A Theory for Nondeterminism, Parallelism, Communication, and Concurrency

Manfred Broy

Abstract

The theory of multidomains is given. A fixed point semantics for a nondeterministic, applicative programming language working with finite and infinite streams is based on it. An operational semantics for the parallel evaluation of expressions of this language is defined in the form of the term rewrite rules. Systems of expressions communicating by streams are introduced by mutually recursive fixed point equations for streams. The fixed point theory on multidomains is complemented by a second theory allowing for the definition of McCarthy's ambiguity operator which is needed for the definition of nonstrict merging. The resulting language can be taken as foundation of semantics for data flow languages as well as for discussing nonconventional hardware architectures such as reduction machines and data flow machines. Finally directions of future research are outlined.

Acknowledgement and Preliminary Remarks

The main part of this work was first presented at the International Summer School on "Theoretical Foundations of Programming Methodology", Marktoberdorf, August 1981. A shortened version completely based on combinations of various powerdomains instead of multidomains will appear at the IFIP TC 2 Working Conference on "Formal Description of Programming Concepts II", Garmisch-Patenkirchen, June 1982.

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 like to thank my colleagues of the projects A2 and CIP, as well as the colleagues of the project A1 for numerous discussions.

Table of Contents

1. Introduction

The last two decades of computer science are characterised 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 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 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 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 behavior 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* of nondeterminism (cf. /Kennaway, Hoare 80/, /Broy, Wirsing 81a/) 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 the solutions to which may give proper formal foundations for software and hardware architectures for "computers of the fifth generation" (cf. /Japan 81/). In the sequel a brief overview is 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 an 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 beares 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. /Plotkin 76/, /Smyth 78/), because there the "Egli-Milner ordering" represents only a quasiordering. So a domain construction is needed that, in contrast to the powerdomain, works sufficiently well in this case. As a partial solution to this problem the multidomain construction is defined. As a most fundamental example the multidomain of finite partial, finite total and infinite streams is introduced.

Systems of Expressions Communicating by Streams: Formally a system of communicating expressions is defined by a mutually recursive system of fixed point equations for streams with the resp. stream processing functions on the right-hand side. In order to obtain not always trivially undefined as fixed points of such systems one has to use nonstrict constructor functions for streams and additionally specific computation rules. This includes in straightforward manner infinite objects. 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 (multi-)sets of deterministic equations for streams rather than as equations for (multi-)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 resp. nondeterminstic 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 following question: how good have the approximations for the input 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 into applicative languages it is necessary to include McCarthy's ambiguity operator. Such an operator is necessary for defining a nonstrict, nonsequential merge function for streams such that merge($\perp$, s) = merge(s, $\perp$) = s and merge(x1&s1, x2&s2) = ((x1&merge(s1, x2&s2)) $\square$ (x2&merge(x1&s1, s2))) which is an inevitable prerequisite for networks of communicating agents.

Thus concurrency does not only require a free straightforward choice between concurrent computations, but it reqires a choice depending on particular termination properties of the concurrent computations. This brings all the problems of unbounded nondeterminism, its nonconti-nuity as found in the fairness discussion (cf. /Park 80/, /Broy 81a/, /Apt, Plotkin 81/) and even worse problems concerning monotonicity. For solving these problems a fixed point for recursive equations containing the ambiguity operator is 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 sub(multi)set 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 /Vuillemin 75/, 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 Σ^1_1.

Computational Models: It is the very nature of notions like "parallism" 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 rewrite rules). In addition to classical operational semantics the following five aspects are of major importance when dealing with multiprograms:

(1) Nondeterminism implies nonconfluent term rewrite 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.

(3) A communicating process consumes its input piecewise and produces its output piecewise. So respective computation rules for communication have to be used.

(4) 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.

(5) 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 end 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 resp. expressions and all formal parameters are replaced by the resp. 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 also can be taken as the basis for a software engineering discipline including the specification, development, and verification of software for such systems. It is briefly discussed and outlined how the language defined in this paper can be taken as formal foundations for such concepts. In particular a formal definition for the semantics of a data flow language is 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 sequential stored program architectures appears just as an extreme case.

In the following an attempt is undertaken to give a solution to these problems in one integrated approach completely based on fixed point theory. In particular the paper is structured as follows.

Section 2 establishes the theory of multidomains constructed by the ideal completion of the set of finite multisets over the set of finite elements of an given algebraic complete partial ordering (cpo). As an important example the multidomain of streams is considered.

Section 3 gives a simple applicative nondeterministic language and its mathematical semantics based on multidomains and fixed point theory.

Section 4 considers a nondeterministic computation rule for the parallel evaluation of nondeterministic recursive programs modelling data driven reduction.

Section 5 extends the possibilities of communication between processes that evaluate expressions by the introduction of expressions which mutually communicate by streams. Thus networks of expressions communicating by streams can be defined. The mathematical semantics is again based on fixed point theory. The operational semantics is given by term rewrite rules.

Section 6 introduces an ambiguity operator leading to real concurrency. This operator allows to suppress certain "less defined" nondeterministic alternatives and in particular serves as a basis for defining disjunctive (alternative, parallel, multiple) waiting. However, the introduction of this operator causes a radical change of our domain and our notion of computability. Besides that the notions of strictness and sequentiality of functions are discussed in detail and related to classical notions of partial recursive functions.

Section 7 investigates and compares several nonconventional computational models like reduction and data flow. For data flow networks a formal semantics is defined. Several properties and examples for stream-processing networks and functions are discussed. Finally a comparison to classical procedural programs is given.

Section 8 contains a brief discussion of related work. A list of topics of possible directions of future research is presented.

2. Multidomains

For giving semantics to nondeterministic programs, /Plotkin 76/ suggests the use of *powerdomains*, which are particular subsets of the given domains representing the sets of possible values (cf. also /Smyth 78/). However, for nonflat domains the well-known Egli-Milner ordering does not work, since the induced ordering generally is only a quasiordering. Therefore in /Lehmann 76/ as an alternative to fixed point theory, category theory is suggested for giving meaning to nondeterministic programs.

In the sequel a different approach is given which uses multisets (cf. for instance /Dershowitz, Manna 79/) as suitable representations of the (multi-)set of possible results. Roughly speaking, a multiset is a "set, where multiple occurrences of elements are allowed". A nondeterministic program can be viewed as a finitary, possibly infinite tree. In the powerdomain construction the set of terminal nodes (including $\perp$ for infinite paths) is

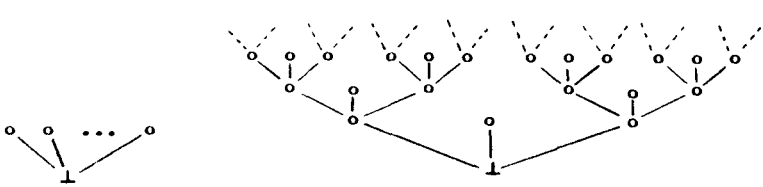


Fig 1. A flat and a nonflat domain (corresponding to the domain $\{tt, ff\}^S$ of streams of truth-values

associated with the program. In our approach of multidomains, the multiset of terminal nodes is associated with the program (including $\perp$ for every infinite decision path).

All proofs are given in the appendix.

2.1 Multisets

Given a set S, a multiset M over S is a total mapping

$$M : S \to \mathbb{N}^+$$

where $\mathbb{N}^+ =_{def} \mathbb{N} \cup \{\infty\}$. The set of multisets over a given set S is denoted by $\mathcal{M}(S)$. Every multiset M defines also a set

$$SET(M) = \{x \in S : M(x) \geq 1\}$$

The cardinality of a multiset M is denoted by $|M|$ where

$$|M| =_{def} \sum_{x \in SET(M)} M(x)$$

Trivially, every set $S1 \subseteq S$ denotes a multiset M with

$$M(x) = \begin{cases} 1 & x \in S1 \\ 0 & \text{otherwise} \end{cases}$$

Therefore we often use the set-notation for representing multisets.

For multisets $M1$, $M2$ we define the *multiset sum* $M1 \overset{\cdot}{+} M2$ and *multiset difference* $M1 \overset{\cdot}{-} M2$ by

$$(M1 \overset{\cdot}{+} M2)(x) = M1(x) + M2(x)$$

$$(M1 \dot{-} M2)\ (x) = \begin{cases} M1(x) - M2(x) & \text{if } M1(x) > M2(x) \\ 0 & \text{otherwise} \end{cases}$$

The set union and set intersection are extended to multisets by

$$(M1 \cup M2)\ (x) = \max \{M1(x), M2(x)\}$$

$$(M1 \cap M2)\ (x) = \min \{M1(x), M2(x)\}$$

Given a multiset M, we define a multiset $\biguplus_{x \in M} F(x)$ where $F: S \to \mathcal{M}(S)$

$$\left(\biguplus_{x \in M} F(x) \right)(y) =_{def} \sum_{x \in SET(M)} \sum_{i=1}^{M(x)} F(x)(y)$$

In analogy to the set notation, we write

$$M1 \subseteq M2 \quad \text{iff} \quad \forall x \in S: \quad M1(x) \leq M2(x)$$

Multisets can be viewed as a "natural" extension of set theory. Although it seems interesting to study the properties of multisets much more deeply, we restrict ourselves to the definition of multidomains as particular subsets of the set of multisets.

2.2 Multidomains

In this section multisets over a complete countable algebraic partially ordered set are considered.

A poset $(DOM, \sqsubseteq)$ is a cpo (complete partially ordered set) iff

(i) DOM has a least element $\perp$

(ii) every directed subset X has a least upper bound lub(X)

An element $a \in DOM$ is called finite if for every directed subset $X \subseteq DOM$: $a \sqsubseteq lub(X) \Rightarrow \exists\, x \in X : a \sqsubseteq x$

DOM is called countably algebraic if

(i) the set of finite elements is countable ,

(ii) every element in DOM is the lub of a directed set of finite elements.

We shall refer to countably algebraic cpo's simply as domains.

In a domain DOM an element $x \in DOM$ is called partial if $\exists\, y \in DOM, y \neq x : x \sqsubseteq y$, total if x is not partial, i.e. x is maximal in DOM, infinite if x is not finite.

As a generalisation of the well-known Egli-Milner ordering on sets over DOM (cf. /Egli 75/, /Milner 73/) the relation $\sqsubseteq$ is defined on $M(DOM)$ by :

Let $M1, M2 \in M(DOM)$:

$M1 \sqsubseteq M2$ iff $\exists\ G : SET(M1) \to M(DOM)$:

(1) $\forall\, x \in SET(M1) : M1(x) \leq |G(x)| \ \wedge$
$\forall\, y \in SET(G(x)) : x \sqsubseteq y$

(2) $\biguplus_{y \in SET(M1)} G(y) = M2$

Intuitively speaking, $M1 \sqsubseteq M2$ holds, i.e. M1 approximates M2, if M2 can be obtained from M1 by substituting every element x of M1 by nonempty multisets G(x) which consist only of elements which can be approximated by x . This immediately gives:

If $M1 \sqsubseteq M2$, then $SET(M1) \sqsubseteq_{EM} SET(M2)$ where the Egli-Milner ordering (cf. /Egli 75/, /Milner 73/, /de Bakker 76/) is

defined by

$$S1 \sqsubseteq_{EM} S2 \text{ iff } (\forall x \in S1\ \exists y \in S2 : x \sqsubseteq y) \wedge (\forall y \in S2\ \exists x \in S1 : x \sqsubseteq y).$$

Like the Egli-Milner ordering, the relation $\sqsubseteq$ in general does not define an ordering but only a quasiordering.

A multiset M can also be represented by a set S_M of pairs. S_M may be defined by

$$S_M = \{(n, x) \in \mathbb{N} \times DOM : 1 \leq n \leq M(x)\}$$

Lemma: $M1 \sqsubseteq M2$ iff $\exists H : S_{M2} \to S_{M1}$:

(1) H is surjective

(2) $\forall (i, y) \in S_{M2} : (j,x) = H(i,y) \Rightarrow x \sqsubseteq y$

(H is right-monotonic) □

Lemma: The relation "$\sqsubseteq$" defines a quasiordering on $M(DOM)$. □

However on a subset of $M(DOM)$ the relation "$\sqsubseteq$" even defines a cpo . To isolate this subset the cpo is constructed in the classical way as ideal completion of the finite elements.

Following the ideal-theory of /Scott 80b/, at first the finite elements of $M(DOM)$ are considered, i.e. the finite nonempty multisets of finite elements of DOM. The set of these multisets is denoted by $FM(FDOM)$ where $FDOM \subseteq DOM$ denotes the finite elements in DOM and for each set S $FM(S)$ denotes the multisets represented by the functions

$f: S \to \mathbb{N}$ where $\{x \in DOM : f(x) > 0\}$ is finite and nonempty.

In contrast to the powerdomain, where the Egli-Milner ordering generally defines only a quasi-ordering even on finite sets of finite elements, the relation $\underline{\sqsubseteq}$ denotes a partial ordering on $FM(FDOM)$.

Lemma: $(FM(FDOM), \underline{\sqsubseteq})$ forms a poset with minimal element $\{\bot\}$ □

Now $FM(FDOM)$ is extended to a complete partial ordering (cpo) by taking its ideal completion. A nonempty subset $I \subseteq FM(FDOM)$ is called an ideal iff

$$(1)\ M1 \underline{\sqsubseteq} M2 \wedge M2 \in I \Rightarrow M1 \in I$$
$$(2)\ M1, M2 \in I \Rightarrow \exists\, M3 \in I:\ M1 \underline{\sqsubseteq} M3 \wedge M2 \underline{\sqsubseteq} M3$$

So I is an ideal if it is (1) downward closed and (2) directed. Let $\mathcal{ID}(FDOM)$ denote the set of ideals. As well-known $(\mathcal{ID}(FDOM), \subseteq)$ forms a countably algebraic cpo.

Since it seems more convenient to talk about multisets than about ideals, we isolate a subset $\mathcal{MD}(DOM)$ of $M(DOM)$, such that $(\mathcal{MD}(DOM), \underline{\sqsubseteq})$ is isomorphic to $(\mathcal{ID}(FDOM), \subseteq)$. So every ideal is represented by one particular multiset. For every $M \in M(DOM)$ we define

$$I_M = \{M1 \in FM(FDOM) : M1 \underline{\sqsubseteq} M\}$$

If $M \in FM(FDOM)$ trivially I_M forms an ideal.

On the other hand we may associate a multiset M_I with every subset $I \subseteq FM(FDOM)$ by

$$M_I(x) = \operatorname*{glb}_{\substack{M1 \in I \\ y \underline{\sqsubseteq} x \\ y \in FDOM}} \ \operatorname*{lub}\ \{M2(z) : M1 \underline{\sqsubseteq} M2 \in I \wedge y \underline{\sqsubseteq} z \underline{\sqsubseteq} x\}$$

Here lub and glb denote the least upper bound and greatest lower bound in IN^+.

Lemma: (1) For every ideal I with $M = M_I$ we have $I = I_M$.
(2) For ideals I1, I2 we have $I1 \subseteq I2 \Rightarrow M_{I1} \sqsubseteq M_{I2}$ □

Accordingly we define the multidomain $(\mathcal{MD}(DOM), \sqsubseteq)$ by

$$\mathcal{MD}(DOM) = \{M \in \mathcal{M}(DOM): \exists\, I \in \mathcal{ID}(FDOM): M_I = M \wedge I_M = I\}$$

Of course $\mathcal{FM}(FDOM) \subseteq \mathcal{MD}(DOM)$.

Since the multidomain is not closed under the usual multiset sum, we define a new sum operator for elements from the multidomain:

$$M_{I1} \oplus' M_{I2} = M_I \text{ where } I = \{M \in \mathcal{FM}(FDOM): \exists M1 \in I1,\ M2 \in I2: M \sqsubseteq M1 \oplus M2\}$$

Obiviously on finite multisets:

$$M1 \sqsubseteq M1' \wedge M2 \sqsubseteq M2' \Rightarrow M1 + M2 \sqsubseteq M1' + M2'$$

Similary we define for $I \in \mathcal{ID}(FDOM)$, f: $DOM \to \mathcal{ID}(FDOM)$

$$\bigoplus_{y \in M_I}{}' M_{f(y)} = M_{I'} \text{ where}$$

$$I' = \{M \in \mathcal{FM}(FDOM) : \exists M' \in I, M_y \in f(y): M \sqsubseteq \sum_{y \in M'} M_y\}$$

For simplicity we write $\oplus$ for $\oplus'$ in the sequel and use $\oplus'$ also for ideals.

Due to the results above the following proposition holds:

Lemma: The multidomain $MD(DOM)$ forms a countably algebraic cpo, i.e. it forms a domain. □

Note that the particular subset $MD(DOM)$ of $M(DOM)$ is determined by the fact, that only multisets are to be considered, which are approximable by finite multisets of finite elements, and the notion of approximability is determined by the particular choice of the ordering, which is defined such that the most important constructs which one wants to have in a programming language, such as function application, finite choice, and conditional are monotonic and continuous.

Now the continuity of some basic functions is established.

