings is a value contained in the denotational semantics.

If one ambiguity operator does occur, then, for a nondiverging computation

$$C[E_1 \triangledown E_2] \rightarrow \cdots \rightarrow C[E'_1 \triangledown E'_2],$$

either the length of the computations for $C[E_1]$ or $C[E_2]$ has to be bounded. Note that the ambiguity operator chooses the minimal length of the computation sequence for $E_1 \triangledown E_2$ if evaluated by $R_0$ and one that is at most $i$ steps longer than the minimal one if evaluated by $R_i$. So, if the length of the evaluation sequences for $C[E_1]$ or $C[E_2]$ is bounded, the same holds for $C[E_1 \triangledown E_2]$. Induction on the occurring number of ambiguity operators gives the result.

For programs with stream definitions the argumentation above can also be applied. After a finite number of computation steps each stream (that is not $\bot$) is rewritten into some form that allows to communicate its first element.

Similar considerations as above hold for the semantic definition of BF. Now, in a system of recursively defined streams in our operational semantics, for every stream a finite or infinite number of atoms is produced by every stream in a computation and transmitted.
In this way a tuple of streams for the tuple of stream-identifiers is defined. According to the properties of substitution for this tuple, there must be functions for the recursive identifiers in the expressions on the right-hand side such that this tuple of streams is a least fixed point. Vice versa, for every least fixed point the effectivity and completeness of the rules immediately give that there is a respective computation leading to that fixed point (or the fixed point corresponds to the closure for those operations).

That the rules are finitary may be simply proved by induction on the syntactic patterns. Together with the proof above Theorem 4.3(1) is obvious. Part (2) trivially holds since more rules simply lead to more computations and thus to more results. The same holds for the induction in (3). The inequality of (3) can be seen from the program

\[
\text{funct } f \equiv \lambda \ n, m. \\
\text{if } m \text{ divides } n \text{ then } 0 \\
\text{else } f(n \ast m, \text{choose}(1)) \text{ fi },
\]

where choose is defined as in Section 2. In every finitary rule, choose can only evaluate to a finite set of natural numbers and so \( \perp \in B[f(n, m)] \). This is not true for \( \text{OP}_\infty \).

It remains to prove (4). If the ambiguity operator does not occur in \( P \), then even \( \text{OP}_0 \) is complete: For every expression \( E \) the rule \( R \) defines a finitary tree that is infinite iff \( E \) may fail. Every path through that tree corresponds to a computation sequence. Branches occur if and only if \( E \) contains the choice operator such that the rule for the choice operator is applied. Note that this is the only 'overlapping' left-hand side of the term rewriting rules. If an atomic value \( a \) (or a stream with first element \( a \)) occurs in \( B[P] \), then \( a \) (or a stream with first element \( a \)) occurs in \( B[E'] \). But for expressions \( E \) without recursive definitions the completeness can again be proved by structural induction. The same argumentation holds for recursively defined streams.

If the ambiguity operator occurs in \( P \), we know that every computation for \( E \) is obtained by a particular selection of a computation of \( E' \), where \( E' \) is obtained from \( E \) by replacing all ambiguity operators by the straightforward choice operator. A computation of length \( n \) leading to \( a \in B[P] \) (or to a stream with first element \( a \)) must be a computation where the choices can be done in a way such that nondetermining computations can be avoided. But then, at least \( R_n \) may compute \( a \) (or a stream starting with \( a \)) when applied to \( E \). \( \square \)

All the subtleties in Theorem 4.3 above are caused by the ambiguity operator, as can be seen from the following corollary.

4.4. Corollary. If the ambiguity operator \( \triangledown \) does not occur in \( P \), then \( \text{CLOSE}(\text{OP}_i[P]) = B[P] \) for all \( i \in \mathbb{N} \cup \{\infty\} \). So, in particular, \( \rightarrow \) is finitary and tight for such programs.
There is just one really important discrepancy between our operational semantics OP and the mathematical semantics: in mathematical semantics only closed sets are considered, i.e., there exist programs $P$ where, for a chain $\{e_i\}$, one has $e_i \in B[P]$ and thus $e = \text{lub}\{e_i\} \in B[P]$ but $e \notin \text{OP}_i[P]$ for all $i$.

Consider $P$:

\[
\begin{align*}
\text{funct } f_1 &= \lambda n : n \sqcap f(n + 1), \\
\text{funct } f_2 &= \lambda n : f_2(n) \sqcap \text{if } n > 0 \text{ then } 1 \& f_2(n - 1) \text{ else } \epsilon \text{ fi, } f_2(f_1(0)) \, .
\end{align*}
\]

Then, $B[P] = \{1^* \circ (\bot)\} \cup \{1^*\} \cup \{1^\infty\}$, but $\text{OP}_i[P] = \{1^* \circ (\bot)\} \cup \{1^*\}$. So the infinite stream $1^\infty$ is in $B[P]$ because all its approximations are elements of $B[P]$, but it is not in $\text{OP}_i[P]$ because a sequence of expressions $\{E_i\}_{i \in \mathbb{N}}$ with

\[
E_i \rightarrow E_{i+1} \rightarrow E_{i+2} \rightarrow \ldots
\]

does not exist. Note that the definition of completeness of a rule with respect to infinite objects $e$ (cf. Theorem 4.3(4)) could be also changed to

\[
e = \langle e_i \rangle_{i \in \mathbb{N}} \Rightarrow \exists \{E_i\}_{i \in \mathbb{N}} : S \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots
\]

The actually chosen definition corresponds to 'finite observability' or to 'finite experiments'. This is in accordance with the approximation principle in algebraic domains on which fixed-point theory is based and where an infinite element is exclusively determined by its finite approximations.

The introduction of the ambiguity operator into AMPL radically changes the expressive power of the language: as already mentioned, using the ambiguity operator the set $\{x \in \text{DOM} \mid \bot \in f(x)\}$ may be $\Sigma_1^1$ for recursively defined functions. This fact is no longer surprising if one adapts the notion of loose nondeterminism (coined in [74]) to computation rules for nondeterministic functions. Let $CR$ be the set of finitary, consistent, and effective computation rules. A computation rule $R \in CR$ may be considered as a recursive set-valued function $R : \text{PROGRAM} \rightarrow \mathcal{P} (\text{DOM})$. A function $f : \text{DOM}^n \rightarrow \mathcal{P} (\text{DOM})$ is called definable in AMPL if there exists a program $P(x_1, \ldots, x_n)$ such that $\forall x_1, \ldots, x_n \in \text{DOM}$:

\[
f(x_1, \ldots, x_n) = \{y \in \text{DOM} \mid \exists R \in CR : P(x_1, \ldots, x_n) \rightarrow y\}.
\]

This gives a clear hint at why our extended notion of definability (and 'computability') leads to $\Sigma_1^1$: we have to quantify over all finitary, consistent, effective computation rules.

Note that the introduction of AND- and OR-nodes into procedural programming languages as used in [65] has some similarities to our $\sqcap$- and $\sqcup$-operator respectively. However, in our nondeterministic programming language, we can prove $C[E_1 \sqcap E_2] = C[E_1] \sqcap C[E_2]$ for each context $C$, whereas $C[E_1 \sqcup E_2] = C[E_1] \sqcup C[E_2]$ does not hold, while in [65], one always has $C[S_1 \text{ OR } S_2] = C[S_1]$ or $C[S_2]$ as well as $C[S_1 \text{ AND } S_2] = C[S_1]$ and $C[S_2]$. For this simpler case with
more convenient algebraic properties, Chandra [29] proves that both concepts are independent, i.e., that OR cannot be expressed by AND and vice versa. In our language we have \( \text{if } \texttt{tt} \lor \texttt{ff} \text{ then } E_1 \text{ else } E_2 \text{ fi} \) is equivalent to \( (E_1 \oplus E_2) \) but not to \( E_1 \lor E_2 \), which would be the case if one systematically translated the concept of alternation (as the other approach is called in [31]) into our language.

Note that the ambiguity operator \( \lor \) can be used to specify the logical 'parallel or' (and 'parallel and'):

\[
\text{funct paror} = \lambda \ x, y:
\text{if } x \text{ then } \texttt{tt} \text{ else } y \text{ fi} \lor \text{if } y \text{ then } \texttt{tt} \text{ else } x \text{ fi}
\]

Interestingly the function paror can be viewed as a determinate function with \( \text{paror}(\texttt{tt}, x) = \text{paror}(x, \texttt{tt}) = \texttt{tt}, \text{paror}(\texttt{ff}, \texttt{ff}) = \texttt{ff}, \text{paror}(\bot, \bot) = \bot \) for all \( x \in \{\texttt{tt}, \texttt{ff}, \bot\} \).

In a similar way the ambiguity operator \( \lor \) can be used to specify disjunctive (multiple) waiting. Given a guarded wait-expression

\[
\text{await } B_1 \text{ then } E_1 \lor \cdots \lor E_n \text{ then } E_n \text{ endwait},
\]

its meaning can be inductively defined by

\[
(\lambda \ b_1, \ldots, b_n::
\text{if } b_1 \lor \tilde{B}_1 \text{ then } E_1
\text{else await } b_2 \text{ then } E_2 \lor \cdots \lor b_n \text{ then } E_n \text{ endwait fi} ) (B_1, \ldots, B_n),
\]

where \( \tilde{B}_1 = b_1 \) if \( n = 1 \), and for \( n > 1 \),

\[
\tilde{B}_1 = \text{if } b_2 \text{ then } \texttt{ff} \text{ else } \bot \text{ fi} \lor \cdots \lor \text{if } b_n \text{ then } \texttt{ff} \text{ else } \bot \text{ fi}.
\]

One easily proves that in the await-construct the pairs \( (B_i \text{ then } E_i) \) can be arbitrarily permuted without changing the semantics, i.e., that in spite of the unsymmetric expression above, the await-construct is symmetric in its guards.

AMPL is completely free of global nondeterminism, i.e., all decisions can be made locally by the single processes without any feedback of other processes. Of course, certain decisions can only be recognized as feasible after a number of communication steps; however, when the communication has taken place, the decision can be made locally.

In Milner's CCS (cf. [70]) or Hoare's CSP (cf. [49]), communication is generally coupled with a nondeterministic choice. However, a single process cannot decide by itself which alternative for communication is to be chosen. The 'rendezvous'-concept needs some coordination before an actual communication occurs. In particular, if no priorities between the processes are given, a global instance is needed to resolve conflicts. As outlined in [38], there is no way of getting a fully distributed, symmetric implementation without using probabilistic computation techniques. These techniques, however, may not be satisfactory since they may restrict possible implementations on real computing systems in an inadequate way.
5. Expressive power of AMPL

In this section the expressive power of AMPL is analysed and demonstrated in several ways. At first the relationship of AMPL-functions to recursively defined functions is analysed. Second, the practical relationship to nonconventional computational concepts such as data flow and reduction is outlined.

5.1. Nonstrict and nonsequential functions

As it has been seen from the previous sections, nonstrict functions play a prominent role in applicative multiprogramming. So it seems worthwhile to relate these functions to classical notions.

In the mathematical theory of partial recursive functions a partial recursive function \( f \) is a recursively defined, partial mapping \( f: \mathbb{N}^n \rightarrow \mathbb{N} \), where, for each tuple \( x_1, \ldots, x_n \in \mathbb{N} \), the application \( f(x_1, \ldots, x_n) \) is either not defined or yields a natural number \( y \in \mathbb{N} \). This can simply be modelled by a total function \( f': \mathbb{N}^n \rightarrow \mathbb{N} \), where
\[
f'(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n) & \text{if } f(x_1, \ldots, x_n) \text{ is defined}, \\ \bot & \text{otherwise}. \end{cases}
\]

Here, \( \bot \) is just introduced as a symbol for 'undefined'. Now, let us consider a function \( g: (\mathbb{N}^\bot)^n \rightarrow \mathbb{N}^\bot \). The function \( g \) is called \textit{strict} if
\[
g(x_1, \ldots, x_n) = \bot \quad \text{if } x_i = \bot \text{ for some } i, 1 \leq i \leq n;
\]
otherwise, \( g \) is called \textit{nonstrict}. Every partial recursive function can be simply associated with a strict function. However, how are the nonstrict functions related to partial recursive functions?

Following [84, p. 55], we define: \( g \) is called \textit{sequential} if \( g \) is constant (e.g., \( n = 0 \)) or if there is an index \( i, 1 \leq i \leq n \), such that
(1) for all \( x_1, \ldots, x_n \in \mathbb{N}^\bot \):
\[
x_i = \bot \Rightarrow g(x_1, \ldots, x_n) = \bot;
\]
(2) for all \( x \in \mathbb{N}^\bot \): \( h_x: (\mathbb{N}^\bot)^{n-1} \rightarrow \mathbb{N}^\bot \), where
\[
h_x(y_1, \ldots, y_{n-1}) = g(y_1, \ldots, y_{i-1}, x, y_i, \ldots, y_{n-1})
\]
is a sequential function.
Otherwise, \( g \) is called \textit{nonsequential}.

Clearly, strict functions are sequential. Classical examples for nonstrict but sequential functions are \textit{if-then-else}, 'sequential and' and 'sequential or' (cf. [11]). Classical examples for non-sequential (but monotonic) functions are 'parallel or' and 'parallel and'. Generally, one considers monotonic functions only, which (in the case of flat domains) is equivalent to considering functions with regular tables as defined in [56, p. 334].
A function \( g : (\mathbb{N}^+)^n \to \mathbb{N}^+ \) has a regular table if for all \( i, 1 \leq i \leq n, x_1, \ldots, x_{n-1} \in \mathbb{N}^+ \)

\[
g(x_1, \ldots, x_{i-1}, \perp, x_i, \ldots, x_{n-1}) \neq \perp
\]

the function \( h: \mathbb{N}^+ \to \mathbb{N}^+ \), where

\[
h(x) = g(x_1, \ldots, x_{i-1}, x, x_i, \ldots, x_{n-1}),
\]

is constant. On flat domains every monotonic function has a regular table and even the reverse holds: functions with regular table are monotonic.

Note that, for nonsequential functions \( g \), an evaluation of a function application \( g(E_1, \ldots, E_n) \) requires the parallel (or at least the quasi-parallel) evaluation of (a subset of) the arguments.