Lemma: Function extension is continuous:

Let the function

$$f : DOM^n \to DOM$$

be given and the function

$$F : DOM^n \to MD(DOM)$$

be defined by

$$F(x) = \{f(x)\}$$

then if

(1) f is monotonic, then F is monotonic, too,

(2) f is continuous, then F is continuous, too. □

Lemma:

The multiset-sum is monotonic and continuous. □

As is well-known, the ordering on multidomains induces an ordering on multiple-valued functions. Let

$$f, g : DOM1 \rightarrow MD(DOM2)$$

then we define

$$f \sqsubseteq g \quad \text{if} \quad \forall x \in DOM1 \quad f(x) \sqsubseteq g(x)$$

Let Γ be a functional

$$\Gamma : (DOM1 \rightarrow MD(DOM2)) \rightarrow (DOM1 \rightarrow MD(DOM2))$$

then Γ is called monotonic, if

$$f \sqsubseteq g \text{ implies } \Gamma[f] \sqsubseteq \Gamma[g]$$

Γ is called continuous, if for every chain of continuous functions $\{f_i\}_{i \in \mathbb{N}}$:

$$\Gamma[\text{lub}\{f_i\}] = \text{lub}\{\Gamma[f_i]\}$$

Lemma:

Let $\Gamma : (DOM \rightarrow MD(DOM)) \rightarrow (DOM \rightarrow MD(DOM))$

be defined by

$$\Gamma[f](x) = \bigoplus_{y \in T[f](x)} f(y)$$

where $T : (DOM \rightarrow MD(DOM)) \rightarrow DOM \rightarrow MD(DOM))$

then Γ is

(1) monotonic, provided T is monotonic
(2) continuous, provided T is continuous. □

To give an example how the multidomain construction works in a particular case, the multidomain of streams is considered in the following section.

2.3 The Multidomain of Streams

As an important example for a nonflat, algebraic domain for multiprogramming the domain of *streams* is considered (cf. /Landin 65/, /Burge 75/, /Dennis, Weng 79/). Let a countable set A of atoms be given, such that $\perp \notin A$. As usual $A^{\perp}$ denotes the corresponding flat domain.

The domain A^S of streams over A is defined by

$$A^S = A^* \cup (A^* \times \{\perp\}) \cup A^{\infty}$$

Here A^* denotes the set of finite streams, i.e. finite sequences of atoms from A, and includes ε, 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*. A^{∞} 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:

$$\begin{aligned} \text{ap} &: A^{\perp} \times A^S \to A^S \\ \text{rest} &: A^S \to A^S \\ \text{first} &: A^S \to A^{\perp} \\ \text{isempty} &: A^S \to \{tt, ff\}^{\perp} \end{aligned}$$

defined by

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

Let $a \in A$, $s \in A^S$, $s' = \langle a \rangle \circ s$, then

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

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

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

By $\langle a \rangle$ the one-element sequence is denoted, and by $s \circ s'$ the concatenation of two sequences. 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 $s' \in A^S$. Note, however, that A^S is not closed with respect to concatenation since for $s \in A^* \times \{\bot\}$ and $s' \in A^* \setminus \{\varepsilon\}$: $s \circ s' \notin A^S$.

To make A^S into a domain, an ordering is needed. So we define for $s1, s2 \in A^S$:

$$s1 \sqsubseteq s2 \text{ iff } s1 = s2 \text{ or } \exists\, s3, s4 \in A^S \text{ such that } s1 = s3 \circ \langle \bot \rangle \text{ and } s2 = s3 \circ s4$$

Intuitively, $s1 \sqsubseteq s2$ holds, i.e. $s1$ "approximates" $s2$, if $s1 = s2$ or if $s1$ is a partial stream which is a prefix of $s2$ if $\bot$ is dropped at the end of $s1$. With this ordering A^S forms a countable algebraic cpo. Note that $A^\bot$ can be viewed as a proper subdomain of A^S.

Lemma: The functions ap, rest, top, isempty are monotonic and continuous.

Proof: omitted □

According to the results of section 2.2, the functions ap, rest, top, isempty can be trivially extended to multisets of streams. $MD(A^S)$ is called the *multidomain of streams*.

Note: One might also use mappings $DOM \to [0,1]$ instead of multisets and interprete this as the probability of a result (cf. /Francez, Rodeh 80/). Note, however, that this would lay severe restrictions on the implementations to assure that all probabilities are properly realized.

end of note

3. A Simple Language for Applicative Multiprogramming

In this section we define a simple nondeterministic programming language and its mathematical and operational semantics. By studying specific computation rules we analyse the possibilities of inherent parallelism leading to an (operational) data driven reduction semantics.

3.1 Syntax

The syntax of the language is close to λ-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.

```
< program >::=⌈{funct < function identifier > ≡ < funct abstract >,}*
                < expr > ⌋
< expr > ::= < funct appl > | < cond > | < choice> |
             < object >
< funct appl > ::= < function > ({< expr > {, < expr > }* })
< cond > ::= if < expr > then < expr > else < expr > fi
< choice > ::= < expr > [] < expr >
< object > ::= < primitive object > | < identifier >
< funct abstract > ::= λ {< identifier > {, < identifier >}* } :
                           < expr >
< function > ::= < funct abstract > | < function identifier > |
                 < primitive function >
```

Here we assume a domain DOM of semantic values called primitive objects including ff and 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 partial function

$$\tilde{g} : DOM^n \to DOM$$

where $\tilde{g}(x_1, \ldots, x_n) = \perp$, if one of the x_i is partial, and g is strict, monotonic, and continuous.

An expression is closed, if no free identifiers occur in it. An expression is called primitive, if it is a term built from primitive functions and objects only.

3.2 Mathematical Semantics

In this section we define a mathematical semantics by giving a function (cf. /Broy et al. 78/):

$$B : EXP \to MD(DOM)$$

where EXP denotes the set of closed expressions, i.e. the set of expressions in which no free identifiers or nonprimitive function symbols occur.

$$B[\underline{\text{if}}\ C\ \underline{\text{then}}\ E1\ \underline{\text{else}}\ E2\ \underline{\text{fi}}] = \left(\bigoplus_{tt \in B[C]} B[E1]\right) \oplus \left(\bigoplus_{ff \in B[C]} B[E2]\right) \oplus \left(\bigoplus_{\perp \in B[C]} \{\perp\}\right),$$

$$B[E1 \,[\!]\, E2] = B[E1] \oplus B[E2],$$

$$B[(\lambda\, x_1, \ldots, x_n : E_{n+1})\ (E_1, \ldots, E_n)] = \bigoplus_{e_1 \in B[E_1]} \cdots \bigoplus_{e_n \in B[E_n]} B[E_{n+1}\ [e_1/x_1, \ldots, e_n/x_n]]$$

$$B[e] = \{e\} \quad \text{for } e \in DOM$$

$$B[g(E_1, \ldots, E_n)] = \biguplus_{e_1 \in B[E_1]} \cdots \biguplus_{e_n \in B[E_n]} \{\tilde{g}(e_1, \ldots, e_n)\}$$

for n-ary $g \in P$

Given an expression E in which the nonprimitive function symbols $f_1, \ldots, f_n$ occur in n_i-ary function applications, then for each set

$\tilde{f}_1, \ldots, \tilde{f}_n$ of n_i-ary functions $\tilde{f}_i : DOM^{n_i} \to MD(DOM)$ by

$$B[f_i(E_1, \ldots, E_{n_i})] = \biguplus_{e_1 \in B[E_1]} \cdots \biguplus_{e_n \in B[E_{n_i}]} \tilde{f}_i(e_1, \ldots, e_{n_i})$$

the semantics $B[E]$ is fixed. To express that the function $\tilde{f}_i$ is to be taken for the symbol f_i we often write

$$B[E[\tilde{f}_1/f_1, \ldots, \tilde{f}_n/f_n]]$$

Given an expression E in which the free identifiers $x_1, \ldots, x_n$ occur, in principle one might consider arbitrary closed expressions $E_1, \ldots, E_n$ and the multiset $B[E[E_1/x_1, \ldots, E_n/x_n]]$. However, since we would like to consider free identifiers as identifiers for values rather than as identifiers for expressions, we restrict ourselves to the substitution of semantic values for free identifiers (cf. the remark on call-by-value versus call-by-name in section 4).

According to the propositions in section 2, all the functions defined so far are monotonic and continuous and therefore we may define the semantics of programs:

$$B[\lceil \underline{\text{funct}}\ f_1 = F_1, \ldots, \underline{\text{funct}}\ f_n = F_n,\ E \rfloor] = B[E[\tilde{f}_1/f_1, \ldots, \tilde{f}_n/f_n]]$$

where

$$\tilde{f}_i,\ f_i^{(j)} : DOM^{n_i} \to MD(DOM)$$

are defined by

$$f_i^{(o)}(e_1,\ldots,e_{n_i}) = \{\bot\}$$

$$f_i^{(j+1)}(e_1,\ldots,e_{n_i}) = B[(F_i[f_1^{(j)}/f_1,\ldots,f_n^{(j)}/f_n])(e_1,\ldots,e_{n_i})]$$

$$\tilde{f}_i = \operatorname{lub}_{j\in\mathbb{N}} \{f_i^{(j)}\}$$

The continuity of the operations of our language guarantees

$$(*)\quad \tilde{f}_i(e_1,\ldots,e_{n_i}) = B[F_i[\tilde{f}_1/f_1,\ldots,\tilde{f}_n/f_n]\ (e_1,\ldots,e_{n_i})]$$

and that $\tilde{f}_1,\ldots,\tilde{f}_n$ are the least fixed points of the equation (*).

Note, that for the language considered so far it is not necessary to work with multidomains, if DOM is flat, since powerdomains are sufficient for flat domains (cf. /Broy et al. 78/). However, working with multidomains, we need not necessarily restrict ourselves to flat domains.

4. Operational Semantics: A Nondeterministic Computation Rule for Parallel Evaluation

Now we consider the program

$$\lceil \underline{\text{funct}}\ f_1 = F_1, \ldots,\ \underline{\text{funct}}\ f_n = F_n,\ E \rfloor$$

and give a number of rewrite rules which serve as basic computation steps.

To give a proper definition, at first the predicate ismaximal is defined. Intuitively, an expression is maximal if none of the rewrite rules can be immediately applied, i.e. it is maximal, iff one or more free identifier occur in all decisive positions. The maximality is specified by the following axioms, i.e. ismaximal is the least (weakest) predicate fullfilling the following axioms:

ismaximal (x) for every identifier x.

$(\forall i,\ 1 \leq i \leq n:\ (\text{ismaximal}(E_i) \vee E_i \in \text{DOM}) \wedge$
$\exists i,\ 1 \leq i \leq n:\ \text{ismaximal}(E_i)) \Rightarrow \text{ismaximal}(g(E_1, \ldots, E_n))$

ismaximal (C) ⇒ ismaximal (if C then E1 else E2 fi)

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

Note, that one might be less restrictive in defining the maximality of the conditional.

The computation rule call-in-parallel (parallel evaluation) is specified by the following rewrite rules:

(1) Evaluation of conditional expressions

if tt then E1 else E2 fi → E1,
if ff then E1 else E2 fi → E2,
C → C' ⇒ if C then E1 else E2 fi → if C' then E1 else E2 fi,

(2) Evaluation of choice

(E1 [] E2) → E1,
(E1 [] E2) → E2,

(3) Unfold of recursively defined functions

$f_i(E_1, \ldots, E_n) \rightarrow F_i(E_1, \ldots, E_n)$,

(4) Evaluation of function applications

$$\forall i,\ 1 \le i \le n : (E_i \to E_i' \vee ((\text{ismaximal}(E_i) \vee E_i \in \text{DOM}) \wedge E_i = E_i'))) \wedge$$
$$(\exists i,\ 1 \le i \le n : E_i \to E_i') \to g(E_1,\ldots,E_n) \to g(E_1',\ldots,E_n'),$$
$$(\forall i,\ 1 \le i \le n+1 : (E_i \to E_i' \vee (\text{ismaximal}(E_i) \wedge E_i = E_i'))) \wedge$$
$$(\exists i,\ 1 \le i \le n+1 : E_i \to E_i') \to$$
$$(\lambda x_1,\ldots,x_n : E_{n+1})(E_1,\ldots,E_n) \to (\lambda x_1,\ldots,x_n : E_{n+1}')(E_1',\ldots,E_n')$$

(5) Communication of computed arguments

$$E_i \in \text{DOM} \to (\lambda x_1,\ldots,x_n : E_{n+1})\ (E_1,\ldots,E_n) \to$$
$$(\lambda x_1,\ldots,x_{i-1},\ x_{i+1},\ldots,x_n : E_{n+1}\ [E_i/x_i])$$
$$(E_1,\ldots,E_{i-1},\ E_{i+1},\ldots,E_n).$$

(6) Termination of the evaluation of function applications

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

(7) Simplification of primitive expressions

E primitive, $E \in \text{DOM}$, $B[E] = \{e\} \to E \to e$.

Note that this rule can even be applied to expressions containing free identifiers, as far as these identifiers do not become decisive, i.e. ismaximal (E) does not hold (cf. partial evaluation, mixed computation in /Ershov 78/).

A computation rule R is called

- consistent, if for every pair of programs

$P = \lceil \underline{\text{funct}}\ f_1 \equiv F_1,\ldots,\ \underline{\text{funct}}\ f_n \equiv F_n,\ E \rfloor$,

$P' = \lceil \underline{\text{funct}}\ f_1 \equiv F_1,\ldots,\ \underline{\text{funct}}\ f_n \equiv F_n,\ E' \rfloor$

we have

$$E\ R\ E' \to B[P'] \subseteq B[P],$$

i.e. if P' is descendant (cf. /McCarthy 63/, or implementation cf. /Broy et al. 78/, /Broy et al. 80/) of P.

- complete, if for every program P with $e \in SET(B[P])$ there exists a finite sequence of expressions $\{E_i\}_{0 \leq i \leq n}$ such that

$$E = E_0 \; R \; E_1 \; \ldots \; R \; E_n = e$$

holds, provided e is total and finite.

- finitary, iff for every program P there exists only a finite number of expressions E' such that E R E' ,

- effective, iff for every program P there exists an infinite sequence $\{E_i\}_{i \in IN}$ of expressions with $E \; R \; E_0$ and $E_i \; R \; E_{i+1}$, at most if $e \in B[P]$ with e infinite or partial.

Due to König's Lemma, a finitary, effective rule can only compute programs P with (provided DOM is flat, cf. /König 50/):

$$| \; B[P] \; | < \infty \quad \text{or} \quad \perp \in B[P]$$

This notion is directly related to the notion of finite approximability (cf. section 2) of multidomains over flat domains.

Lemma: The rule "$\rightarrow$" is

(1) consistent
(2) complete
(3) finitary
(4) effective

Sketch of proof:

Let f_j be defined by $f_j^0 = \lambda \, x_1, \ldots, x_{n_j} : \perp$, $f_j^{i+1} = F_j[f_1^i / f_1, \ldots, f_n^i / f_n]$.

(1) According to the definition of our language all rules are consistent.

(2) According to the definition of fixed points, there exists $i \in \mathbb{N}$ such that for $e \in B[E[f_1^i/f_1,\ldots,f_n^i/f_n]]$ appropriate application of the rules leads from

$$E[f_1^i/f_1,\ldots, f_n^i/f_n] \text{ to } e .$$

(3) Proof by structural induction on E .

(4) Assume all $e \in B[P]$ are finite and total, then there exists an $i \in \mathbb{N}$ such that

$$B[P] = B[E[f_1^i/f_1,\ldots f_n^i/f_n]].$$

Every computation step reduces

- the number of occurring "λ"-symbols or
- the number of <u>if-fi</u> constructs or
- the number of formal parameters or
- the number of primitive function symbols or
- the number of λ-symbols

and only a finite number of them may occur.

Since E itself cannot become maximal (since all $e \in B[E]$ are total), after a finite number of computation steps the computation must terminate with $e \in B[E]$, at least if all $e \in B[E]$ are finite and total. □

<u>Example:</u>

Let us consider the recursive program:

$$[\underline{\text{funct}}\ f \equiv F,\ f(1,1)]$$

with the abbreviation:

$$F \triangleq \lambda\, x, y : R[f, x, y]$$

$$R[f,x,y] \triangleq \underline{\text{if}}\ \ x=0\ \ \underline{\text{then}}\ 2 * y\ \underline{\text{else}}\ \ f(x-1,\ f(x-1,\ y+1))\underline{\text{fi}}$$

For the function application f(1,1) we obtain the computation sequence:

f(1,1) →
F(1,1) →
(λy : R[f, 1, y]) (1) →
R[f, 1, 1] →
if ff **then** 2 * 1 **else** f(1-1, f(1-1, 1+1)) **fi** →
f(1-1, f(1-1, 1+1)) → F(1-1, f(1-1, 1+1)) →
F(0, F(1-1, 1+1)) →
(λ y : R[f, 0, y]) (F(1-1, 1+1)) →
(λ y : **if** tt **then** 2 * y **else** f(0-1, f(0-1, y+1)) **fi**) (F(0,2)) →
(λ y : 2 * y) (**if** tt **then** 2*2 **else** f(0-1, f(0-1, 2+1)) **fi**) → ...
(λ y : 2 * y) (4) →
2 × 4 →
8

end of example

The main difference between the computation rule "→" and the rules given in /Manna et al. 73/ is found in the different substitution mechanism. In /Manna et al. 73/ 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 /Manna et al. 73/ only deterministic expressions are considered), while in the rule "→" this indivisible action is divided in up to n+1 independent substitution steps.