Let us consider the applicative language as given by the syntax in Section 2, just leaving away the nondeterministic choice. A partial recursive interpretation (represented as a total function):

\[
I: \text{EXP} \to \mathbb{N}^+
\]

(for simplicity we consider just arithmetic functions) is defined by \( I[E] = x \) iff \( B[E] = \{x\} \). Of course, \( I \) associates a function \( f : (\mathbb{N}^+)^n \to \mathbb{N}^+ \) with every function abstraction \( A = \lambda x_1, \ldots, x_n : E \) by

\[
f(m_1, \ldots, m_n) = I[(\lambda x_1, \ldots, x_n : E)(m_1, \ldots, m_n)]
\]

since \( I \) is homomorphic:

\[
I[(\lambda x_1, \ldots, x_n : E)(E_1, \ldots, E_n)] = f(I[E_1], \ldots, I[E_n]).
\]

Thus, every function abstraction defines a monotonic function.

All recursively definable functions are sequential if all primitive functions on which they are based are sequential (note that the conditional is a nonstrict, but sequential function). If we add, however, a 'nonstrict, nonsequential conditional' (cf. [76, 39]) defined by

\[
I[\text{nif } C \text{ then } E_1 \text{ else } E_2 \text{ fi}] = \begin{cases} I[E_1] & \text{if } I[C] = \text{tt} \lor I[E_1] = I[E_2], \\ I[E_2] & \text{if } I[C] = \text{ff} \lor I[E_1] = I[E_2], \\ \perp & \text{otherwise}, \end{cases}
\]

then all computable functions \( g : (\mathbb{N}^+)^n \to \mathbb{N}^+ \), i.e., all monotonic functions \( g \) which are lubs of a recursively enumerable directed set of monotonic (finite) functions, are definable.

Note that this conditional \emph{nif} is the deterministic counterpart of

\[
\text{funct nif } = \lambda c, e_1, e_2:\
\]

\[
\text{if } e_1 = e_2 \text{ then } e_1 \text{ else } \perp \text{ fi } \lor \text{if } C \text{ then } e_1 \text{ else } e_2 \text{ fi},
\]

i.e., for deterministic expressions \( C, E_1, E_2 \) we have

\[
\{I[\text{nif } C \text{ then } E_1 \text{ else } E_2 \text{ fi}]\} = B[\text{nif}(C, E_1, E_2)].
\]
Nonsequential functions are of interest in multiprogramming for several reasons: they need an at least quasi-parallel evaluation (see demand-driven evaluation in Section 7), and they are related to the ambiguity operator, which can be seen as the 'ultimate' nonsequential function.

For nondeterministic functions $g : (\mathbb{N}^\uparrow)^n \rightarrow \mathcal{P}(\mathbb{N}^\uparrow)$ the definitions can be extended in a straightforward way: $g$ is called *nd-strict* if

$$\bot \in g(x_1, \ldots, x_n) \text{ if } x_i = \bot \text{ for some } 1 \leq i \leq n.$$ 

$g$ is called *nd-sequential* if $g$ is constant (e.g., $n = 0$) or if there is an index $i$, $1 \leq i \leq n$, such that

1. For all $x_1, \ldots, x_n \in \mathbb{N}^\uparrow$:
   $$x_i = \bot \Rightarrow \bot \in g(x_1, \ldots, x_n);$$

2. For all $x \in \mathbb{N}$: $h_x : (\mathbb{N}^\uparrow)^{n-1} \rightarrow \mathcal{P}(\mathbb{N}^\uparrow)$, where
   $$h_x(y_1, \ldots, y_{n-1}) = g(y_1, \ldots, y_{i-1}, x, y_i, \ldots, y_{n-1}),$$

is a nd-sequential function.

Otherwise, $g$ is called *nd-nonsequential*.

We define: $g$ has a *nd-regular table* if, for all $i$, $1 \leq i \leq n$, $x_1, \ldots, x_{i-1} \in \mathbb{N}$ with

$$\bot \notin g(x_1, \ldots, x_{i-1}, \bot, x_i, \ldots, x_{n-1}),$$

the function $h$, where $h : \mathbb{N}^\uparrow \rightarrow \mathcal{P}(\mathbb{N}^\uparrow)$ with

$$h(x) = g(x_1, \ldots, x_{i-1}, x, x_i, \ldots, x_{n-1}),$$

is constant.

From these definitions we easily obtain the following classifications: On the powerset over flat domains in the Egli-Milner ordering, monotonic functions have nd-regular tables. The reverse does not hold.

The choice operator $\oplus$ is nd-strict and hence, also nd-sequential. Combinations of this choice operator with monotonic nonstrict functions may lead to nd-nonstrict functions; the same holds for monotonic nonsequential functions. But the resulting functions still have nd-regular tables and are monotonic in the Egli-Milner ordering.

The ambiguity operator $\forall$, however, is nd-nonstrict and nd-nonsequential. It does not have a nd-regular table and it is not monotonic in the Egli-Milner ordering.

Note that applicative languages that allow only for sequential functions are not 'definitionally complete' since they do not allow to directly define all functions by abstraction and recursion which are recursively computable.

Interestingly, most languages and theories for applicative programming are sequential, i.e., they do not consider nonsequential functions and therefore can be restricted to purely sequential operational semantics. For instance, in [66] besides the *if-then-else* and constants all primitive functions are assumed to be naturally extended, i.e. strict. This is why the leftmost-outermost rule is safe in this theory and thus a fixed-point rule. If one considers non-left-strict or even nonsequential functions, the leftmost-outermost rule is *not* safe, of course.

### 5.2. Nonconventional computational models

For the classical "Von-Neumann" architecture, a sequential, stored-program computer, it is very difficult to obtain simple extensions to parallel computations
for the following reasons: The computational model on which the Von-Neumann architecture is fundamentally based is an optimized, strict, innermost evaluation principle for simple tail-recursive, sequential functions (cf. [15, p. 138]). Only in this way the classical notion of program variables can originate (cf. [61]; for a comprehensive discussion see [11]). All extensions to parallel executions such as multi-processors, array processors, or vector processors have to combine the inherently sequential basic computational model in some (rather artificial) way with concepts of parallelism. This leads to an overhead of synchronization primitives such that the actual logical flow of a parallel computation is hidden behind a mess of unimportant details.

Consequently, it seems worthwhile to look for other computational models than that used for the Von-Neumann architecture as theoretical foundations of innovative hardware architectures, especially if one thinks of a large number of processors (possibly on one chip) running in parallel.

For computations which are not driven by straightforward strict sequential control we may, in principle, consider two other evaluation concepts: demand-driven evaluation and data-driven evaluation.

In demand-driven evaluation the evaluation of a subexpression $E_1$ is started during the evaluation of a given expression $E(E_1)$ only if it turns out that the value of $E_1$ is actually needed for the computation of the value of $E(E_1)$. So demand-driven evaluation leads to (quasi-)parallel evaluations only if nonsequential functions occur. In the case of sequential functions it corresponds to a purely sequential technique to handle nonstrict, recursively defined functions, i.e., optimized call-by-name, and nonstrict primitive functions (infinite objects). In particular, it is the basis for concepts such as call-by-need [85], delayed evaluation [83] and finally, lazy evaluation [43].

In data-driven evaluation a subexpression is evaluated as soon as all decisive data are available. It is the basis for the evaluation concept where operations are enabled as soon as sufficient data have arrived. Since, generally, several operations are enabled simultaneously, this technique enforces a lot of parallelism. It allows to treat even nonsequential functions in a straightforward way. In particular, it includes unbounded parallelism, because the maximal number of parallel actions going on in one computation can generally not be bounded by syntactic considerations only.