Note, that guarded expressions (as a counterpart to guarded commands in /Dijkstra 76/) can be expressed as a notational extension by the rules (cf. also /Bauer, Wössner 81/, page 69):

$$\underline{\text{if}}\ C\ \underline{\text{then}}\ E\ \underline{\text{fi}} = \underline{\text{if}}\ C\ \underline{\text{then}}\ E\ \underline{\text{else}}\ \bot\ \underline{\text{fi}},$$

$$\underline{\text{if}}\ C_1\ \underline{\text{then}}\ E_1\ [\!]\ \ldots\ [\!]\ C_n\ \underline{\text{then}}\ E_n\ \underline{\text{fi}} = (G_1\ [\!]\ \ldots\ [\!]\ G_n),$$

where

$$G_i = \underline{\text{if}}\ C_i\ \underline{\text{then}}\ E_i\ \underline{\text{else if}}\ C_1\ \underline{\text{then}}\ E_1\ [\!]\ldots[\!]\ C_{i-1}\ \underline{\text{then}}\ E_{i-1}\ [\!]\ C_{i+1}\ \underline{\text{then}}\ E_{i+1}\ [\!]\ldots[\!]\ C_n\ \underline{\text{then}}\ E_n\ \underline{\text{fi}}\ \underline{\text{fi}}$$

Thus guarded expressions can be seen as a notational combination of choice and conditional expression.

Remark: 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 $(S^n)^\bot$ versus "cartesian product" $(S^\bot)^n$, cf. /Bauer, Wössner 81/).

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

(P1) $\lceil \underline{\text{funct}}\ f_1 = \lambda\ x : x+x,\quad f_1(o\ [\!]\ 1) \rfloor$

(P2) $\lceil \underline{\text{funct}}\ f_2 = \lambda\ x : 2*x,\quad f_2(o\ [\!]\ 1) \rfloor$

In strict call-by-value we obtain $B[P1] = B[P2] = \{0,2\}$, while (as pointed out in /Hennessy, Ashcroft 76/) in straightforward call-by-name semantics we obtain $B[P1] = \{0, 1, 1, 2\}$ and $B[P2] = \{0, 2\}$, 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 $MD(S^\bot)^n \to MD(S^\bot)$

whereas in call-by-value it suffices to take $(S^n)^\perp \to \mathcal{MP}(S^\perp)$ (cf. /Astesiano, Costa 79/, /Benson 79/, /Hennessy 80/).

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 /Hennessy, Ashcroft 77/. It can be evaluated by an extension of delayed evaluation (cf. /Vuillemin 74/) or call-by-need (cf. /Wadsworth 71/) to nondeterministic functions. It allows one to consider nondeterministic functions as elements from $(S^\perp)^n \to \mathcal{MP}(S^\perp)$. 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 thus are *needed*, but starts the evaluation of the body of the function and of the arguments in parallel, simply eliminating computations of arguments which apparently are no longer needed. So one might, in analogy, talk of enforced evaluation.

With the computation rule "→" 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 identifer 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, than 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 /Broy 80a/.

end of remark

The evaluation rule described in this section can be seen as operational semantics for a kind of a data flow language where, however, in contrast to the straightforward concept of demand driven evaluation (cf. /Dennis 74/, /Kosinski 73, 77/) much more is done in parallel. Of course, the programs of our languages can also be represented as graphs (cf. function graphs in /Keller 80/), such that the evaluation rule describes a graph reduction process.

One may also think of a reduction machine for an efficient implementation of the evaluation rule. Note, that the classical straightforward transformation of applicative programs (see /Bauer, Wössner 81/) generally means a transition to strict innermost evaluation and destroys the possibilities of inherent parallelism (cf. /Broy 80a/) when going to procedural versions.

5. Applicative Communicating Systems

Parallel evaluations of sub-expressions (such as parameters of a function) which do not interfere do not cause any problems. They can be specified, considered, computed and analysed separately. If, however, there is some possibility of communication, i.e. the possibility of transmitting (intermediate) results from one expression to the other, and if the "behavior" of the expressions is influenced by these communi-

cated results, then it is more difficult to consider the meaning of such expressions.

5.1 Specific Rewrite Rules for Communications

The phenomenon of communication can even be observed in the computation rules of the preceding section. A function application

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

can be considered as a system of $n+1$ expressions, the evaluations of which can be viewed as *communicating processes*. If one of the processes evaluating E_i has successfully terminated with result e_i, then this result is communicated to the process evaluating E_{n+1} and we obtain

$$(\lambda x_1,\ldots,x_{i-1}, x_{i+1}, \ldots,x_n : E_{n+1}[e_i/x_i])\ (E_1,\ldots,E_{i-1}, E_{i+1},\ldots,E_n)$$

Obviously this is only a very restricted type of communication, which only occurs if the sending process has already terminated and the communication is its last action.

To obtain more general types of communication, it seems adequate to consider a particular domain

$$\text{DOM} = (\text{ATOM} \cup \text{SEQ})^{\perp}$$

where $\text{SEQ} = \text{ATOM}^*$. We use the functions isempty, first, rest as defined in section 2 and the function

$$\text{app} : \text{ATOM}^{\perp} \times \text{SEQ}^{\perp} \to \text{SEQ}^{\perp}$$

defined by

$$\text{app}(e, s) = \begin{cases} \text{ap}(e,s) & \text{if } e \neq \perp,\ s \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

where ap is defined as in section 2.

With this definition, DOM is a flat domain, provided $ATOM^{\perp}$ is flat. It is a proper subdomain of the domain of streams. In contrast to the usual primitive functions, we define an application app(e, s) as maximal iff ismaximal(e) holds.

Now specific computation rules for evaluating stream processing functions can be given:

$$\forall i,\ 1 \leq i \leq n+1 : (E_i \to E_i' \vee (ismaximal(E_i) \wedge E_i = E_i')) \wedge e \in ATOM \to$$
$$(\lambda\, x_1,\ldots,x_n : E_{n+1})(E_1,\ldots,E_{i-1},app(e,E_i),E_{i+1},\ldots,E_n) \to$$
$$(\lambda\, x_1,\ldots,x_n : E'_{n+1}[app(e,x_i)/x_i])(E_1',\ldots,E_n')$$

and for $e \in ATOM$

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

Lemma: If the computation rules of the preceding sections are complemented by the rules above, the resulting rule is also consistent, complete, finitary and effective.

Proof: The consistency of the rules immediately follows from the language definition and the definition of the function app. Finitarity and completeness is trivial. The effectiveness follows from the fact that all elements $s \in SEQ$ are of finite length.

□

Note that the rule above allows to shorten the computations because if E_{n+1} is maximal because x_i is decisive, a substitution of x_i by app(e, x_i) should allow E_{n+1} to proceed.

Example: The producer/ consumer problem

```
⌈funct produce  ≡ λ x : if x=0 then empty
                                else app(product(x),produce(x-1)) fi

 funct consume  ≡ λ s : if isempty(s) then t
                                      else g(first(s),consume(rest(s)
                        fi

 consume(produce(n))
```

for some $n \in \mathbb{N}$, arbitrary functions product and g, $t \in$ ATOM.

In this example we obtain, assuming product(i) = $p_i \in$ ATOM

```
consume(produce(10))  →
(λ s : ...) ((λ x : ...) (10)) → ...
(λ s : ...) (app(product(10), produce(9))) → ...
(λ s : ...) (app(p10, ap(p9, (λ x : ...) (8)))) →
(λ s : if isempty(app(p10, s)) then ... fi) (ap(p9,...)) → ...
```

However we cannot conclude in this step

$$\text{isempty}(\text{ap}(p_{10}, s)) \rightarrow \text{ff}$$

since s may be $\perp$. Therefore in our example the consumer process has to wait until the producer process has successfully terminated. So the benefits of the two rules of communication cannot be fully exploited.

end of example

So one really has to consider the non flat domain of streams instead of the flat domain used so far.

5.2 Streams

To cope with this problem we would like to have rules like

$$\begin{aligned} \text{isempty}(\text{ap}(e, s)) &= \text{ff}, \\ \text{first}\ (\text{ap}(e, s)) &= e\ , \\ \text{rest}\ (\text{ap}(e, s)) &= s\ , \end{aligned}$$

for all $s \in \text{STREAM}$, $e \in \text{ATOM}$ i.e. we need a *nonstrict* constructor function like the one used in *lazy evaluation* (cf. /Henderson , Morris 76/, /Friedmann, Wise 76/. So we define now for $\text{DOM} = \text{ATOM} \cup \text{STREAM}$ where $\text{STREAM} = \text{SEQ} \cup (\text{ATOM}^* \times \{\bot\}) \cup \text{ATOM}^\infty$

$$\begin{aligned} .\&. &: \text{ATOM}^\bot \times \text{STREAM} \to \text{STREAM} \\ \underline{\text{first}}. &: \text{STREAM} \to \text{ATOM}^\bot \\ \underline{\text{rest}}. &: \text{STREAM} \to \text{STREAM} \\ \underline{\text{isempty}}. &: \text{STREAM} \to \{\text{tt}, \text{ff}\}^\bot \\ \text{empty} &: \to \text{STREAM} \end{aligned}$$

$$B[E \ \& \ S] = \underset{e \in B[E]}{\overset{(+)}{\cup}} \ \underset{s \in B[S]}{\overset{(+)}{\cup}} \{\text{ap}(e, s)\}$$

$$B[\underline{\text{first}}\ S] = \underset{s \in B[S]}{\overset{(+)}{\cup}} \{\text{first}(s)\}$$

$$B[\underline{\text{rest}}\ S] = \underset{s \in B[S]}{\overset{(+)}{\cup}} \{\text{rest}(s)\}$$

$$B[\underline{\text{isempty}}\ S] = \underset{s \in B[S]}{\overset{(+)}{\cup}} \{\text{isempty}(s)\}$$

$B[\text{empty}] = \{ \varepsilon \}$
where ap, first, rest, isempty are defined as in section 2.

From the results of section 2 the continuity of the contructs &, first, rest and isempty immediataly follows. Now our computation rules can be complemented by the rules:

$\underline{\text{first}}\ (E\ \&\ S) \rightarrow E,$

$\underline{\text{rest}}\ (e\ \&\ S) \rightarrow S \qquad \text{for} \quad e \in DOM, \quad e \neq \perp$

$\underline{\text{isempty}}\ (e\ \&\ S) \rightarrow ff \qquad \text{for} \quad e \in DOM, \quad e \neq \perp$

$E \rightarrow E' \Rightarrow E\ \&\ S \rightarrow E'\ \&\ S,$

$E \rightarrow E' \Rightarrow \underline{\text{first}}\ E \rightarrow \underline{\text{first}}\ E',$

$E \rightarrow E' \Rightarrow \underline{\text{rest}}\ E \rightarrow \underline{\text{rest}}\ E',$

$E \rightarrow E' \Rightarrow \underline{\text{isempty}}\ E \rightarrow \underline{\text{isempty}}\ E',$

$\underline{\text{isempty}}\ \text{empty} \rightarrow tt,$

$\text{ismaximal}\ (E\ \&\ S) = \text{ismaximal}\ (E),$

$\text{ismaximal}\ (\underline{\text{first}}\ S) = \text{ismaximal}\ (\underline{\text{rest}}\ S) = \text{ismaximal}\ (\underline{\text{isempty}}\ S) = \text{ismaximal}\ (S).$

Using these rules one obtains really systems of communicating expressions, however, the communication paths are still noncyclic. For expressing real "feed back" we introduce systems of mutually recursively defined streams.

5.3 Systems of Expressions Communicating by Streams

For allowing cyclic communication paths, systems of expressions which mutually communicate via streams are considered using the following syntax:

< c-program > ≡ ⌈{funct <function identifier>≡<funct abstract>,}*
{stream<stream identifier>≡<expr>,}* <expr>⌋

Throughout this section we consider the scheme R of the from:

$$\lceil \underline{\text{funct}}\ f_1 \equiv F_1, \ldots, \underline{\text{funct}}\ f_n \equiv F_n,$$
$$\underline{\text{stream}}\ s_1 \equiv S_1, \ldots, \underline{\text{stream}}\ s_m \equiv S_m,\ E \rfloor$$

where s_i does not occur in F_j.
To define a semantics for R again fixed point theory is applied.

A system of expressions communicating by streams corresponds to a mutually recursive definition of streams. This coincides with the result of /Broy 80b/, 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 stream s = E and the recursive definition of a nullary function funct s = λ : E (cf. the proposal of /Kahn 74/, /Kahn, MacQueen 77/).

Note, however, that for instance the system S1

⌈funct f = λ : (1 ▯ 2) & empty, first f()=first f()⌋

is different from the system S2

⌈stream s = (1 ▯ 2) & empty, first s = first s⌋

since functions are treated by simple substitution ("call-by-name"-like with respect to the body of the function), while streams are to be substituted (elementwise) only after evaluation ("call-by-value" or more precisely "call time choice"), such that SET(B[S1]) = {tt, ff} while SET(B[S2]) = {tt}.

So to avoid mixing alternative possibilities of behavior (cf. the "merge anomaly" described in /Keller 78/, /Brock, Ackermann 81/) we have to consider the stream equations as multisets of fixed point equations rather than one fixed point equation for a multi-set-valued ("nondeterministic") function. To explain these problems let us consider the following example:

Example: Consider the two programs

P1 : ⌈funct f ≡ λ ; (0 & f()) ‖ ((1 + first f())&f()), first f()⌋

P2 : ⌈stream s ≡ (0 & s) ‖ ((1 + first s) & s) , first s ⌋

Our definition gives $SET(B[P1]) = \mathbb{N} \cup \{\bot\}$, however for the stream we would like to obtain $SET(B[P2]) = \{0\} \cup \{\bot\}$

end of example

Intuitively speaking, a stream equation defines a process with one determinate identity for each of its applications in one particular instance of evaluation, while a function equation defines a nondeterminate function where for every application a new individual choice can be taken.

Therefore we introduce a new semantic function BF , associating with every expression E in which at most the free stream identifiers $s = (s_1,\ldots,s_m)$ occur a functional which gives for every m-tuple of streams a multiset of deterministic partial functions, which can be taken as functions for defining streams as their least fixed points.

$$BF : EXP[s] \to (STREAM^m \to \mathcal{M}(STREAM^m \to STREAM))$$

Now we define:

$$BF\,[\text{if } C \text{ then } E1 \text{ else } E2 \text{ fi}]\,(\tilde{s}) =$$

$$\overset{(+)}{p \in BF[C](\tilde{s})} \begin{cases} \overset{(+)}{f \in BF[E1](\tilde{s})} \{cond(p,f)\} & \text{if } p(\tilde{s}) = tt \\ \overset{(+)}{f \in BF[E2](\tilde{s})} \{cond(\neg p,f)\} & \text{if } p(\tilde{s}) = ff \\ \{\Omega\} & \text{otherwise} \end{cases}$$

$BF[s_i]\ (\tilde{s}) = \{\lambda\ s_1,\ldots,s_n : s_i\}$

$BF[E1 \parallel E2]\ (\tilde{s}) = B[E1]\ (\tilde{s}) \overset{(+)}{\cup} B[E2]\ (\tilde{s})$

$BF[e]\ (\tilde{s}) = \{\lambda\ s_1,\ldots,s_m : e\}$ for $e \in DOM$

$$BF[(\lambda\ x_1,\ldots,x_n : E_{n+1})\ (E_1,\ldots,E_n)]\ (\tilde{s}) =$$

$$\overset{(+)}{f_1 \in BF[E_1](\tilde{s})} \cdots \overset{(+)}{f_n \in BF[E_n](\tilde{s})} BF[E_{n+1}\ [f_1(s)/x_1\ \ldots\ f_n(s)/x_n]](\tilde{s})$$

$$BF[g(E_1,\ldots,E_n)]\ (\tilde{s}) =$$

$$\overset{(+)}{f_1 \in BF[E_1](\tilde{s})} \cdots \overset{(+)}{f_n \in BF[E_n](\tilde{s})} \{\lambda\ s : g(f_1(s),\ldots,f_n(s))\}$$

where Ω denotes the totally undefined function and cond:
$(STREAM^m \to \{tt, ff, \perp\}) \times (STREAM^m \to DOM) \to (STREAM^m \to DOM)$
is defined by cond $(p, f)(\tilde{s})$ = if $p(\tilde{s})$ then $f(\tilde{s})$ else $\perp$.
So given the system R, we associate with f_i the least fixedpoint
$BF[f_i] = BF[E_i]$ where $F_i = \lambda x_1,\ldots,x_{n_i} : E_i$.

So we may define

$$BF[f_i(E_1,\ldots,E_{n_i})]\ (\tilde{s}) = \overset{(+)}{h_1 \in BF[E_1](\tilde{s})} \cdots \overset{(+)}{h_{n_i} \in BF[E_{n_i}](\tilde{s})}$$

$$\overset{(+)}{f \in BF[f_i](h_i(\tilde{s}_1),\ldots,h_{n_i}(\tilde{s}))} \{\lambda s : f(h_1(s),\ldots,h_{n_i}(s))\}.$$

We associate with $s = s_1,\ldots s_n$ the multiset S:

$$S = \overset{(+)}{s \in STREAM^m} \begin{cases} \overset{(+)}{h_1 \in BF[S_1](s)} \\ \vdots \\ \overset{(+)}{h_m \in BF[S_m](s)} \end{cases} \begin{cases} \{s\} \text{ if } s = Y\tilde{s} : (h_1(\tilde{s}),\ldots,h_m(\tilde{s})) \\ \emptyset \text{ otherwise} \end{cases}$$

where Y denotes the least-fixed-point operator for streams.

In particular systems of expressions communicating by streams can be used to represent networks of processes as considered for instance in /Arnold 79/.

Again a number of computation rules can be given to evaluate such systems of communicating expressions.

Now we define 4 rewrite rules. We use the abbreviations:

$$R1 = (\underline{\text{stream}}\ s_1 = S_1, \ldots, \underline{\text{stream}}\ s_m = S_m, S_{m+1})$$

$$R2 = (\underline{\text{stream}}\ s_1 = S'_1, \ldots, \underline{\text{stream}}\ s_m = S'_m, S'_{m+1})$$

(1) Parallel computation

$$\forall i,\ 1 \leq i \leq m+1 : (S_i \rightarrow S'_i \vee (\text{ismaximal}(S_i) \wedge S_i = S'_i)) \Rightarrow R1 \rightarrow R2$$

(2) Communication

$$(S_j = e \,\&\, \tilde{S}_j \wedge \forall i,\ 1 \leq i \leq m+1 : (i \neq j \Rightarrow S'_i = S_i[e \,\&\, s_j/s_j]) \wedge S'_j = \tilde{S}_j[e \,\&\, s_j/s_j]) \Rightarrow R1 \rightarrow R2$$

(3) Termination of single processes

$$(S_j = \text{empty} \wedge \forall i,\ 1 \leq i \leq m+1 : (i \neq j \Rightarrow S'_i = S_i[\text{empty}/s_j]) \Rightarrow R1 \rightarrow (\underline{\text{stream}}\ s_1 = S'_1, \ldots, \underline{\text{stream}}\ s_{j-1} = S'_{j-1}, \underline{\text{stream}}\ s_{j+1} = S'_{j+1}, \ldots, \underline{\text{stream}}\ s_m = S'_m, S'_{m+1})$$

(4) Termination of the system

$$S_{m+1} = e \Rightarrow R1 \rightarrow e \qquad \text{where } e \in \text{DOM}$$

The first 2 of these rules can be combined into one rule which guarantees the simultaneous progress of the evaluations of the single processes (note that this is just one possibility for a computation rule assuring the continuous computation and communication progress for all processes).

$\forall i,\ 1 \leq i \leq m + 1$:

$$(\!(S_i \rightarrow \tilde{S}_i \vee (\text{ismaximal}(S_i) \wedge S_i = \tilde{S}_i)) \wedge C_i = s_i) \vee$$
$$(S_i = e_i \ \&\ \tilde{S}_i \wedge C_i = e_i \ \&\ s_i \wedge i \neq m + 1)) \wedge$$
$$S'_i = \tilde{S}_i \left[C_1/s_1, \ldots, C_m/s_m\right] \Rightarrow R1 \rightarrow R2$$

For communicating intermediate results to the outside world one might add rules like (cf. rule (4)):

$$S_{m+1} = e \ \&\ S'_{m+1} \Rightarrow R1 \rightarrow e \ \&\ R2$$

Again the computation rule resulting from adding this rewrite rule to the previous rules is effective, finitary, consistent and complete.

Similar rules may be considered as an extension of the technique of call-by-need (cf. /Wadsworth 72/) or delayed evaluation (cf. /Vuillemin 74/) to primitive functions which previously have always been assumed to be strict (cf. /Manna et al. 73/) leading to the concept of lazy evaluation (cf. /Friedman, Wise 76/ , /Henderson, Morris 76/, /Bauer 79/). This allows a treatment of infinite objects such as infinite streams by particular computation rules. The rules given above, however, then could be called "speedy evaluation", or also "busy evaluation" or, extending the notion of section 4 to primitive functions, parallel evaluation, since the evaluation of an infinite stream is not delayed until a selector function is applied as in lazy evaluation, but the evaluation is continuously enforced such that an infinite stream is consecutively generated.

Example:

Let the function f be defined by :

funct f ≡ λ t, n : if n = 0 then empty
else (first x+1) & f(rest x, n-1) fi ,

and the interactive system E by :

E ≙ ⌈stream t ≡ 0 & f(t, 2), t ⌋ .

We obtain a computation sequence for E by our computation rule. UNFOLDING f and simplifying the result we get:

⌈stream t ≡ 1 & f(t, 1), 0 & t⌋

Applying the rule again we get :

⌈stream t ≡ f(1 & t, 1), 0 & 1 & t⌋

and: ⌈stream t ≡ 2 & f(t, 0), 0 & 1 & t⌋

and: ⌈stream t ≡ f(2 & t, 0), 0 & 1 & 2 & t⌋

and: ⌈stream t ≡ empty, 0 & 1 & 2 & t⌋

and finally : 0 & 1 & 2 & empty

Since no nondeterminism is involved, we can use simple iteration to compute the fixed point of the stream function: The iteration gives

$K^{(0)} = \{\bot\}$
$K^{(1)} = \{0 \,\&\, \bot\}$
$K^{(2)} = \{0 \,\&\, 1 \,\&\, \bot\}$
$K^{(3)} = \{0 \,\&\, 1 \,\&\, 2 \,\&\, \bot\}$
$K^{(4)} = \{0 \,\&\, 1 \,\&\, 2 \,\&\, \text{empty}\}$
$K^{\infty} = K^{(4+n)} = K^{(4)}$

end of example

Example: Applicative Loops

"Applicative loops"(the name is due to Keller) can be seen as specific "iterative" programs producing an (eventually infinite) stream. They can conveniently be used to compute sequences of results of a function given in the form of course-of-value recursion. Consider for instance

```
funct f = λ n : if n=0 then E0
           else if n=1 then E1
                       else  g(f(n-2), f(n-1), n)  fi
```

then with

```
stream s =  E0 & (E1 & h(s, 2))
```

where

```
funct h = λ s, n : g(first s, first rest s, n) & h(rest s, n+1)
```

One simply proves, that

$$\underline{\text{first}}\ \underline{\text{rest}}^{\,i}\ s = f(i)$$

This is a simple example for a formally justified transformation from classical applicative programs to stream processing programs. Note that the program above consists of an "inner most" recursive definition of the stream s (in contrast to "outermost"-recursion or tail-recursion for recursive functions representing loops).

end of example

Example

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./Dijkstra 76/), in ascending order one may use three communicating streams:

```
⌈funct streammult ≡ λ n, s : (n × first s) & streammult(n,rest s),
 funct merge ≡ λ s1, s2 : if first s1 ≤ first s2
                          then first s1 & merge(rest s1, s2)
                          else first s2 & merge(s1, rest s2) fi,
 stream s1 ≡ streammult(5, 1 & s1),
 stream s2 ≡ merge(streammult(3, 1 & s2), s1),
 stream s3 ≡ merge(streammult(2, 1 & s3), s2),        s3          ⌋
```

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

end of example

6. Concurrency

One of the most intricate issues in multiprogramming is that of concurrency. Analogously to everyday life one may talk of two (or more) concurrent candidates (processes, expressions, programs), if these two candidates both compete for something (for instance to be served or to be elected). For resolving a competition a choice has to be made.

6.1 Ambiguity Operator and Nonstrict Merging

One may consider an expression (E1 ‖ E2) as a competition of the expressions E1 and E2 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 E1 or E2. Therefore in the expression

⌈stream s1 = S1, stream s2 = S2, if C(s1,s2)
then first s1 else first s2 fi⌋

there is no way to formulate the predicate C such that the first alternative is chosen if first s1 ≠ ⊥ and the second one is chosen if first s2 ≠ ⊥ (and ambiguously one of them is chosen if both are ≠ ⊥). If such a predicate would be definable, then functions g would be definable, such as the parallel or

$$g(tt, \bot) = tt, \quad g(\bot, tt) = tt, \quad g(ff, ff) = ff$$

According to /Hennessy, Ashcroft 80/ such a function is not definable in a nondeterministic language like the one defined in section 3. Note that all definable functions are either constant or strict in at least one argument. But even the "parallel" or would not solve the above problem.
However, one may use a more strongly defined choice operator, such as McCarthy's ambiguity operator, the meaning of which is specified in /McCarthy 63/ as follows:

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

Formally the definition may be written:

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

and C(s1, s2) in the program above may be expressed by
amb(¬ isempty s1 , isempty s2).
This is a nonstrict extension (and so not a natural one) of the choice operator.

However, the ambiguity operator causes several problems:

- it allows writing noncontinuous functions such as

  funct f = λ x : amb(x, f(x+1))

  where f(0) represents a multiset which is not finitely approximable, i.e. it is not a member of the multidomain (cf. also the discussion on *fairness* in /Park 80/, /Broy 81/, /Apt, Plotkin 81/, /Broy, Wirsing 81b/)

- it is not even monotonic, neither in the Egli-Milner ordering nor the multiset-ordering, since $\{1\} \sqsubseteq \{1\}$, $\{\perp\} \sqsubseteq \{2\}$, but $amb(1, \perp) = \{1\} \not\sqsubseteq \{1,2\} = amb(1,2)$.
  This is even worse since in the case of noncontinuity but monotonicity (like in the fairness-discussion) one can still work with least fixed points (cf. /Apt, Plotkin 81/).

Moreover one cannot hope to obtain finitary, consistent, complete, and effective computation rules which compute f(o) , since, due to König's Lemma, a finitary tree (i.e. a tree with a finite number of branches at each node) may only contain an infinite number of nodes if there exists an infinite path (i.e. the tree is infinite) and therefore a nonterminating computation for f(o) can not be excluded by a complete, finitary rule.

So one has to choose between dropping completeness or finitarity. Since infinitary rules do not seem very realistic, because no real machine can be assumed which makes an infinite number of choices within finite time, we drop the requirement of completeness and rather consider an infinite set of finitary, effective and consistent computation rules, such that for each feasible value x there exists at least one rule which computes x .

This very clearly reflects the needs of software for parallel processsing. For every program P any implementation or system S does not guarantee that every feasible value x of P can be the result of running P on S, but every result x computed by S when applied to P is a feasible value. Thus one might talk of "loose" nondeterminism here (as an interpretation of this notion as found in /Park 80/).

In accordance with McCarthy's ambiguity operator a choice operator ∇ is used in infix notation specified by

$$B[E1 \nabla E2] = \bigcup^{(+)}_{e1 \in B[E1]} \bigcup^{(+)}_{e2 \in B[E2]} B[amb(e1, e2)]$$

Note: The use of a special element none as advocated in /Henderson 80/ corresponds to backtracking on finite errors and reflects a much simpler concept, causing no problems with continuity.

end of note

As a consequence, "∇" is not continuous and not even monotonic.

Therefore we use another approach, exploiting the following two facts:

Lemma: All language constructs are monotonic with respect to the descendent relation, i.e. multiset-inclusion:

$B[E] \subseteq B[E']$ implies $B[CN[E]] \subseteq B[CN[E']]$ for all contexts CN .

Proof: Structural induction (cf. /Broy et al. 78/) □

This property is in particular of interest for program development by refinement. Now we define a second choice operator Δ :

$$B[E1 \,\Delta\, E2] = \bigcup^{(+)}_{e1 \in B[E1]} \bigcup^{(+)}_{e2 \in B[E2]} \{e1, e2\}$$

Trivially Δ is monotonic and continuous.

Lemma: Let E be an expression and $\tilde{E}$ be the expressiong resulting from replacing all occurrences of ∇ in E by Δ then

$$B[E] \subseteq B[\tilde{E}]$$

Proof: $B[E1 \nabla E2] \subseteq [E1 \Delta E2]$ for all $E1$, $E2$. □

Given a program P:

$$[\underline{\text{funct}}\ f_1 = F_1, \ldots, \underline{\text{funct}}\ f_n = F_n\ ,$$
$$\underline{\text{stream}}\ s_1 = S_1, \ldots, \underline{\text{stream}}\ s_m = S_m,\ E]$$

and the program $\tilde{P}$ resulting from substituting all occurrences of "∇" in P by "Δ". The semantics of $\tilde{P}$ is well-defined. So functions $\tilde{f}_i$ can be associated with the recursive definitions in $\tilde{P}$ by taking least fixed points

$$\tilde{f}_i : DOM^{n_i} \to MD\ (DOM)$$

of the equations $\tilde{f}_i\ (x_1, \ldots, x_{n_i}) = B[(\tilde{F}_i[\tilde{f}_1/f_1, \ldots, \tilde{f}_n/f_n])(x_1, \ldots, x_{n_i})]$. Now we associate functions f'_i with the functions in P by taking the $\supseteq$-least (i.e. $\subseteq$-greatest) fixed points

$$f'_i : DOM^{n_i} \to M\ (DOM)$$

of the equations

$$f'_i\ (x_1, \ldots x_{n_i}) = B[\,(F_i[f'_1/f_1, \ldots,\ f'_n/f_n])(x_1, \ldots x_n)]$$

where $f'_i\ (x_1, \ldots, x_{n_i}) \subseteq \tilde{f}\ (x_1, \ldots,\ x_{n_i})$.

Due to the $\subseteq$ - monotonicity and Tarski's fixed point theorem such fixed points exist and are uniquely defined.
Note, that the definition of the constructs of our language (apart from fixed point definitions) also works for arbitrary multisets not contained in $\mathcal{MD}(DOM)$.

Similarly we define

$$BF[E1 \nabla E2]\ (\tilde{s}) = \overset{\oplus}{h1 \in BF[E1](\tilde{s})} \overset{\oplus}{h2 \in BF[E2](\tilde{s})} \begin{cases} \{h1,h2\} & \text{if } h1\ (\tilde{s}) \neq \perp,\ h2\ (\tilde{s}) \neq \perp \\ \{h1\} & \text{if } h1\ (\tilde{s}) \neq \perp,\ h2\ (\tilde{s}) = \perp \\ \{h2\} & \text{if } h1\ (\tilde{s}) = \perp,\ h2\ (\tilde{s}) \neq \perp \\ \{\Omega\} & \text{if } h1\ (\tilde{s}) = \perp,\ h2\ (\tilde{s}) = \perp \end{cases}$$

Now in systems of recursively defined streams we may associate with the f_i a multiset of functions $BF[f_i]$ as the $\subseteq$ - maximal fixed point of

$$BF[f_i] = BF[E_i] \qquad \text{where } F_i = \lambda x_1,\ldots,x_{n_i} : E_i \text{ and}$$

$$BF[f_i] \subseteq BF[\tilde{F}_i]$$

and the $BF[\tilde{f}_i]$ are the fixed points associated with $\tilde{F}$.
So we may define

$$S = \overset{\oplus}{s \in STREAM^m} \overset{\oplus}{h_1 \in BF[S_1]}\ (s)$$

$$\overset{\oplus}{h_m \in BF[S_m](s)} \vdots \begin{cases} \{s\} & \text{iff } s = \tilde{Y}s : h_1(\tilde{s}),\ldots,h_m(\tilde{s}) \\ \emptyset & \text{otherwise} \end{cases}$$

Note, that the functions in $BF[f_i](\tilde{s})$ are still monotonic and continuous in spite of using the ambiguity operator.

Finally

$$B[P] = \overset{\oplus}{(s'_1,\ldots,s'_m) \in S}\ B[E[s'_1/s_1,\ldots,\ s'_m/s_m,\ f'_1/f_1,\ldots,f'_n/f_n]]$$

where the f'_i in E are defined by the fixed points above.

For lack of space we restrict ourselves to one particular rewrite rule for the operator ∇ . At first a condition for the maximality is given (note that the expression $(E1 \parallel E2)$ is never maximal)

$$\text{ismaximal}(E1) \wedge \text{ismaximal}(E2) \rightarrow \text{ismaximal}(E1 \nabla E2)$$

As a rewrite rule one may use

$$\neg\, \text{ismaximal}(E1 \nabla E2) \qquad \wedge$$
$$(E1 \rightarrow E1' \vee (\text{ismaximal}(E1) \wedge E1 = E1',)) \wedge$$
$$(E2 \rightarrow E2' \vee (\text{ismaximal}(E2) \wedge E2 = E2')) \rightarrow (E1 \nabla E2) \rightarrow (E1' \nabla E2') \ ,$$

$$e \in \text{DOM} \setminus \{\bot\} \rightarrow \quad (e \nabla E) \rightarrow e, \quad (E \nabla e) \rightarrow e \ ,$$

and in addition for streams

$$e \in \text{ATOM} \setminus \{\bot\} \rightarrow ((e \,\&\, s1) \nabla s2) \rightarrow (e \,\&\, s1), (s1 \nabla (e \,\&\, s2)) \rightarrow (e \,\&\, s2).$$

Complementing the computation rule of the preceding section by these rewrite rules leads to an effective, consistent and finitary rule, which is no longer complete, however. Note that replacing $E1 \rightarrow E1'$ and $E2 \rightarrow E2'$ by $E1 \overset{n}{\rightarrow} E1'$ and $E1 \overset{m}{\rightarrow} E1'$ resp., $n, m \in \mathbb{N} \setminus \{0\}$, where $E \overset{n}{\rightarrow} E'$ means: there exist expressions $E_1, \ldots, E_{n-1}$ such that $E \rightarrow E_1 \rightarrow \ldots \rightarrow E_{n-1} \rightarrow E'$ holds, immediately leads to an infinite family of effective, consistent and finitary rules, such that for each feasible value x there is a rule which possibly computes x . We like to talk about "loose implementations of nondeterminism" interpreting the notion "loose nondeterminism" in /Park 80/ (cf. also /Broy, Wirsing 81a/).