In the computation rule described in the previous sections, all constructs are treated by data-driven evaluation apart from conditional expressions which (in respect to its alternatives) are evaluated in a demand-driven mode. For a concrete hardware architecture with a high number of processors, the evaluation strategy should always be mixed: data-driven as long as enough processors are available, switching partly (locally) to demand-driven as soon as all processors are busy.

5.3. Reduction

Reduction is probably the most general way to describe operational semantics. It simply corresponds to textual substitution rules. So it describes a concrete
A theory for nondeterminism, parallelism, communication, and concurrency

implementation of the process that reduces by a number of steps the program given in one particular representation into some 'normal form' (cf. "textual substitution machine" and "Herbrand-Kleene-machine" in [11, p. 51]).

Since practically everything, programs, control states, data states, etc., can be represented by terms, every computational model can be abstractly described by a reduction model.

In spite of this, reduction is used in a more technical sense as a sketchword for a number of attempts to get hardware organizations where the term rewriting process for some applicative language is implemented directly. In all these approaches the key question centres around the problem finding an adequate internal representation for the intermediate terms. So, generally, two reduction techniques are distinguished with respect to implementation details: string reduction and graph reduction.

In **string reduction**, the programs are represented by a string of literals and semantic values. Sometimes string reduction is connected to a 'by value' call mechanism [81], because in string reduction multiple occurrences of identifiers are often simply substituted by copies of the corresponding expression and this process is only efficient if the corresponding expressions are reduced to values first (cf. [13, 64]).

In **graph reduction**, one introduces 'references' to the respective expressions for each occurring identifier. So if the value of the corresponding identifier is actually needed, then the expression is evaluated and the resulting value is kept (stored) under the reference so that it can be accessed whenever it is needed (cf. [54, 82]). If no further accesses may occur, the corresponding reference and its value may be deleted ('garbage collection'). Graph reduction is considered just suited to outermost mechanisms (cf. [81, p. 24]).

As shown by the parallel evaluation rule, one may use a string representation and nevertheless have a fully parallel evaluation rule and also nonstrict functions (which generally are the result of outermost evaluation rules). It is the result of \(\lambda\)-notation and of an appropriate combination of techniques of mixed computation (partial evaluation), parallel reduction, and the separation of an application of an \(n\)-ary function into \(n + 1\) processes, in particular into \(n\) slave-processes evaluating the arguments and one master process evaluating the body expression by partial evaluation (cf. also [15, p. 18]).

The enforced computation rule ('speedy' or 'busy' evaluation) as defined in Section 3 is neither a pure outermost nor a pure innermost rule. As shown in [66], innermost rules are not safe for nonstrict functions. However, an innermost substitution step is only unsafe if it is done within an application of a nonstrict function. It seems even dubious to use just the terms 'outermost' and 'innermost' for classifying computation rules. If one allows \(\lambda\)-abstraction, \(f(f(e))\) can equivalently be written \((\lambda x.f(x))(f(e))\). For the latter term, the classification 'innermost call' does not make much sense.

The evaluation rule can also be applied to expressions containing free variables both for objects and functions leading to partial evaluations or mixed computations.
(as suggested in [37]). In particular, one may apply the rule to the bodies of the recursively defined functions to get optimized versions (cf. [82, 11]).

Doubtless, the most decisive problem for a reduction-machine architecture is the efficient representation of the program and its intermediate versions. In the reduction process the program is step-by-step changing its shape and its size such that one may think of a pulsating graph. A reduction machine, however, is a static device which has to store the pulsating graph in the most efficient way.

As far as no infinite, recursively defined objects occur, the program expressions can be represented as trees (cf. Kantorovic-tree in [11]). Certain reduction steps may simplify and decrease the size of the trees, whereas other steps (like unfold for recursively defined functions) may increase the size of the trees. If the evaluation process is to be executed on a network of processors (for instance, on a cellular automaton, cf. [64]), then a net-structure has to be found such that the distribution of the elements of the trees all over the network as well as the communications of the intermediate results can be performed in an efficient way.

If the reduction process is to be executed sequentially, then, of course, demand-driven evaluation techniques are appropriate. Efficient strategies are needed for finding the subexpression where the next reduction steps are to be performed. Turner [82] suggests to use combinators from combinatory logic for this purpose. Berkling [13] uses an organization of three stacks through which linearized versions of the trees are pushed and stepwise evaluate (reduced).

5.4. Data flow concepts

A concept that works with a static graph and is therefore much closer to concrete hardware architectures is the data flow concept. Maybe this is the reason why it has obtained even more attention than the reduction idea. It has been suggested first in the form of the "single assignment approach" in [80]. There, the concept of a program variable, which can dynamically change its value an unbounded number of times, is replaced by identifiers which can be attached to at most one value ("single assignment"). However, this restriction makes it impossible to use iteration and loops as in procedural programs, because, for loops in procedural programs, the repeated assignment to program variables is essential (cf. [11]). The introduction of recursive definitions, however, destroys the static character of the flow graph and leads to reduction concepts. An approach which keeps the static flow graph but allows for a specific form of iteration are data flow graphs with loops. They are proposed in numerous papers (cf. [4, 32, 58]), but at least in the case of nondeterminism, the formal definition of their semantics has so far not been solved in a satisfactory way. Therefore, a completely formal definition of the semantics of a simple graphical data flow language will be given in the sequel.

A data flow program is a directed graph $G = (V, A, I, O, L, OUT)$, where

- $V$ is a finite set of nodes;
- $A$ is a finite set of arcs called streams;
• $I$ and $O$ are functionality functions, $I: V \rightarrow A^*$, $O: A \rightarrow V \cup \{in\}$;
• a labelling function $L: V \rightarrow SPF$;
• a subset $OUT \subseteq A$.

Here, SPF is a set of function identifiers, the corresponding stream processing functions are recursively defined, and for all nodes $x$ the function $L(x)$ is $n$-ary iff $I(x)$ is a word of length $n$, and $O$ is injective.

The arcs $a$ with $O(a) = in$ are called input streams, the arcs in $OUT$ are called output streams.

The meaning of a data flow program is given by the set of mutually recursive stream equations

$$\text{stream } s_a = f_a(s_{a_1}, \ldots, s_{a_n})$$

for each arc $a$ with $a = O(x)$, $f_a = L(x)$, and $(a_1 \ldots a_n) = I(x)$. Thus, for each set of input streams, a recursively defined system of streams is given.