The introduction of the ∇-operator into the nondeterministic programming language essentially changes our notion of computability. According to the results of /Chandra 78/ now all sets in Σ_1^1 may occur as domains of nonterminating computations. This fact is no longer surprising, if one adapts the notion of *loose nondeterminism* (coined in /Park 80/) 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 multiset function

$$r : \text{PROGRAM} \to M(\text{DOM})$$

A function

$$f : \text{DOM}^n \to M(\text{DOM})$$

is called *definable* iff there exists a program $P(x_1,\ldots,x_n)$

$$\forall\, x_1,\ldots,x_n \in \text{DOM} :$$
$$\text{SET}(B[f(x_1,\ldots,x_n)]) = \{y \in \text{DOM} : \exists\, r \in CR : P(x_1,\ldots,x_n) \xrightarrow{r} y\}$$

Accordingly not *one* complete computation rule is considered, but an infinite number of computation rules, each of which is not complete, but which together are "complete". This properly models the (in principle) infinite number of implementations of a concurrent programming language, each of which is not required to be complete, but consistent, effective and (according to the impossibility to do an infinite number of decisions in finite time) finitary.

Note, that the introduction of (and)- and (or)- nodes into procedural programming languages as used in /Manna 70/ has some similarities to our ∇ - and $\|$-operator resp. However, in our nondeterministic programming language, we can prove $C[E1 \,\|\, E2] = C[E1] \,\|\, C[E2]$ for each context C, however $C[E1 \,\nabla\, E2] = C[E1] \,\nabla\, C[E1]$ does not hold, while in /Manna 70/ one always has C[S1 (or) S2] = C[S1] (or) C[S2] as well as C[S1 (and) S2] = C[S1] (and) C[S2]. For this simpler case with more convenient algebraic properties, /Chandra 74/ proves that both concepts are independent, i.e. that (or) cannot be expressed by (and) and vice versa. In our language we have: if tt ∇ ff then E1 else E2 fi is equivalent to (E1 $\|$ E2)

and not to E1 ∇ E2 , which would be the case if one systematically translated the concept of alternation (as the other approach is called in /Chandra et al. 81/) into our language.

Note, that the ambiguity operator ∇ can also be used to specify the logical "parallel or" (and "parallel and"):

```
funct paror = λ x, y :
                  if x then tt else y fi ∇ if y then tt else x fi
```

Interestingly the function paror can be viewed as a determinate function (in the powerdomain approach) with paror(tt,x) = paror(x, tt) = tt, paror(ff,ff) = ff, paror($\perp$, $\perp$) = $\perp$ for all x $\in$ {tt, ff, $\perp$}, while in the multidomain approach it is nondeterministic.

In a similar way the ambiguity operator ∇ can be used to specify disjunctive (multiple) waiting. Given a guarded wait command:

$$\underline{\text{await}}\ B_1\ \underline{\text{then}}\ E_1\ \nabla\ \ldots\ \nabla\ B_n\ \underline{\text{then}}\ E_n\ \underline{\text{endwait}}$$

which can be defined by

$$(\lambda\ b_1, \ldots, b_n:\ \underline{\text{if}}\ b_1\ \nabla\ \tilde{B}_1\ \underline{\text{then}}\ E_1\ \underline{\text{else await}}\ b_2\ \underline{\text{then}}\ E_2\ \nabla\ \ldots\ \nabla\ b_n\ \underline{\text{then}}\ E_n\ \underline{\text{endwait fi}})\ (B_1, \ldots, B_n)$$

where

$$\tilde{B}_1 = \underline{\text{if}}\ b_2\ \underline{\text{then}}\ \text{ff}\ \underline{\text{else}}\ \perp\ \underline{\text{fi}}\ \nabla\ \ldots \nabla\ \perp\ b_n\ \underline{\text{then}}\ \text{ff}\ \underline{\text{else}}\ \perp\ \underline{\text{fi}}$$

One proves easily, that in the await - construct the pairs (B_i 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.

Example: Disjunctive Waiting

```
⌈funct table ≡ λ s : g(first s) & table (rest s),
 stream  s1  ≡ table(1 & s1), stream s2 ≡ table(2 & s2), merge(s1,s2)⌋
```

where

```
 funct merge ≡ λ s1, s2 :
    await ¬ isempty s1 then (first s1)& merge(rest s1,s2)
     ∇   ¬ isempty s2 then (first s2) & merge(s1, rest s2)
     ∇     isempty s1 ∧   isempty s2 then empty          endwait
```

and g is some recursive or primitive function. The result of this program is an infinite stream, iff at least one of the streams s1 or s2 is infinite. Note that it is not a fair merging in the sense of /Keller 78/, since if both s1 and s2 are infinite, s1 as well as s2 may be a possible result, where s2 (or s1 resp.) do not contribute anything. Fair merging seems only to be a fair concept (cf. /Broy 81a/), if we switch to real time processing.

end of example

Our system is completely free of *global nondeterminism* , i.e. all decisions can be made 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. /Milner 80a/) or Hoare's CSP (cf. /Hoare 78/) the 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 "rendezvouz" - 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 /Francez, Rodeh 80/ 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.

6.2. 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'

$$f' : \mathbb{N}^n \rightarrow \mathbb{N}^\perp$$

where

$$f'(x_1,\ldots,x_n) = \begin{cases} f(x_1,\ldots,x_n) & \text{if } f(x_1,\ldots,x_n) \text{ is defined} \\ \perp & \text{otherwise} \end{cases}$$

Here $\perp$ is just introduced as a symbol for "undefined". Now let us consider a function

$$g : (\mathbb{N}^\perp)^n \rightarrow \mathbb{N}^\perp$$

The function g is called strict if

$$g(x_1, \ldots, x_n) = \perp \quad \text{if} \quad x_i = \perp \text{ for some } i,\ 1 \le i \le n,$$

otherwise g is called 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 /Vuillemin 75/, page 55, we define:
g is called sequential, if (g is constant (e.g. $n=0$)) or if there is an index i, $1 \le i \le n$, such that

(1) for all $x_1, \ldots, x_n \in \mathbb{N}^\perp$:

$$x_i = \perp \Rightarrow \quad g(x_1, \ldots, x_n) = \perp$$

(2) for all $x \in \mathbb{N}^\perp$:

$$h_x : (\mathbb{N}^\perp)^{n-1} \to \mathbb{N}^\perp \quad \text{where} \quad h_x(y_1, \ldots, y_{n-1}) = g(y_1, \ldots, y_{i-1}, x, y_i, \ldots, y_{n-1})$$

is a sequential function

otherwise g is called nonsequential.

Clearly strict functions are sequential. Classical examples for non-strict but sequential functions are if-then-else, "sequential and" and "sequential or" (cf. /Bauer, Wössner 81/). Classical examples for non-sequential (but monotonic) functions are "parallel or" and "parallel and". Generally one considers only monotonic functions, which (in the case of flat domains) is equivalent to considering functions with regular tables as defined in /Kleene 52/, page 334:

A function

$$g : (\mathbb{N}^\perp)^n \to \mathbb{N}^\perp$$

has a regular table, if for all i, $1 \le i \le n$, $x_1, \ldots, x_{n-1} \in \mathbb{N}^\perp$ with $g(x_1, \ldots, x_{i-1}, \perp, x_i, \ldots, x_{n-1}) \neq \perp$ the function h where

$$h : \mathbb{N}^\perp \to \mathbb{N}^\perp$$

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):

$$I : EXP \to \mathbb{N}^\perp$$

(for simplicity we consider just arithmetic functions) is defined by $I[E] = x$ iff $B[E] = \{x\}$. Of course, I associates a function f

$$f : (\mathbb{N}^{\perp})^n \to \mathbb{N}^{\perp}$$

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 iff 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. /Plotkin 77/, /Friedman, Wise 78b/), defined by

$$I[\underline{\text{nif}}\ C\ \underline{\text{then}}\ E1\ \underline{\text{else}}\ E2\ \underline{\text{fi}}] = \begin{cases} I[E1] & \text{if } I[C] = tt \vee I[E1] = I[E2] \\ I[E2] & \text{if } I[C] = ff \vee I[E1] = I[E2] \\ \perp & \text{otherwise} \end{cases}$$

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

Note, that this conditional nif is the deterministic counterpart to

$$\underline{\text{funct}}\ \text{nif} = \lambda c, e1, e2 :\ \underline{\text{if}}\ e1 = e2\ \underline{\text{then}}\ e1\ \underline{\text{else}}\ \perp\ \underline{\text{fi}} \nabla \underline{\text{if}}\ C\ \underline{\text{then}}\ e1\ \underline{\text{else}}\ e2\ \underline{\text{fi}}$$

i.e. for deterministic expressions C, E1, E2 we have

$$\{I[\underline{\text{nif}}\ C\ \underline{\text{then}}\ E1\ \underline{\text{else}}\ E2\ \underline{\text{fi}}]\} = \text{SET}(B[\text{nif}(C,E1,E2)]).$$

Nonsequential functions are of interest in multiprogramming for several reasons: They need an at least quasi-parallel evaluation (see de-

mand-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

$$g : (\mathbb{N}^{\perp})^n \to \mathcal{P}(\mathbb{N}^{\perp})$$

the definitions can be extended in a straightforward way:

g is called nd-strict, if

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

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

(1) for all $x_1, \ldots, x_n \in \mathbb{N}^{\perp}$:

$$x_i = \perp \Rightarrow \perp \in g(x_1, \ldots, x_n)$$

(2) for all $x \in \mathbb{N}$:

$$h_x : (\mathbb{N}^{\perp})^{n-1} \to \mathcal{P}(\mathbb{N}^{\perp})$$

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.

Note, that in /Päppinghaus, Wirsing 81/ nd-sequential functions are called "hereditarily guarded".

We define: g has a nd-regular table, if for all i, $1 \le i \le n$, $x_1, \ldots, x_{n-1} \in \mathbb{N}$ with $\perp \notin g(x_1, \ldots, x_{i-1}, \perp, x_i, \ldots, x_{n-1})$ the function h where

$$h : \mathbb{N}^{\perp} \to \mathcal{P}(\mathbb{N}^{\perp})$$

where

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

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 "[]" 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 "∇", 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 define directly 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 /Manna et al. 73/ 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.

7. Nonconventional Computational Models

For the classical "von Neuman" 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 Neuman architecture is fundamentally based is an optimized, strict, innermost evaluation principle for simple tail-recursive, sequential functions (cf. /Broy 80a/, p. 138). Only this way the classical notion of program variables can originate (cf. /Landin 65/; for a comprehensive discussion see /Bauer, Wössner 81/). All extensions to parallel executions such as multiprocessors, 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 Neuman 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 E1 is started during the evaluation of a given expression E(E1), only if it turns out that the value of E1 is actually needed for the computation of the value of E(E1). 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 /Wadsworth 71/, delayed evaluation /Vuillemin 74/ and finally lazy evaluation /Henderson, Morris 76/.

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 non-sequential 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 counted 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.

7.1. Reduction

Reduction is probably the most general way to describe operational semantics. It simply corresponds to textual substitution rules. So it describes a concrete implementation of the process reducing the program given in one particular representation by a number of steps into some "normal form" (cf. "textual substitution machine" and "Herbrand-Kleene-machine" in /Bauer, Wössner 81/, p. 51/).

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

where the term rewriting process for some applicative language is implemented directly. In all these approaches the key question centres arround 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 (/Treleaven et al. 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. /Berkling 75/, /Mago 79/).

In graph reduction one introduces "references" to the resp. 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. /Keller 80/, /Turner 79/). If no further accesses may occur the corresponding reference and value may be deleted ("garbage collection"). Graph reduction is considered just suited to outermost mechanisms (cf. /Treleaven et al. 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 -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 functing 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 /Broy 80a/, 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 /Manna et al. 73/ 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 innnermost for classifying computation rules. If one allows λ-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 /Ershov 78/). In particular one may apply the rule to the bodies of the recursively defined functions to get optimized versions (cf. /Turner 79/, /Bauer, Wössner 81/).

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 seize 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 a most efficient way.

As far as no infinite, recursively defined objects occur the program expressions can be represented as trees (cf. Kantorovic-tree in /Bauer, Wössner 81/). Certain reduction steps may simplify and decrease the seize of the trees, whereas other steps (like unfold for recursively defined functions) may increase the seize of the trees. If the evaluation process is to be executed on a network of processors (for instance on a cellular automaton, cf. /Mago 79/), 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 79/ suggests to use combinators from combinatory logic for this purpose. /Berkling 75/ uses an organisation of three stacks through which linearized versions of the trees are pushed and stepwise evaluated (reduced).

7.2. Data Flow Concepts

A concept that works with a static graph and therefore is much closer to concrete hardware architectures is the data flow concept. May be 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 /Tesler, Enea 68/. 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 at most to 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. /Bauer, Wössner 81/). 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. /Kosinski 73, 77/, /Dennis 74/, /Arvind, Gostelow 78/), but at least in the case of nondeterminism the formal definition of their semantics has not been solved satisfactory so far. Therefore a completely formal definition of the semantics of a simple graphical data flow language is 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 $I : V \to A^*$, $O : A \to V \cup \{IN\}$
- a labelling function $L : V \to SPF$
- a subset $OUT \subseteq A$.

where 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.

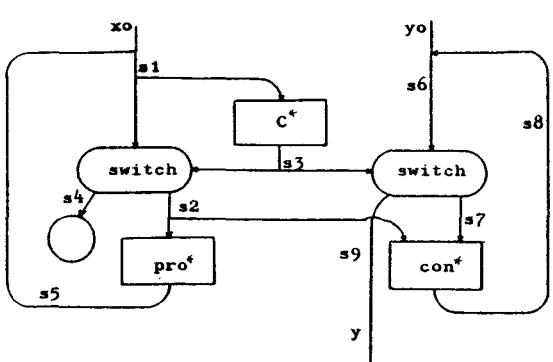
The meaning of a data flow program is given by the set of mutually recursive stream equations

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

for each arc a with $a = O(x)$, $f_a = L(x)$, $(a_1 \ldots a_n) = I(x)$.

For each set of input streams thus a recursively defined system of streams is given.

Example: With the following simple data flow graph:



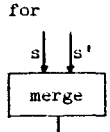
one associates the mutually recursive system of streams:

```
stream s1 = merge(sx_0, s5) ,
stream s2 = nfilter(s3, s1) ,
stream s3 = C*(s1) ,
stream s4 = pfilter(s3, s1) ,
stream s5 = pro*(s2) ,
stream s6 = merge(sy_0, s8) ,
stream s7 = nfilter(s3, s6) ,
stream s8 = con*(s2, s7) ,
stream s9 = pfilter(s3, s1) .
```

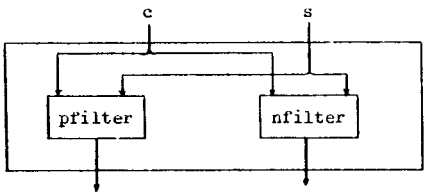
where

```
funct pfilter = λ c, s : if isempty(c) then empty else
      if first c then first s & pfilter(rest c, rest s)
                 else filter(rest c, rest s)          fi fi ,

funct C* = λ s : if isempty(s) then empty
                               else C(first s) & C*(rest s) fi ,

funct nfilter = λ c, s : if isempty(c) then empty else
      if first c then nfilter(rest c, rest s)
                 else first s & nfilter(rest c, rest s)  fi fi ,

funct pro* = λ s : if isempty(s)  then empty
                                  else pro(first s) & pro (rest s) fi ,

funct con* = λ s1, s2 : if isempty(s1)   isempty(s2)
                        then empty
                        else con(first s1, first s2) &
                                        con*(rest s1, rest s2) fi .
```

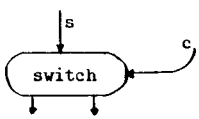
Note, that the harmless looking junction  of arcs is used as an abbreviation for



where merge is the "nonstrict" form using the ∇-operator (see page 56). For the filtering network



we use the special abbreviation (following the notation of /Kosiniski 73/)



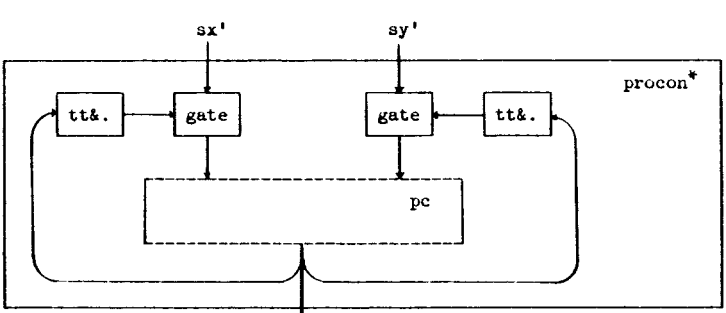
The data flow graph implements the function procon:

funct procon = λ x, y :
if C(x) then y else procon(pro(x), con(x, y)) fi

If we initialize $sx_0 = x \,\&\, \perp$, $ay_0 = y \,\&\, \perp$, then procon(x, y) = first s9. 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

funct procon* = λ sx, sy :
procon(first sx, first sy) & procon*(rest sx, rest sy)

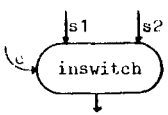
is implemented by the following net:



```
stream sxo =  gate(tt&sy, sx'),
stream syo =  gate(tt&sy, sy'),

funct gate = λ s1, s2 :
      if isempty(s1) or isempty(s2)
         then empty else first s2 & gate( rest s1, rest s2) fi
```

This frame guarantees, that a "new" value enters the flow network only when the old computation has finished.