5.4.1. Examples
5.1. Example. With the simple data flow graph of Fig. 3 one associates the mutually recursive system of streams

$$\text{stream } s_1 = \text{merge}(s_{x_0}, s_5), \quad \text{stream } s_2 = \text{nfilt}(s_3, s_1),$$
$$\text{stream } s_3 = C^*(s_1), \quad \text{stream } s_4 = \text{pfilter}(s_3, s_1),$$
$$\text{stream } s_5 = \text{pro}^*(s_2), \quad \text{stream } s_6 = \text{merge}(s_{y_0}, s_8),$$
$$\text{stream } s_7 = \text{nfilt}(s_3, s_6), \quad \text{stream } s_8 = \text{con}^*(s_2, s_7),$$
$$\text{stream } s_9 = \text{pfilter}(s_3, s_1),$$

where

$$\text{funct} \ \text{pfilter} = \lambda c, s: \text{if isempty}(c) \ \text{then} \ \text{empty} \ \text{else}$$
$$\text{if first } c \ \text{then} \ \text{first } s \ \text{& pfilter}(\text{rest } c, \text{rest } s)$$
$$\text{else pfilter}(\text{rest } c, \text{rest } s) \ \text{fi fi},$$

$$\text{funct} \ C^* = \lambda s: \text{if isempty}(s) \ \text{then} \ \text{empty}$$
$$\text{else } C(\text{first } s) \ \text{& } C^*(\text{rest } s) \ \text{fi},$$

Fig. 3.
funct nfilter = \(c, s: \text{if isempty}(c) \text{ then empty else}
\)
\[\text{if first } c \text{ then nfilter}(\text{rest } c, \text{rest } s)
\]
\[\text{else first } c \& \text{nfilter}(\text{rest } c, \text{rest } s) \text{ fi fi},\]

funct pro\* = \(s: \text{if isempty}(s) \text{ then empty}
\)
\[\text{else pro(first } s) \& \text{pro(rest } s) \text{ fi},\]

funct con\* = \(s_1, s_2: \text{if isempty}(s_1) \& \text{isempty}(s_2)
\]
\[\text{then empty}
\]
\[\text{else con(first } s_1, \text{first } s_2)
\]
\[\& \text{con*(rest } s_1, \text{rest } s_2) \text{ fi}.\]

Note that the harmless looking junction

\[
\begin{array}{c}
s \quad s' \\
\downarrow \\
\text{merge}
\end{array}
\]

of arcs is used as an abbreviation for

where merge is the 'nonstrict' form using the \(\vee\)-operator (see merge\(_2\) in Section 2.3.2). For the filtering network of Fig. 4(a) we use the special abbreviation shown in Fig. 4(b) (following the notation of [53]).

The data flow graph implements the function procon:

\[
\text{funct procon = } \lambda \ x, y: \\
\text{if } C(x) \text{ then } y \text{ else procon(pro(x), con(x, y)) fi}.
\]

If we initialize \(s_{x_0} = x \& \bot, s_{y_0} = y \& \bot\), then procon\((x, y) = \text{first } s_y\). For making the network into a correct stream processing function, one has to provide the net with gates making sure that a new argument is not allowed to enter the network before a result has been produced. So the function

\[
\text{funct procon* = } \lambda \ s_x, s_y:
\]
\[\text{procon(first } s_x, \text{first } s_y) \& \text{procon*(rest } s_x, \text{rest } s_y) \]

is implemented by the net in Fig. 5.
A theory for nondeterminism, parallelism, communication, and concurrency

stream \( s_{x_0} = \text{gate}(tt \& s_9, s_{x_r}) \),
stream \( s_{y_0} = \text{gate}(tt \& s_9, s_{y_r}) \),
funct gate = \( \lambda s_1, s_2 : \\
\text{if isempty}(s_1) \text{ or isempty}(s_2) \text{ then empty else first } s_2 \& \text{gate}(\text{rest } s_1, \text{rest } s_2) \). ft.

This frame guarantees, that a 'new' value enters the flow network only when the old computation has been finished. However, the gate can also be combined with the merge to an 'inbound switch';

\[
\begin{array}{c}
\text{inswitch} = A c, s_1, s_2 : \\
\text{if first } c \text{ then first } s_1 \& \text{inswitch}(\text{rest } c, \text{rest } s_1, s_2) \\
\text{else first } s_2 \& \text{inswitch}(\text{rest } c, s_1, \text{rest } s_2) \). ft.
\end{array}
\]

Thus we obtain the deterministic data flow program shown in Fig. 6.

Vice versa, a mutually recursive system of stream equations defines a data flow program. The one for Example 5.5 at the end of Section 5 is shown in Fig. 7.
In particular, the elements of microelectronics can be seen as such data flow networks. Let us consider a flip-flop. A straightforward attempt to model flip-flops by data flow graphs could look as shown in the following example.

5.2. Example *(Flip-flops (bistable circuits)).* A representation of an abstract flip-flop by a data flow network is shown in Fig. 8 (cf. [10, p. 272], where

\[
\text{funct } \text{nor}^* = \lambda s_1, s_2: \neg(\text{first } s_1 \lor \text{first } s_2) \land \text{nor}^*(\text{rest } s_1, \text{rest } s_2).
\]

The semantic definition for data flow nets immediately gives

\[
\text{stream } v_1 = x_1 \land \text{nor}^*(s_1, v_2), \quad \text{stream } v_2 = x_2 \land \text{nor}^*(s_2, v_1).
\]

The boolean values \(x_1\) and \(x_2\) correspond to the initialization of the flip-flop (of course, we assume \(x_1 \neq x_2\)).
A theory for nondeterminism, parallelism, communication, and concurrency

Table 1

<table>
<thead>
<tr>
<th>first $s_1$</th>
<th>first $s_2$</th>
<th>first $v_1$</th>
<th>first $v_2$</th>
<th>first rest $v_1$</th>
<th>first rest $v_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>0</td>
<td>L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>L</td>
<td>0</td>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>L</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
<td>0</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>0</td>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>L</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>0</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>0</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

However, as can be seen from Table 1, the function does not react as a flip-flop since a reset operation always changes the state of the flip-flop (the output) to (0, 0). However, a repeated input gives the right output. This simply means that the flip-flop is idle in switching and that there is some ‘time’ needed to reset the flip-flop. This can be modelled by adding two functions to the flip-flop, just doubling the elements of the input and taking away each second element of the output as is shown in Fig. 9. Here

$$\text{funct double} = \lambda s : \text{first s & first s & double(rest s)}$$

$$\text{funct del} = \lambda s : \text{first s & del(rest rest s)}.$$

Obviously, del(double(s)) = s for all infinite and partial streams s.

Note that doubling the input leads to stable states for all feasible inputs; i.e., using three or more times the same input would not change the resulting state. This is not true for the unfeasible input which may lead to unstable states.
Of course, it is also possible to design hierarchical data flow networks, i.e., networks with nodes which are again (hierarchical) data flow networks. We may even allow recursion here, too, which leads to infinite networks or (following reduction concepts) to dynamic networks.

In data flow networks and especially hierarchical networks, the generalization to nodes with several output arcs seems convenient. Also functions with tuples of streams as result should be considered, which does not cause any additional problems. More examples for stream-processing data flow graphs and networks are given in the following section.

5.5. Properties of stream-processing functions and networks

In a network of communicating agents the nodes are associated with stream-processing functions, which are needed to define the streams of communications in the net. For giving a proper foundation for the specification and classification of stream-processing functions in such networks as well as for a design methodology, different classes of stream-processing functions have to be distinguished and analysed, and their particular role in stream processing has to be figured out. It seems impossible to do this in a comprehensive way in this framework, but nevertheless, a first brief approach is given showing in which way such a classification can be done.

Let $\text{SMF}_n \subseteq \text{STREAM}^n \rightarrow \mathcal{P}(\text{STREAM})$, where $f \in \text{SMF}_n$ iff $f$ is $\subseteq_\text{E}$-monotonic. The resumption $\text{res}$ of an $n$-ary $\subseteq_\text{E}$-monotonic stream-processing function $\text{res} : \text{SMF}_n \times \text{STREAM}^n \rightarrow \text{SMF}_n$ is defined by

$$\text{res}(f, s_1, \ldots, s_n)(s'_1, \ldots, s'_n) = \{ y \in \text{STREAM} \mid \exists x \in \text{STREAM} : x \oplus y \in f(s_1 \oplus s'_1, \ldots, s_n \oplus s'_n)$$

$$\land x \in f(s_1, \ldots, s_n) \land x \text{ is partial} \},$$

where, for $x, y \in \text{STREAM}$,

$$x \oplus y = \begin{cases} x & \text{if } x \text{ is total (finite or infinite),} \\ x' \circ y & \text{if } x = x' \circ (\bot). \end{cases}$$

Obviously, $\text{res}$ and $\oplus$ are homomorphic in the following sense:

$$\text{res}(\text{res}(f, s_1, \ldots, s_n), s'_1, \ldots, s'_n) = \text{res}(f, s_1 \oplus s'_1, \ldots, s_n \oplus s'_n).$$

The validity of this equation immediately follows from the $\subseteq_\text{E}$-monotonicity property of the stream-processing functions. Resumptions can be very conveniently used to discuss properties of $\subseteq_\text{E}$-monotonic stream-processing functions.

At first we want to look at synchronous functions $f \in \text{SMF}_n$. $f$ is called synchronous iff $\text{length}(y) = \text{min}(\text{length}(s_1), \ldots, \text{length}(s_n))$ for all $y \in f(s_1, \ldots, s_n)$, where

$$\text{length}(\varepsilon) = \text{length}(\bot) = 0, \quad \text{length}(\text{ap}(e, s)) = 1 + \text{length}(s).$$

$f$ is called $k$-local if, for all partial $s_i \in \text{STREAM}$ with $\text{length}(s_i) = k$, $1 \leq i \leq n$, $\text{res}(f, s_1, \ldots, s_n) = f$; otherwise, $f$ is called accumulative.
Of course, every \( n \)-ary function \( g \) on atoms defines a \( 1 \)-local stream-processing function \( g^* \) by

\[
\text{funet } g^* = \lambda s_1, \ldots, s_n:
\begin{align*}
\text{if } &\text{isempty}(s_1) \text{ or } \ldots \text{ or } \text{isempty}(s_n) \\
&\text{then empty} \\
\text{else } &g(\text{first } s_1, \ldots, \text{first } s_n) \land g^*(\text{rest } s_1, \ldots, \text{rest } s_n) \text{ fi}
\end{align*}
\]

Given a \((n+1)\)-ary function \( g \) on atoms and a (possibly neutral) element \( e \), we generally immediately get an accumulation function \( \overline{g}_e \) versus

\[
\text{funct } \overline{g}_e = \lambda s_1, \ldots, s_n, h : h(s_1, \ldots, s_n, e), \\
\text{funct } h = \lambda s_1, \ldots, s_n, x: \\
\begin{align*}
\text{if } &\text{isempty}(s_1) \text{ or } \ldots \text{ or } \text{isempty}(s_n) \\
&\text{then empty} \\
\text{else } &x \land h(\text{rest } s_1, \ldots, \text{rest } s_n, \\
&g(\text{first } s_1, \ldots, \text{first } s_n, x)) \text{ fi}.
\end{align*}
\]

If in a stream processing network only \( 1 \)-local functions occur and the graph is free of cycles, then the network just defines a composition of \( 1 \)-local functions and implements a \( 1 \)-local function, too. But if cycles occur, even in the case of \( 1 \)-local functions, the resulting functions may be accumulative.

5.3. Example. We consider the accumulative function \( \overline{\text{add}}_0 \), defined by

\[
\text{funct } \overline{\text{add}}_0 = \lambda s : \text{aux}(s, 0), \\
\text{funct } \text{aux} = \lambda s, x : x \land \text{aux}(\text{rest } s, x + \text{first } s).
\]

This definition can be used in a nonrecursive stream definition: \( \text{stream } \text{sum} = \overline{\text{add}}_0(s) \).
We have \( \overline{\text{add}}_0(s) = 0 \land \overline{\text{add}}^*(s, \overline{\text{add}}_0(s)) \) and we obtain

\[\text{stream } \text{sum} = 0 \land \overline{\text{add}}^*(s, \text{sum}).\]

This corresponds to the data flow program of Fig. 10. In this example the recursion in the definition of \( \overline{\text{add}}_0 \) is transformed into a recursion for the stream \( \text{sum} \) (cf. also [11, p. 295]).

\[
\text{Fig. 10.}
\]

The operator turning a function \( g \) into the stream-processing function \( \overline{g}_e \) may be seen as an example for a \textit{combinator} for writing such functions without explicit recursion as is illustrated in the following example.
5.4. Example. Given the infinite stream \( s = 1 \& s \), which consists of an infinite sequence of 1s, and addition \( \text{add} \) and multiplication \( \text{mult} \), then \( \text{mult}_1(\text{add}_1(s)) \) defines the stream of \( m! \) for \( m = 0, 1, \ldots \).

An appropriate choice of a number of basic functions obviously allows for writing a large number of stream-processing functions simply by combinators. This leads to languages as suggested in [6].

An important example of a stream-processing accumulative function is obtained by embedding the stack-principle into stream processing:

5.5. Example \((\text{LIFO-nodes})\). Since streams are very similar to stacks, it is very easy to program a LIFO-like mechanism:

\[
\begin{align*}
\text{funct} & \quad \text{reverse} = \lambda s: \text{store}(s, \text{empty}) \\
\text{funct} & \quad \text{store} = \lambda s_1, s_2:
\quad \text{if } B(\text{first } s_1) \text{ then } \text{first } s_1 \& \text{clear}(\text{rest } s_1, s_2)
\quad \text{else } \text{store}(\text{rest } s_1, \text{first } s_1 \& s_2) \fi \\
\text{funct} & \quad \text{clear} = \lambda s_1, s_2:
\quad \text{if } \text{isempty}(s_2) \text{ then } \text{reverse}(s_1)
\quad \text{else } \text{first } s_2 \& \text{clear}(s_1, \text{rest } s_2) \fi.
\end{align*}
\]

We assume that \( B \) denotes a deterministic predicate. Obviously, \('\text{reverse}'\) is an accumulative function. For some stream \( s = (s_n \& (\ldots (s_0 \& s') \ldots) \) with \( \neg B(s_i) \) for all \( i, 1 \leq i \leq n \) and some \( s_0 \) with \( B(s_0) \) we have

\[ \text{reverse}(s) = (s_0 \& (s_1 \& (\ldots (s_n \& \text{reverse}(s') \ldots). \]

The function \( \text{reverse} \) can be used to transform general linear recursion into tail-recursion:

\[
\begin{align*}
\text{funct} & \quad f = \lambda x: \text{if } B(x) \text{ then } T(x) \text{ else } g(f(h(x)), x) \fi \\
\text{funct} & \quad \text{down} = \lambda x:
\quad \text{if } B(x) \text{ then } x \& \text{empty} \text{ else } x \& \text{down}(h(x)) \fi, \\
\text{funct} & \quad \text{up} = \lambda s, x, y:
\quad \text{if } \text{first } s = x \text{ then } y
\quad \text{else } \text{up}(\text{rest } s, x, g(y, \text{first rest } s)) \fi.
\end{align*}
\]