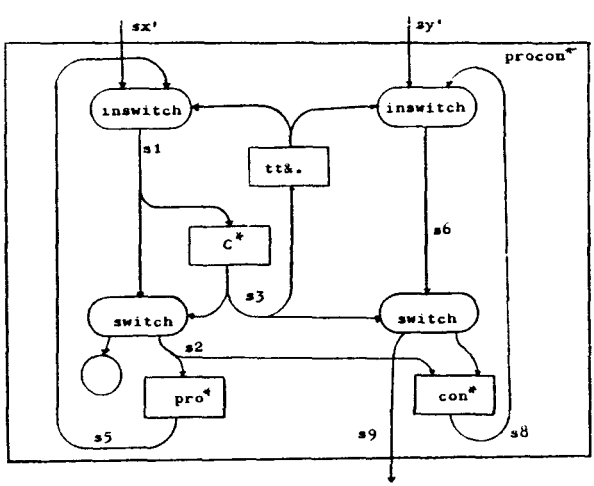
However, the gate can also be combined with the merge to an "inbound switch"



where

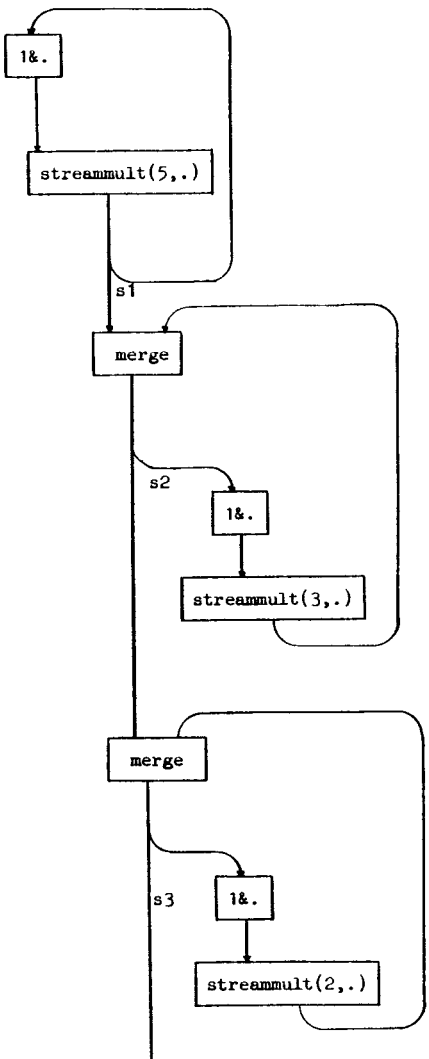
```
funct inswitch = λ c, s1, s2:
   if first c then first s1 & inswitch( rest c, rest s1, s2)
              else first s2 & inswitch( rest c, s1, rest s2) fi.
```

Thus we obtain the deterministic data flow program:



end of example

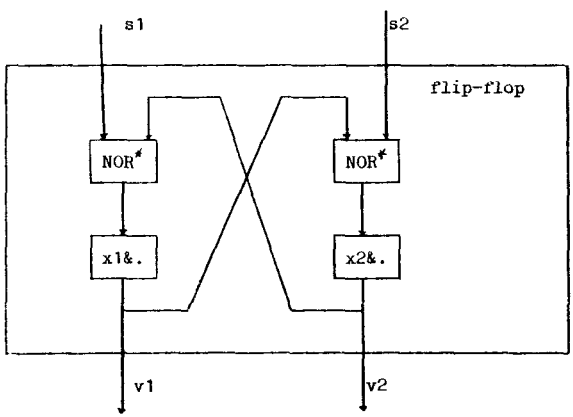
Vice versa a mutually recursive system of stream equations defines a data flow program. The one for the example at the end of section 5 reads:



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 like:

Example: Flip-Flops (Bistable Circuits)

A representation of an abstract flip-flop by a data flow network looks like (cf. /Bauer et al. 65/, p. 272):



where

funct nor* = λ s1, s2 : $\neg$ (first s1 $\vee$ first s2) & nor*(rest s1, rest s2).

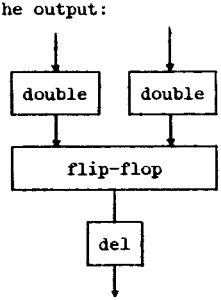
The semantic definition for data flow nets gives immediately

stream v1 = x1 & nor*(s1, v2) ,
stream v2 = x2 & nor*(s2, v1) .

The boolean values x1 and x2 correspond the initialization of the flip-flop (of course we assume $x1 \neq x2$).

first s1	first s2	first v1	first v2	first rest v1	first rest v2	
0	0	0	0	L	L	forbidden
0	0	0	L	0	L	read
0	0	L	0	L	0	
0	0	L	L	0	0	excluded internal state
0	L	0	0	L	0	
0	L	0	L	0	0	set left
0	L	L	0	L	0	
0	L	L	L	0	0	excluded internal state
L	0	0	0	0	L	
L	0	0	L	0	L	set right
L	0	L	0	0	0	
L	0	L	L	0	0	excluded internal state
L	L	0	0	0	0	
L	L	0	L	0	0	forbidden
L	L	L	0	0	0	input
L	L	L	L	0	0	

However, as can be seen from the table above the function does not react like 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:



where

```
funct double = λ s : first s & first s & double(rest s)
funct del =    λ s : 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 times 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.

endofexample

Of course, it is also possible to design hierarchical data flow networks, i.e. networks where certain which are again (hierarchical) data flow networks. We even may 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.

7.3. 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, analysed, and their particular role in stream processing has to be figured out. It seems impossible to do this in this framework in a comprehensive way, but nevertheless a first brief approach is given, showing in which way such a classification can be done.

Let

$$SMF_n \subseteq STREAM^n \to \mathcal{P}(STREAM)$$

where $f \in SMF_n$ iff f is submonotonic, i.e. for all $s_i, s_i' \in STREAM^n$:

$$\text{If } s_i \sqsubseteq s_i' \text{ for } 1 \le i \le n \text{ then}$$
$$\forall\, y \in f(s_1,\ldots,s_n)\ \exists\, y' \in f(s_1',\ldots,s_n') : y \sqsubseteq y'$$

All recursively defined functions

$$\tilde{f} : STREAM^n \to M(STREAM)$$

of our language define submonotonic functions f by

$$f(s_1, \ldots, s_n) = SET(\tilde{f}(s_1, \ldots, s_n)).$$

The resumption res of a n-ary submonotonic stream-processing function

$$res : SMF_n \times STREAM^n \to SMF_n$$

is defined by

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

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

$$x \oplus y = \begin{cases} x & \text{if } x \text{ is total (finite or infinite)} \\ x' \circ y & \text{if } x = x' \circ \langle \perp \rangle \end{cases}$$

Obviously res and $\oplus$ are homomorphic in the following sense:

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

The validity of this equation follows immediately from the submonotonicity property of the stream processing functions.

Resumptions can be used very conveniently to discuss properties of submonotonic stream processing functions.

At first we want to look at synchronous functions $f \in \mathrm{SMF}_n$. f is called <u>synchronous</u> iff

$$\mathrm{length}(y) = \min(\mathrm{length}(s_1), \ldots, \mathrm{length}(s_n))$$

for all $y \in f(s_1, \ldots, s_n)$ where

$$\mathrm{length}(\varepsilon) = \mathrm{length}(\perp) = 0$$
$$\mathrm{length}(\mathrm{ap}(e, s)) = 1 + \mathrm{length}(s)\ .$$

f is called <u>k-local</u>, if for all partial $s_i \in \mathrm{STREAM}$, $\mathrm{length}(s_i) = k$, $1 \le i \le n$:

$$\mathrm{res}(f, s_1, \ldots, s_n) = f$$

otherwise f is called <u>accumulative</u>.

Of course every n-ary function g on atoms defines a 1-local stream processing function g^* by

$$\underline{\text{funct}}\ g^{*} = \lambda\ s_1, \ldots, s_n :$$
$$\quad \underline{\text{if}}\ \text{isemtpy}(s_1)\ \underline{\text{or}}\ \ldots\ \underline{\text{or}}\ \text{isempty}(s_n)$$
$$\quad\quad \underline{\text{then}}\ \text{empty}$$
$$\quad\quad \underline{\text{else}}\ g(\underline{\text{first}}\ s_1, \ldots, \underline{\text{first}}\ s_n)\ \&\ g^{*}(\underline{\text{rest}}\ s_1, \ldots, \underline{\text{rest}}\ s_n)\ \underline{\text{fi}}$$

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

$$\underline{\text{funct}}\ \overline{g}_e = \lambda\ s_1, \ldots, s_n : h(s_1, \ldots, s_n, e),$$
$$\underline{\text{funct}}\ h = \lambda\ s_1, \ldots, s_n, x :$$
$$\quad \underline{\text{if}}\ \text{isempty}(s_1)\ \underline{\text{or}}\ \ldots\ \underline{\text{or}}\ \text{isempty}(s_n)$$
$$\quad\quad \underline{\text{then}}\ \text{empty}$$
$$\quad\quad \underline{\text{else}}\ x\ \&\ h(\underline{\text{rest}}\ s_1, \ldots, \underline{\text{rest}}\ s_n,$$
$$\quad\quad\quad\quad g(\underline{\text{first}}\ s_1, \ldots, \underline{\text{first}}\ s_n, x))\ \underline{\text{fi}}\ .$$

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.

Example: We consider the accumulative function $\overline{\overline{\text{add}}}_0$, defined by

$$\underline{\text{funct}}\ \overline{\overline{\text{add}}}_0 = s : \text{aux}(s, 0),$$
$$\underline{\text{funct}}\ \text{aux} = s, x : x\ \&\ \text{aux}(\underline{\text{rest}}\ s, x + \underline{\text{first}}\ s).$$

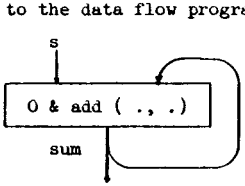
This definition can be used in a nonrecursive stream definition:

$$\underline{\text{stream}}\ \text{sum} = \overline{\overline{\text{add}}}_0(s).$$

We have $\overline{\overline{\text{add}}}_0(s) = 0\ \&\ \text{add}^{*}(s, \overline{\overline{\text{add}}}_0(s))$ and we obtain:

$$\underline{\text{stream}}\ \text{sum} = 0\ \&\ \text{add}^{*}(s, \text{sum}).$$

That corresponds to the data flow program:



In this example the recursion in the definition of $\overline{add}_o$ is transformed into a recursion for the stream sum (cf. also /Bauer, Wössner 81/, p. 295).

end of example

The operator turning a function g into the stream-processing function $\overline{g}_e$ may be seen as an example of a combinator for writing such functions without explicit recursion:

Example: Given the infinite stream $s = 1 \& s$ which consists of an infinite sequence of 1 and addition add and multiplication mult, then

$$\overline{mult}_1(\overline{add}_1(s))$$

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 like suggested in /Backus 78/.

end of example

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

Example: LIFO-nodes. Since streams are very similar to stacks, it is very easy to program a LIFO-like mechanism

```
funct reverse = λ s : store(s, empty)
funct store = λ s1, s2 :
      if B(first s1) then first s1 & clear(rest s1, s2)
                     else store(rest s1, first s1 & s2) fi

funct clear = λ s1, s2 :
      if isempty(s2) then reverse(s1)
                     else first s2 & clear(s1, rest s2) fi
```

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

$$reverse(s) = (s_o \& (s_1 \& (\ldots(s_n \& reverse(s')\ldots) .$$

The function reverse can be used to transform general linear recursion into tail-recursion:

```
funct f = λ x :  if B(x) then T(x) else g(f(h(x)), x) fi
```

using the auxiliary functions:

```
funct down = λ x :
      if B(x) then x & empty else x & down(h(x)) fi ,

funct up = λ s, x, y :
      if first s = x then y
                     else up(rest s, x, g(y, first rest s)) fi
```

We have:

```
f(x) = up(reverse(down(x)), x, T( first reverse(down(x)) )),
```

or eliminating the common subexpression:

```
f(x) =  (λ t : up(t, x, T(first t))(reverse(down(x))
```

and

```
f*(s) = (λ t : up'(t, s, T(first t))(reverse(down'(s)))
```

where

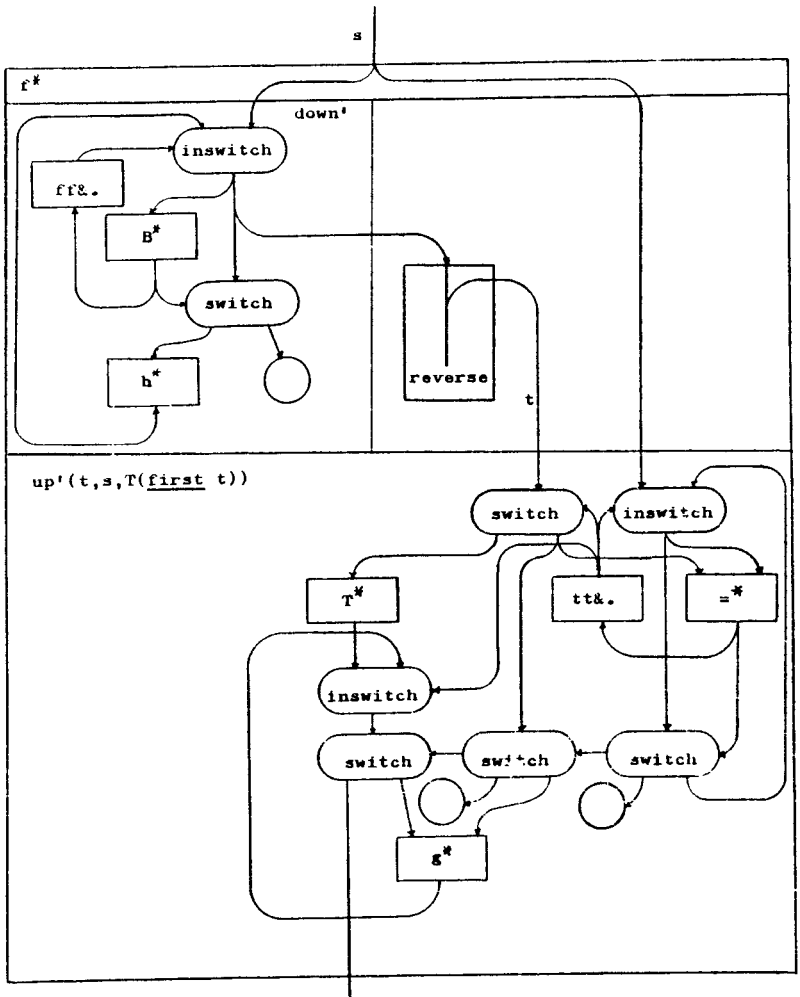
```
funct down' = λ s:
      if B(first s) then first s & down'(rest s)
                    else first s & down'(h(first s) & rest s) fi ,

funct up' = λ s, sx, y :
      if first s = first sx
         then y & up'(rest s, rest sx, T(first rest s))
         else up'(rest s, sx, g(y, first rest s))        fi
```

In particular the following equation holds:

```
reverse(down'(e & s)) = reverse(down(e)) conc reverse(down'(s)).
```

Here conc denotes the concatenation of streams.

Based on this functions and some additional simple transformations one obtains a general data flow scheme for implementing linear recursion:


s
f*
down'
inswitch
ff&.
B*
switch
h*
reverse
t
up'(t,s,T(first t))
switch
inswitch
T*
tt&.
=*
inswitch
switch
switch
switch
g*

7.4. Comparison to the Conventional Procedural Computational Model

For understanding the advantages and drawbacks of nonconventional (i.e. non-von-Neuman) 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:

$$\lceil \underline{\text{funct}}\ f = \lambda\ x : \underline{\text{if}}\ C(x)\ \underline{\text{then}}\ f(F(x))\ \underline{\text{else}}\ T(x)\ \underline{\text{fi}}\ ,\ f(E_o) \rfloor$$

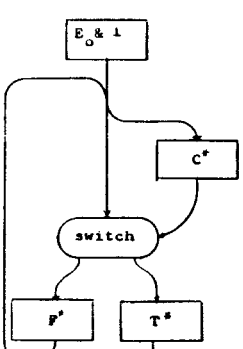
Applying our parallel computation rule to $f(E_o)$ we may obtain for instance

$$f(E_o) \rightarrow \ \ldots$$
$$(\lambda\ x : \underline{\text{if}}\ C(x)\ \underline{\text{then}}\ f(F(x))\ \underline{\text{else}}\ T(x)\ \underline{\text{fi}})(e_o) \rightarrow$$
$$\underline{\text{if}}\ C(e_o)\ \underline{\text{then}}\ f(F(e_o))\ \underline{\text{else}}\ T(e_o)\ \underline{\text{fi}} \rightarrow \qquad \ldots$$
$$T(F(\ldots\ F(e_o)\ \ldots))$$

A corresponding while program looks like

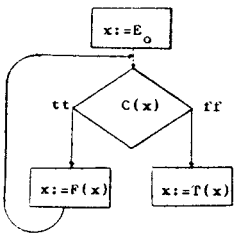
```
x := E_o;
while C(x) do x := F(x) od ;
x := T(x);
```

A data flow version may look like

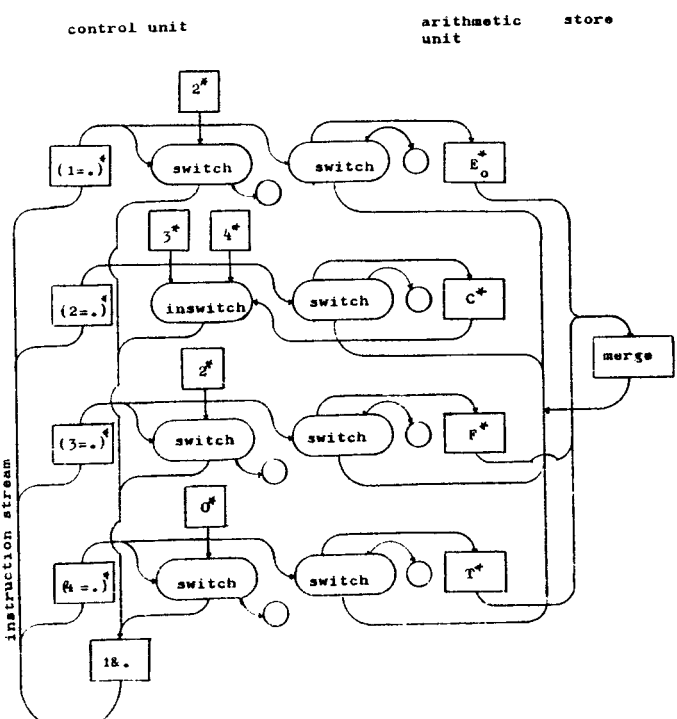