We have

\[ f(x) = \text{up}(\text{reverse}(\text{down}(x)), x, T(\text{first reverse}(\text{down}(x)))) , \]

or, eliminating the common subexpression,

\[ f(x) = (\lambda t: \text{up}(t, x, T(\text{first } t))(\text{reverse}(\text{down}(x)))) \]

and

\[ f^*(s) = (\lambda t: \text{up}'(t, s, T(\text{first } t))(\text{reverse}(\text{down}'(s)))) , \]
where

\[ \text{funct} \ down' = \lambda s: \]
\[ \begin{cases} \text{if } B(\text{first } s) \text{ then } \text{first } s \ & \down'(\text{rest } s) \\ \text{else } \text{first } s \ & \down'((h(\text{first } s) \ & \text{rest } s)) \end{cases}, \]

\[ \text{funct} \ up' = \lambda s, s_x, y: \]
\[ \begin{cases} \text{if } \text{first } s = \text{first } s_x \text{ then } y \ & \up'(\text{rest } s, \text{rest } s_x, \ T(\text{first } \text{rest } s)) \\ \text{else } \up'(\text{rest } s, \text{rest } s_x, g(y, \text{first } \text{rest } s)) \end{cases}. \]

In particular, the following equation holds:

\[ \text{reverse}(\down'(e \ & \ s)) = \text{reverse}(\down(e)) \ \text{conc} \ \text{reverse}(\down'(s)). \]

Here, \text{conc} denotes the concatenation of streams.

Based on these functions and some additional simple transformations one obtains a general data flow scheme for implementing linear recursion (see Fig. 11).

5.6. Comparison to the conventional procedural computational model

For understanding the advantages and drawbacks of nonconventional (i.e., non-Von-Neumann) computational models, we now compare the Von-Neumann model with reduction and especially with data flow. We just use an example. Consider the applicative tail-recursive program:

\[ \text{[funct } f = \lambda x: \text{if } C(x) \text{ then } f(F(x)) \text{ else } T(x) \text{ fi, } f(E_0)]. \]

Applying our parallel computation rule to \( f(E_0) \) we may obtain, for instance,

\[ f(E_0) \rightarrow \cdots \]
\[ \rightarrow (\lambda x: \text{if } C(x) \text{ then } f(F(x)) \text{ else } T(x) \text{ fi})(E_0) \]
\[ \rightarrow \text{if } C(E_0) \text{ then } f(F(E_0)) \text{ else } T(E_0) \text{ fi} \rightarrow \cdots \]
\[ \rightarrow T(F(\ldots F(E_0)\ldots)). \]

A corresponding while-program has the form

\[ x := E_0; \ \text{while } C(x) \ \text{do } x := F(x) \ \text{od}; x := T(x); \]

A data flow version may look as depicted in Fig. 12, with

\[ \text{stream } s_x = \text{merge}(E_0 \ & \bot, \ F^*(s_y)), \quad \text{stream } s_y = \text{pfilter}(s_y, s_x), \]
\[ \text{stream } s_b = C^*(s_x), \quad \text{stream } s_t = \text{nfilter}(s_b, s_x), \]
\[ \text{stream } r = T^*(s_x). \]

The control-flow diagram of the procedural version has the form shown in Fig. 13. The procedural program may also be represented by a data flow program with explicit control arcs (see Fig. 14).
In this data flow graph the instructions are labelled by numbers. The instruction stream is tested by the ‘control unit’ for the occurrence of the respective label. The corresponding instruction is activated then and the label of the successive instruction is added to the instruction stream. The actual data flow trivially consists of a merge-node for the variable $x$ with input arcs from all nodes corresponding to
assignments to $x$ and output arcs to all nodes corresponding to statements evaluating $x$. An assignment $x := f(x)$ is then replaced by a node $f^*(\text{filter}(c, s_x))$ where $c$ is the control stream and $s_x$ is the stream of the variable. If the control stream contains $\text{ff}$, the corresponding value of $x$ is simply ignored, otherwise it is taken; the node becomes 'active'.

With the program variable $x$ the stream $s_x$ is associated (the output stream of the respective merge-node), which just represents the stream of values (in order of time) 'flowing through' the variable $x$ during the computation. This stream is of course identical to the stack that the variable substituted via recursion removal (cf. [15, p. 133; 11, 323]). Note that the 'control unit' with its labels defines the structure of
the graph of the control flow diagram, while the data flow is not graphically represented in the control diagram but given by the string representation of the statements. Of course, the data flow program above does not model a classical Von-Neumann architecture where there is only one arithmetic unit and the statements are encoded like data in a stream flowing through this unit. However, this can also be modelled by a data flow graph.

Along the lines outlined above, procedural programs can be turned into data flow programs making control flow and actual data flow explicit. A less schematic treatment, however, eliminating superfluous lines of control flow may lead to much simpler data flow programs, more efficient and easier to comprehend.

The translation of a given data flow program into a classical procedural program is less straightforward. A data flow program corresponds to a network of computing agents connected by streams. Viewing this network as a system of procedural programs, the agents can be represented by sequential processes communicating via send/receive primitives (following the concepts of [16, 19]) working on the streams which then have to be viewed as queues (for more on the duality between streams and queues, see [22]).

Only in very specific cases where the streams between the communicating agents or more precisely, the respective queues can be kept of length one or zero, the system can be substituted by synchronous systems with hand-shaking communication or the streams can even be substituted by simple program variables if we choose an appropriate sequentialization.

6. Concluding remarks: related work and areas of future research

To clarify the notions of multiprogramming and multiprocessing, numerous approaches have been published reaching from initially very machine-oriented concepts to more and more abstract ones (for a brief overview, see [19, 22]). Nowadays a point is reached where we are actually beginning to understand most of the phenomena of concurrent programs including their interrelationships. To this understanding the current approach tries to contribute. In the following, the relation to other works is briefly outlined. Finally, a number of future research topics are briefly sketched that may be based on this approach.

6.1. Related work

Originally, the issues of concurrent programming have been motivated by particular properties of multi-processor machines. Due to the fact that these machines are of the Von-Neumann type, early proposals and investigations were strictly procedure-oriented, centering around the problem of how to protect and synchronize the access to shared memory, which was considered as the only way of communication between programs executed in parallel. First attempts to overcome these difficulties can be found in the single assignment approach (cf. [80]). This proposal
was completed to data flow language concepts (cf. [32]), based on the demand-driven evaluation (cf. also [6]). Numerous papers have been published on this issue, few of them, however, containing much formal foundation. Friedman and Wise [40] suggest a language related to ours. There already the concepts of lazy evaluation are incorporated to obtain a LISP-extension which is suitable for applicative multiprogramming. There ‘ferns’ are used instead of streams. Other approaches use ‘tagged’ values (cf. [59, 60]) or ‘scenarios’ (cf. [14]).

A far developed theory is Milner’s Calculus of Communicating Systems (CCS, cf. [69, 68, 70, 71]). In CCS a communication mechanism is integrated into an applicative programming language. The communication in CCS follows the rendezvous principle (like CSP in [49]) and therefore can be seen as the applicative counterpart of CSP.