```
stream sx = merge(E_o & ⊥, F*(sy)) ,
stream sy = pfilter(sb, sx) ,
stream sb = C*(sx) ,
stream sz = nfilter(sb, sx) ,
stream r  = T (sz) .
```

The control-flow diagram of the procedural version has the form



The procedural program may also be represented by a data flow program with explicit control arcs:



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 resp. label. The corresponding instruction is activated then and the label of the successive instruction is added to the instruction stream. The actual data flow consists trivially of a merge-node for the variable x with input arcs from all nodes corresponding to assignments to x and ouput arcs to all nodes corresponding to statements evaluating x. An assignment x := f(x) is then replaced by a node f^*(filter(c, sx)) where c is the control stream and sx is the stream of the variable. If the control stream contains ff the corresponding value of x is simply ignored, otherwise it is taken; the node becomes "active".

With the program variable x the stream sx is associated (the output stream of the resp. 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. /Broy 80a/, p. 133, and /Bauer, Wössner 81/, p. 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 treatement, 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 the 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 /Broy 80b/ and /Broy 81c/) working on the streams which then have to be viewed as queues (for more on the duality between streams and queues see /Broy, Bauer 81/).

Only in very specific cases, where the streams between the communicating agents or more precisely the resp. 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.

8. 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 /Broy 81c/, /Broy, Bauer 81/). Nowadays a point is reached, where we actually are beginning to understand most of the phenomena of concurrent programs including their interrelationsships. To this understanding the current approach tries to contribute. In the following the relation to other work is briefly outlined. Finally a number of future research topics are briefly sketched that may be based on this approach.

8.1 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 attemps to overcome these difficulties can be found in the single assignment approach (cf. /Tesler, Enea 68/). This proposal was completed to data flow language concepts (cf. /Dennis 74/), based on the demand driven evaluation (cf. also /Backus 78/). Numerous papers have been published on this issue, few of them, however, containing much formal foundation. /Friedman, Wise 80a/ 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. /Kosinski 77/, /Kosinski 79/) or "scenarios" (cf. /Brock, Ackermann 81/).

A far developed theory is Milner's Calculus of Communicating Systems (CCS, cf. /Milner 73/, /Milne , Milner 77/, /Milner 80/). In CCS a communication mechanism is integrated into an applicative programming language. The communication in CCS follows the rendevouz principle (like CSP in /Hoare 78/) and therefore can be seen as the applicative counterpart of CSP.

A paper with much impact on the field of concurrent programming is /Kahn, MacQueen 77/ , where an approach is outlined which is very similar to the streams described in section 2.3 and section 5 (cf. also /MacQueen 79/). However Kahn's approach does not include nondeterminism nor concurrency in the sense of section 6 (cf. the discussion at the end of /Keller 78/).

In the field of nondeterministic applicative programming languages several papers on fixed point theory have been published (cf. for instance /Hennessy, Ashcroft 76, 77, 80/, /de Bakker 76/, /Arnold, Nivat 77/, /Broy et al. 78/, /Nivat 80/). For surprisingly few approaches to multiprogramming, however, attempts have been undertaken to give a fixed point semantics (cf. for instance /Arvind, Gostelow 78/, /Hewitt , Baker 78/, /Kosinski 79/).

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 /Broy, Wirsing 80/. 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.

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. The extension and application of this approach to more explicitly communication-oriented languages like CCS or to procedural languages challenges further investigations. Moreover, the consequences of the identifications of certain infinite multisets in multidomains are not completely understood by the author and deserve further attention.

In /Broy 82/ therefore a semantics is given for the applicative multiprogramming language which is using fixedpoint theory in a less conventional way (similar to the definition of the fixed points for functions containing the ambiguity operator in section 6). Instead of using multidomains and least fixed points, the fixed points are characterized by a combination of several orderings and domains.

8.2. 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 quaters to this question (cf. /Japan 81/) 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 located completely 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 of the future of 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 realized independently 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 com-

municating processes (using send/receive primitives as defined in /Broy 81c/). Appropriate transformation methods may be used to perform this realization in a proper way. Using the definitions of /Broy 81c/ considering the streams as queues at the procedural level (cf. /Broy, Bauer 81/) the verification techniques of Gries/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 nonstrict operations and infinite objects is one additional issue. Another important issue concern 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 6 precisely mirrors two concepts of program verification, the first "approximated" semantics using "$\Box$" instead of "$\forall$" is sufficient to prove partial correctness, i.e. an axiomatization of partial correctness can be based on this simple semantics. The second semantics defined by the $\subseteq$-maximal fixed points has to 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 socalled "rendezvous"-concept, where hand shaking communication is used, leading to tightly-coupled ("synchronous") systems (cf. /Milner 80b/). So 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 a stream processing function produces in every computation step either a real atom or a dummy date. This leads to strongly communicating processes, where, as well-known, a number of problems do not appear any longer (cf. the introduction of "silent communication" in /Milner 80a/ and the condition "is guarded" in /Milner 80b/, p. 10) just as for real-time processes (cf. /Broy 81a/).

An alternative to modelling communication by mutually recursively defined streams is the direct incorporation of communication primitives into an applicative language (cf. /Milner 80a/). 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,
- multi-function special purpose chips (logical design of special purpose hardware),
- storage as networks (active storage),
- networks with uniform multi-function 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.

Appendix: Proofs

(page 16, first lemma)

Proof: "<=": Define $G: SET(M1) \to \mathcal{M}(DOM)$ by

$$G(x) = \overset{(+)}{1 \le j \le M1(x)} \quad \overset{(+)}{(j,x) \le H(i,y)} \{y\}$$

where the function $H: S_{M2} \to S_{M1}$ is surjective and right-monotonic.

"=>": Since $M1(x) = G(x)$ there are surjective functions

$$h_x: \quad S_{G(x)} \to \{(n,x): 1 \le n \le M1(x)\}$$

which trivially are right-monotonic. Since

$$\overset{(+)}{y \in SET(M1)} \quad G(y) = M2$$

there exists a mapping

$$G': \quad S_{M2} \to SET(M1)$$

which is surjective and $G'(i,y) = x$ iff $y \in G(x)$. Define

$$H(i,y) = h_x(i,y) \qquad \text{for} \qquad x = G'(i,y).$$

(page 16, 2nd lemma)

Proof: (1) Reflexivity: The identity is surjective and right-monotonic.

(2) Transitivity: Composition of surjective, right-monotonic functions yields surjective, right-monotonic functions.

(page 16)

Proof: "$\sqsubseteq$" is a quasiordering.

(i) Antisymmetry: We prove by induction:

If $M1, M2 \in \mathcal{M}(FDOM)$ and both $H1: S_{M2} \to S_{M1}$ and $H2: S_{M1} \to S_{M2}$ are surjective and right-monotonic, then both H1 and H2 are the right-identity.

If $|M1| = 1$, this is trivial. Let $|M1| = n+1$ and x be a minimal element in M1. Then $H1(H2(1,x)) = (i,x)$ because of the right-monotonicity of H1, H2 and the minimality of x.

(ii) Minimality: For every $M \in \mathcal{FM}(\mathrm{FDOM})$ the function

$$H: S_M \to S_{\{\perp\}} \text{ whith } H(i,x) = (1,\perp)$$

is right-monotonic and surjective.

(page 18)

Proof: (1) For proving that for $M' \in I$: $M' \sqsubseteq M_I$ it is first shown:

(a) $\forall\, x \in M: \exists\, y \in M': \; y \sqsubseteq x$,

(b) $\forall\, M'' \subseteq M': \exists\, M3 \subseteq M: \; M'' \sqsubseteq M3$.

The proposition (a) follows immediately from

$$\operatorname*{glb}_{\substack{M1 \in I \\ y \sqsubseteq x \\ y \in \mathrm{FDOM}}} \operatorname{lub} \{M2(z): \; M1 \sqsubseteq M2 \in I \wedge y \sqsubseteq z \sqsubseteq x\} = M_I(x) > 0,$$

since otherwise the lub would be 0 for $M1 = M'$.

Due to the construction of the ideal completion there exists a $\sqsubseteq$-chain of multisets $\{M_i\}_{i \in \mathbb{N}}$, $M_i \in \mathcal{FM}(\mathrm{FDOM})$, such that the $I_i =_{\mathrm{def}} I_{M_i}$ form a $\subseteq$-chain and

$$I = \bigcup I_i .$$

So for $M'' = \{x_o^{(1)}, \ldots, x_o^{(n)}\}$ one may define chains $\{x_i^{(j)}\}_{i \in \mathbb{N}}$

$$\biguplus_{1 \le j \le n} \{x_i^{(j)}\} \subseteq M_i .$$

So one may choose

$$M3 = \biguplus_{1 \le j \le n} \operatorname{lub} \{x_i^{(j)}\} .$$

From (a) and (b) the existence of some

$$G: \mathrm{SET}(M') \to \mathcal{M}(\mathrm{DOM})$$

as required immediately follows.

Now let $M' \in I_{M'}$. For every $x \in M'$ there exists $M_x \in I$ with $x \in M_x$. Due to the fact that M' is finite and I is an ideal there exists $M_o \in I$ such that $M_x \sqsubseteq M_o$ for all $x \in M'$. Now a $\sqsubseteq$-chain

$\{M_i\}_{i \in \mathbb{N}}$, $M_i \in I$, can be constructed according to the following rules:

(A) If there exists $a \in M_i$ such that there does not exist $z \in M'$ with $z \sqsubseteq a$, then there exists $x \in M$, $z \in M'$ with $a \sqsubseteq x$ and $z \sqsubseteq x$. Since a and z are finite there exists $b \in FDOM$ such that $a \sqsubseteq b \sqsubseteq x$ and $z \sqsubseteq b$ and for all c with $c \sqsubseteq b$, $c \neq b$: $c \notin M$. Define $M_{i+1} =_{def} (M_i \cup \{a\}) \dot{\cup} \{b\}$. Obviously $M_i \sqsubseteq M_{i+1}$ and $M_{i+1} \in I$.

(B) Otherwise, if there exists $M'' \leq M'$ such that for all $M3 \leq M_i$ the relation $M'' \sqsubseteq M3$ does not hold, then there exists an multiset $\overline{M} \in I$, such that there exists $M3 \leq \overline{M}$ with $M'' \sqsubseteq M3$. Now choose $M_{i+1} \in I$, such that $\overline{M} \sqsubseteq M_{i+1}$ and $M_i \sqsubseteq M_{i+1}$, which can be done , because I forms an ideal.

Note, that if the condition of (A) is valid once, then (B) will never produce a multiset M_i, such that the condition of (A) does not hold for M_i. Because M' and M_o are finite the construction ends after a finite number of steps with some M_i such that $M' \sqsubseteq M_i$. Because I is an ideal thus $M' \in I$, too.

(2) If the set I in the definition of M_I is increased then either $M_I(x)$ remains unchanged, or $M_I(x)$ is increased, or $M_I(x)$ is decreased by $n \in \mathbb{N}$, but for some y_i, $1 \leq i \leq n$, $x \sqsubseteq y_i$, $M_I(y_i)$ is increased by

$$(\dot{\cup} \{y_i\})(y_i).$$

(page 19, first lemma)

Proof: (1) If $x1 \sqsubseteq x2$, then $f(x1) \sqsubseteq f(x2)$ and so $\{f(x1)\} \sqsubseteq \{f(x2)\}$, i.e. $F(x1) = F(x2)$.

(2) If $\text{lub}\{x_i\} = x$ then

$$F(x) = \{f(x)\} = \{\text{lub}\{f(x_i)\}\} = \text{lub}\{\dot{\cup} f(x_i)\}\} = \text{lub}\{F(x_i)\}.$$

(page 19, 2nd lemma)

Proof: In the ideal representation the multiset-sum for ideals I1 and I2 is defined by

$$I1 \,(+)' \, I2 = \{ M \in \mathcal{M}(FDOM) : \exists\ M1 \in I1,\ M2 \in I2 :\ M \sqsubseteq M1 \,(+)\, M2 \}\ .$$

(1) Monotonicity simply follows from the monotonicity of the multiset sum on finite elements:
Let $X1, Y1, X2, Y2 \in \mathcal{IM}(DOM)$, $X1 \sqsubseteq X2$, $Y1 \sqsubseteq Y2$. Then there exist $G1: SET(X1) \to \mathcal{M}(DOM)$ and $G2: SET(X2) \to \mathcal{M}(DOM)$, such that the conditions of the definition of $\sqsubseteq$ are fulfilled. Define

$$G:\ SET(X1 \,(+)\, X2) \to \mathcal{M}(DOM)$$

by $G(x) = G1(x) \,(+)\, G2(x)$. Then G fulfills the required conditions.

(2) Now let $\{X_i\}_{i \in \mathbb{N}}$, $\{Y_i\}_{i \in \mathbb{N}}$ be chains of ideals and

$$X =_{def} \bigcup X_i,\quad Y =_{def} \bigcup Y_i,\quad Z =_{def} \bigcup (X_i \,(+)'\, Y_i).$$

Trivially the monotonicity of the multiset sum gives

$$Z = X \,(+)'\, Y.$$

Now let $M \in X \,(+)'\, Y$; then $M \sqsubseteq M1 \,(+)\, M2$ for some $M1 \in X$, $M2 \in Y$. Thus $M1 \in X_i$ and $M2 \in Y_j$ for some $i, j \in \mathbb{N}$. Thus

$$M \in X_k \,(+)'\, Y_k \quad \text{for} \quad k =_{def} \max(i, j).$$

Hence $M \in Z$.

(page 20)

Proof: In ideal representation

$$T, \Gamma\ :\ (DOM \to \mathcal{ID}(DOM)) \to (DOM \to \mathcal{ID}(DOM))$$

the ideal $\Gamma[f](x)$ reads:

$$\{ M \in \mathcal{IM}(FDOM):\quad M' \in T[f](x),\ M_y \in f(y):\ M \sqsubseteq \mathop{(+)}_{y \in M'} M_y \}$$

(1) Then the monotonicity trivially follows from the monotonicity of the finite multiset-sum.

(2) Monotonicity immediately gives:

$$\Gamma[f_i](x) \subseteq \Gamma[f](x)$$

for continuous chains $\{f_i\}_{i \in \mathbb{N}}$ of functions

$$f_i: \ DOM \to_c ID(FDOM), \quad \text{where} \quad f = \bigcup f_i .$$

If $M \in T[f](x)$, then there exists $M' \in T[f](x)$ and thus $M' \in T[f_i](x)$ for some $i \in \mathbb{N}$, such that

$$M \sqsubseteq \overset{(+)}{\underset{y \in M'}{\bigcup}} M_y \quad , \quad \text{where} \quad M_y \in f(y).$$

Since M' is finite, there exists j such that for all $y \in M'$: $M_y \in f_j(y)$ and thus $M \in T[f_j](x)$, too.

References

/Apt, Plotkin 81/
K.R. Apt, G. Plotkin: A Cook's Tour of Countable Nondeterminism. In: S. Even, O. Kariv (eds.): 8th International Colloquium on Automata, Languages and Programming, Haifa 1981, Lecture Notes in Computer Science 115, Berlin - Heidelberg - New York: Springer 1981, 479-494

/Arnold 79/
A. Arnold: Operational and Denotational Semantics of Nets of Processes. Universite P. et M. Curie, Universite Paris 7, Laboratoire Informatique Theoretique et Programmation, LITP Report No. 79-35, June 1979

/Arnold, Nivat 77/
A. Arnold, M. Nivat: Nondeterministic Recursive Program Schemes In: M. Karpinski (ed): Fundamentals of Computation Theory. Lecture Notes in Computer Science 56, Berlin - Heidelberg - New York: Springer 1977, 12-21

/Arvind, Gostelow 78/
Arvind, K.P. Gostelow: Some Relationships between Asynchronous Interpreters of a Dataflow Language. In: /Neuhold 78/, 96-119

/Astesiano, Costa 79/
E. Astesiano, G. Costa: Sharing in Nondeterminism. In: H.A. Maurer (ed.): 6th Int. Coll. on Algorithms, Languages and Programming. Lecture Notes in Computer Sciences 71, Berlin - Heidelberg - New York: Springer 1979, 1-13

/Backus 78/
J. Backus: Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Comm. ACM 21:8, August 1978, 613-641

/de Bakker 76/
J.W. de Bakker: Semantics and Termination of Nondeterministic Recursive Programs. In: S. Michaelson, R. Milner (eds.): Proc. of the 3rd International Colloquium on Automata, Languages and Programming, Edinburgh: Edinburgh University Press 1976, 435-477

/Bauer 79/
F.L. Bauer: Detailization and Lazy Evaluation, Infinite Objects and Pointer Representation. In: /Bauer, Broy 79/, 235-236

/Bauer, Broy 79/
F.L. Bauer, M. Broy (eds.): Program Construction. Lecture Notes of the International Summer School on Program Construction, Marktoberdorf 1978. Lecture Notes in Computer Science 69, Berlin - Heidelberg - New York: Springer 1979

/Bauer et al. 65/
F.L. Bauer, J. Heinhold, K. Samelson, R. Sauer: Moderne Rechenanlagen. Stuttgart: B.G. Teubner Verlagsgesellschaft 1965

/Bauer, Wössner 81/
F.L. Bauer, H. Wössner: Algorithmische Sprache und Programmentwicklung. Berlin - Heidelberg - New York: Springer 1981

/Benson 79/
D.B. Benson: Parameter Passing in Nondeterministic Recursive Programs. Journal of Computer and System Sciences 19, 1979, 50-62

/Berkling 75/
K. Berkling: Reduction Machines for Reduction Languages. Proc. Second Int. Symp. Computer Architecture, 1975, 133-140

/Brock, Ackermann 81/
J.D. Brock, W.B. Ackermann: Scenarios: A Model of Non-determinate Computation. In: J. Diaz, I. Ramos (eds.): Formalization of Programming Concepts, Peniscola 1981, Lecture Notes in Computer Science 107, Berlin - Heidelberg - New York: Springer 1981, 252-267

/Broy 80a/
M. Broy: Transformation parallel ablaufender Programme. Technische Universität München, Dissertation an der Fakultät für Mathematik, Februar 1980

/Broy 80b/
M. Broy: Transformational Semantics for Concurrent Programs. IPL 11:2, October 1980, 87-91

/Broy 81a/
M. Broy: Are Fairness-assumptions Fair? In: Proc. of the Second International Conference on Districuted Computing Systems, Paris April 8-10, 1981, IEEE 1981, 116-125

/Broy 81b/
Broy, M.: Prospects of New Tools for Software Development. In: Duijvestijn, A.J.W, Lockemann, C.P. (eds.): Trends in Information Processing Systems. Lecture Notes in Computer Science 123. Berlin - Heidelberg - New York: Springer 1981, 106-121

/Broy 81c/
M. Broy: On Language Constructs for Concurrent Programming. In: W. Händler (ed.): CONPAR 81. Lecture Notes in Computer Science 111, Berlin-Heidelberg-New York: Springer 1981, 141-154

/Broy 82/
M. Broy: Fixedpoint Theory of Communicating and Concurrent Processes. IFIP TC 2 Working Conference on "Formal Description of Programming Concepts II", Garmisch-Patenkirchen, June 1982

/Broy, Bauer 81/
M. Broy, F.L. Bauer: A Rational Approach to Language Constructs for Concurrent Programs. Lecture at the CREST Course on Design of Numerical Algorithms for Parallel Computing, Bergamo, June 1981

/Broy, Wirsing 80/
M. Broy, M. Wirsing: Initial versus Terminal Algebra Semantics for Partially Defined Abstract Types. Technische Universität München, Institut für Informatik, TUM-I8018, December 1980

/Broy, Wirsing 81a/
M. Broy, M. Wirsing: On the Algebraic Specification of Nondeterministic Programming Languages. In: E. Astesiano, C. Böhm (eds.): 6th Colloquium on Trees in Algebra and Programming, Genua 1981, Lecture Notes in Computer Science 112, Berlin - Heidelberg - New York: Springer 1981, 162-179

/Broy, Wirsing 81b/
M. Broy, M. Wirsing: Unbounded Nondeterminism - An exercise in Abstract Data Types. INRIA 1981

/Broy et al. 78/
M. Broy, R. Gnatz, M. Wirsing: Semantics of Nondeterministic and Noncontinous Constructs. In: /Bauer, Broy 79/, 553-592

/Broy et al. 80/
M. Broy, H. Partsch, P. Pepper, M. Wirsing: Semantic Relations in Programming Languages. In: S.H. Lavington (ed.): Information Processing 80, Proceedings of the IFIP Congress 80, Amsterdam - New York - Oxford: North-Holland Publ. Comp. 1980, 101-106

/Burge 75/
W.H. Burge: Stream Processing Functions. IBM Journal of Research and Development 19, January 1975, 12-25

/Chandra 74/
A.K. Chandra: The Power of Parallelism and Nondeterminism in Programming. In: J.L. Rosenfeld (ed.): Information Processing 74, Proc. of the IFIP Congress 74, Amsterdam: North Holland Publ. 1974, 461-465

/Chandra 78/
A.K. Chandra: Computable Nondeterministic Functions. Proc. of the 19th Annual Symposium on Foundations of Computer Science, October 1978, 127-131

/Chandra et al. 81/
A.K. Chandra, D.C. Kozen, L.J. Stockmeyer: Alternation. J. ACM 28:1, January 1981, 114-133

/Dennis 74/
J.B. Dennis: First Version of a Data Flow Procedure Language. In: B. Robinet (ed.): Colloque sur la Programmation, Lecture Notes in Computer Science 19, Berlin - Heidelberg - New York: Springer 1974, 362-367

/Dennis, Weng 79/
J.B. Dennis, K. K.-S. Weng: An Abstract Implementation for Concurrent Computation with Streams. In: Proc. of the 1979 International Conference on Parallel Processing, August 1979, 35-45

/Dershowitz, Manna 79/
N. Dershowitz, Z. Manna: Proving termination with Multiset Orderings. Comm. ACM 22:8, August 1979, 465-476

/Dijkstra 76/
E.W. Dijkstra: A Discipline of Programming. Prentice Hall, Englewood Cliffs N.J. 1976

/Egli 75/
H. Egli: A Mathematical Model for Nondeterministic Computations. Unpublished report ETH Zürich 1975

/Ershov 78/
A.P. Ershov: On the Essence of Compilation. In: /Neuhold 78/, 391-418

/Francez, Rodeh 80/
N. Francez, M. Rodeh: A Distributed Abstract Data Type Implemented by a Probabilistic Communication Scheme. 21st Annual Symposium on Foundations of Computer Science, October 1980

/Friedmann, Wise 76/
D.P. Friedmann, D.S. Wise: CONS Should not Evaluate its Arguments. In: S. Michaelson, R. Milner (eds.): Proc. of the 3rd International Colloquium on Automata, Languages and Programming. Edinburgh: Edinburgh Univ. Press, 1976, 257-284

/Friedmann, Wise 78a/
D.P. Friedmann, D.S. Wise: Applicative Multiprogramming. Indiana University, Computer Science Department, Technical Report 72, Januar 1978, revised December 1978

/Friedmann, Wise 78b/
D.P. Friedmann, D.S. Wise: A Note on Conditional Expressions. Comm. ACM 21:11, November 1978, 931-933

/Henderson 80/
P. Henderson: Functional Programming: Application and Implementation. Englewood Cliffs, NJ: Prentice Hall International 1980

/Henderson, Morris 76/
P. Henderson, J.H. Morris: A Lazy Evaluator. University of Newcastle upon Tyne, Computing Laboratory, Techn. Report Series No. 85

/Hennessy 80/
M.C.B. Hennessy: The Semantics of Call-by-Value and Call-by-Name in a Nondeterministic Environment. SIAM J. Comput. 9:1, February 1980, 67-84

/Hennessy, Ashcroft 76/
M. Hennessy, E.A. Ashcroft: The Semantics of Nondeterminism. In: S. Michaelson, R. Milner (eds.): Proc. of the 3rd International Colloquium on Automata, Languages and Programming, Edinburgh: Edinburgh University Press 1976, 479-493

/Hennessy, Ashcroft 77/
M. Hennessy, E.A. Ashcroft: Parameter Passing mechanisms and Nondeterminism. In: Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 1977, 306-311

/Hennessy, Ashcroft 80/
M. Hennessy, E.A. Ashcroft: A Mathematical Semantics for Typed -calculus. Theoretical Computer Science 10, 1980, 225-248

/Hewitt, Baker 78/
C. Hewitt, H. Baker: Actors and Continuous Functionals. In: /Neuhold 78/, 367-390

/Hoare 78/
C.A.R. Hoare: Communicating Sequential Processes. Comm. ACM 21:8, August 1978, 666-677

/Japan 81/
Preliminary report on study and research on fifth-generation-computers. Japan, October 1981

/Kahn 74/
G. Kahn: The Semantics of a Simple Language for Parallel Processing. In: J.L. Rosenfeld (ed.): Information Processing 74, Proc. of the IFIP Congress 74, Amsterdam: North-Holland 1974, 471-475

/Kahn, MacQueen 77/
G. Kahn, D. MacQueen: Coroutines and Networks of Parallel Processes. In: B. Gilchrist (ed.): Information Processing 77, Proc. of the IFIP Congress 77, Amsterdam: North-Holland 1977, 994-998

/Keller 78/
R.M. Keller: Denotational Models for Parallel Programs with Indeterminate Operators. In: /Neuhold 78/, 337-366

/Keller 80/
R.M. Keller: Semantics and Applications of Function Graphs. University of Utah, Department of Computer Science, Technical Report UUCS-80-112, October 1980

/Kennaway, Hoare 80/
J.R. Kennaway, C.A.R. Hoare: A Theory of Nondeterminism. In: J. de Bakker, J.v.d. Leuwen (eds.): Proc. of the 7th International Colloquium on Algorithms, Languages and Programming. Lecture Notes in Computer Science 85, Berlin - Heidelberg - New York: Springer 1980, 338-350

/Kleene 52/
S.C. Kleene: Introduction to Metamathematics. New York: Van Nostrand 1952

/König 50/
D. König: Theorie der endlicher und unendlicher Graphen. New York: Chelsea Publishing Company 1950

/Kosinski 73/
P.R. Kosinski: A Data Flow Language for Operating Systems Programming. SIGPLAN Notices 8:9, September 1973, 89-94

/Kosinski 77/
P.R. Kosinski: A Straightforward Denotational Semantics for Nondeterminate Data Flow Programs. Proc. of the 5th Annual Symposium on Principles of Programming Languages, 1977

/Kosinski 79/
P.R. Kosinski: Denotational Semantics of Determinate and Nondeterminate Data Flow Programs. MIT, Laboratory for Computer Science, TR-220, May 1979

/Landin 65/
P.J. Landin: A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. Comm. ACM 8:2, February 1965, 89-101

/Lehmann 76/
D.J. Lehmann: Categories for Fixpoint-Semantics. Proc. of the 17th Annual Symposium on Foundations of Computer Science 1976, 122-126

/MacQueen 79/
D.B. MacQueen: Models for Distributed Computing. IRIA Rapport de Recherche No 351, April 1979

/Mago 79/
G.A. Mago: A Network of r Microprocessors to Execute Reduction Languages. Int. Journ. Compt. and Inf. Sciences 8:5 / 8:6, 1979, 349-385 / 435-471

/Manna 70/
Z. Manna: The Correctness of Nondeterministic Programs. Artificial Intelligence 1, 1970, 1-26

/Manna et al. 73/
Z. Manna, S. Ness, J. Vuillemin: Inductive Methods for Proving Properties of Programs. Comm. ACM 16:8, August 1973, 491-502

/McCarthy 63/
J. McCarthy: A Basis for a Mathematical Theory of Computation. In: P. Braffort, D. Hirschberg (eds.): Computer Programming and Formal Systems, Amsterdam: North-Holland 1963

/Milne, Milner 77/
G. Milne, R. Milner: Concurrent Processes and their Syntax. University of Edinburgh, Department of Computer Science, CSR-2-77, May 77

/Milner 73/
R. Milner: Processes: A Mathematical Model of Computing Agents. Proc. Logic Colloquium, Bristol, Amsterdam: North-Holland 1973, 157-173

/Milner 80a/
R. Milner: A Calculus of Communicating Systems. Lecture Notes in Computer Science 92, Berlin - Heidelberg - New York: Springer 1980

/Milner 80b/
R. Milner: On relating Synchrony and Asynchrony, University of Edinburgh, Department of Computer Science, Internal Report CSR-75-80, December 1980

/Neuhold 78/
E.J. Neuhold (ed.): Formal Descriptions of Programming Concepts. Amsterdam: North-Holland 1978

/Nivat 80/
M. Nivat: Nondeterministic Programs: An Algebraic Overview. In: S.H. Lavington (ed.): Information Processing 80, Proc. of the IFIP Congress 80, Amsterdam - New York - Oxford: North-Holland Publ. Comp. 1980, 17-28

/Päppinghaus, Wirsing 81/
P. Päppinghaus, M. Wirsing: Nondeterministic Partial Logic: Isotonoic and Guarded Truth-Functions. University of Edinburgh, Dept. of Computer Science, Report CSR-83-81, June 1981

/Park 80/
D. Park: On the Semantics of Fair Parallelism. In: D. Björner (ed.): Abstract Software Specification. Lecture Notes in Computer Science 86, Berlin - Heidelberg - New York: Springer 1980, 504-526

/Plotkin 76/
G. Plotkin: A Powerdomain Construction. SIAM J. on Computing 5, 1976, 452-486

/Plotkin 77/
G. Plotkin: LCF considered as a programming language. Theoretical Computer Science 5:3, December 1977, 223-256

/Scott 80a/
D. Scott: Lambda Calculus: Some Models, Some Philosophy. In: J. Barwise, H.J. Keisler, K. Kunen (eds.): The Kleene Symposium. North-Holland Publ. Comp. 1980, 223-265

/Scott 80b/
D. Scott: Lectures on a Mathematical Theory of Computation. University of Oxford, Mathematical Institute, Preliminary Version, completed November 1980

/Smyth 78/
M.B. Smyth: Power Domains. J. CSS 16, 1978, 23-36

/Tesler, Enea 68/
L.G. Tesler, H.J. Enea: A Language Design for Concurrent Processes. Spring Joint Computer Conference 1968, 403-408

/Treleaven et al. 81/
P.C. Treleaven, D.R. Brownbridge, R.P. Hopkins: Data Driven and Demand Driven Computer Architecture. University of Newcastle upon Tyne, Technical Report Series 168, June 1981

/Turner 79/
D.A. Turner: A New Implementation Technique for Applicative Languages. Software-Practice and Experience 9, 1979, 31-49

/Vuillemin 74/
J. Vuillemin: Correct and Optimal Implementation of Recursion in a Simple Programming Language. J. Comp. Sci. 9:3, June 1974, 332-354

/Vuillemin 75/
J. Vuillemin: Syntaxe, Semantique et Axiomatique d'un Langage de Programmation Simple. Interdisciplinary Systems Research, Vol. 12, Basel - Stuttgart: Birkhäuser 1975

/Wadsworth 71/
C. Wadsworth: Semantics and Pragmatics of Lambda Calculus. Oxford, Ph. D. Dissertation 1971