A paper with much impact on the field of concurrent programming is [52], where an approach is outlined which is very similar to the streams described in Section 2.3 and Section 5 (cf. also [63]). However, Kahn’s approach does not include nondeterminism nor concurrency in the sense of Section 6 (cf. the discussion at the end of [53]).

In the field of nondeterministic applicative programming languages, several papers on fixed-point theory have been published (cf., for instance, [45, 46, 47, 7, 3, 26, 73]). For surprisingly few approaches to multiprogramming, however, attempts have been undertaken to give a fixed-point semantics (cf., for instance, [4, 49, 60]).

We prefer a fixed-point oriented approach for the following reasons: First, the joint consideration of operational and mathematical semantics gives valuable insights into the structure of the concepts. Second, we can always check whether our intuition actually leads to computable, formally sound semantics. Third, the technical difficulty and complexity of particular concepts gives hints on their comprehensive complexity and also on the difficulties to find appropriate methodologies for the design and verification of such programs.

Of course, the semantics of the language for applicative multiprogramming could be described also by algebraic means along the lines of [23]. However, in addition to the reasons cited above, it seems worthwhile to develop the concepts of multiprogramming from the nowadays well-understood concepts of sequential programming, for which fixed-point based concepts of denotational semantics are well-understood.

Hence, in the preceding sections a strictly fixed-point oriented approach to the concepts of nondeterminism, parallelism, communication, and concurrency has been undertaken. Not all results are satisfactory yet although most of the important notions can be described properly and, to some extent, fit in naturally with the framework of fixed-point theory.

6.2. Areas of future research

There are several severe reasons for increasing investigation efforts in the field of applicative programming along the lines pointed out in this paper. Besides economic
and political motivations concerned with the enormous amount of financial efforts devoted in some quarters to this question (cf. [50]), there are quite a number of practical, technical, and scientific reasons for such efforts: It seems inevitable that concurrency in nets and networks will play one of the most prominent roles in future information processing, be it in distributed systems of components geographically spread all over the world or in distributed systems completely located on a single chip in VLSI. In both cases the problem of the logical design and specification of such systems, their analysis, verification, and development will be the most decisive challenge in future information processing. And such an approach will include systems which, in their final form, are given in (machine-oriented) software as well as systems which are actually realized in hardware. In particular, the interface between hardware and software realizations should be made flexible such that a variable transition from software to hardware becomes possible. So hardware design can finally be completely determined by the needs of software design which may be a step to overcome the so-called software crisis.

**Software engineering methodology:** According to the theory of flow graphs a distributed system can be designed as a network by specifying the streams in the arcs and the stream-processing functions in the nodes. Appropriate specification techniques are to be developed going beyond fixed-point definitions by recursion. the network fixes the overall structure of the distributed system. Then the agents located in the nodes can be independently realized again by networks (leading to a hierarchy of networks), by applicative stream-processing functions (i.e., as a data flow machine with reduction machines as computing nodes) or by procedural sequential communicating processes (using send/receive primitives as defined in [19]). Appropriate transformation methods may be used to perform this realization in a proper way. Using the definitions of [19], considering the streams as queues at the procedural level (cf. [27]) the verification techniques of Gries and Owicki can be applied to prove partial correctness of the resulting programs (if not already guaranteed by the development) independently since the proofs can be kept interference-free. Another issue concerns transformation rules for transforming applicative and procedural programs into data flow programs and vice versa.

**Languages for applicative multiprogramming:** Although powerful enough to discuss the theoretical foundations of applicative multiprogramming, the language used in this paper is far from being a sufficient tool for practical applicative multiprogramming. Tools for type denotations, hierarchical design, and notational variants are missing. The introduction of alternative and generalized data structures with non-strict operations and infinite objects is one additional issue. Another important issue concerns the translation and compilation of purely functional languages into data flow graphs.

**Theoretical foundations:** Of course, there are numerous theoretical problems and questions still open. One important issue concerns the axiomatization of applicative multiprograms by assertions. The 'two-level' fixed-point theory of Section 3 precisely mirrors two concepts of program verification, the first 'approximated' semantics
using $\Box$ instead of $\triangledown$ is sufficient to prove partial correctness, i.e., an axiomatization of partial correctness can be based on this simple semantics. The final semantics can be used as a basis for proof methods for total correctness, in particular for termination, absence of deadlock, etc.

So far the language is well-suited to describe just loosely coupled ('asynchronous') systems since all communications may take place as soon as the sender is ready without taking into account the situation of the receiver. Another possibility is found in the so-called 'rendezvous'-concept, where handshaking communication is used, leading to tightly coupled ('synchronous') systems (cf. [71]). The relationships between these two concepts have to be investigated. Roughly speaking, a loosely coupled system has, under certain restrictions, tightly coupled computations. So tightly coupled systems should be obtainable from loosely coupled ones by restriction.

The incorporation of real time considerations and of fairness concepts is another issue of further theoretical investigations. One possibility is 'hiatonization' (the word is due to Wadge) which means the introduction of dummy data such that in every computation step a stream-processing function produces either a real atom or a dummy date. This leads to strongly communicating processes, where, as is well-known, a number of problems do not appear any longer (cf. the introduction of "silent communication" in [70] and the condition "is guarded" in [71, p. 10]) just as for real-time processes (cf. [17]).

An alternative to modelling communication by mutually recursively defined streams is the direct incorporation of communication primitives into an applicative language (cf. [70]). The differences and similarities of these two approaches have to be further explored.

A further question concerns the suitability of languages for applicative multiprogramming as semantic models for the formal definition of procedural languages for multiprogramming by denotational techniques.

Innovative hardware architecture and VLSI: The direct implementation, i.e., interpretation of applicative languages by innovative hardware based on LSI- and VLSI-design leads to a high number of interesting investigations such as
- interpreters as data flow networks,
- flexible hardware structures (networks) for tree representations as reduction machines and cellular automata,
- multifunction special purpose chips (logical design of special purpose hardware),
- storage as networks (active storage),
- networks with uniform multifunction chips (cellular automata).

All these investigations comprise hardware design (at least the logical part of it) as an integrated part of software development. The long-range objective is the development of information processing systems not bottom-up, designing software for a given, already fixed hardware, but rather top-down designing software just for solving a specific problem and then constructing special purpose hardware for this particular software.
Acknowledgment and general remarks


Discussions with numerous people, in particular with Jack Dennis, Tony Hoare, Peter Lauer, Gordon Plotkin, Michel Sintzoff, Philip Treleaven, and David Turner, have helped to clarify the work considerably. First and foremost, however, I gratefully acknowledge stimulating, encouraging conversations with F.L. Bauer and G. Seegmüller. Certainly, my long joint cooperation with M. Wirsing has definitely influenced the approach. Moreover, I am indebted to B. Möller and P. Pepper for a number of valuable remarks.

Basically, the work has grown out of investigations performed within the Sonderforschungsbereich 49, Programmiertechnik. So I would like to thank my colleagues of the projects A2 and CIP, as well as the colleagues of the project A1 for numerous discussions. Remarks by André Arnold, Samson Abramsky, Friederike Nickl, and Thomas Streicher have helped to correct short-comings in former versions.

References

A theory for nondeterminism, parallelism, communication, and concurrency


[58] P.R. Kosinski, A data flow language for operating systems programming, SIGPLAN Notices 8(9) (1973) 89–94.


A theory for nondeterminism, parallelism, communication, and concurrency