PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 43/96

On Algorithmic Fork Relations: Effectiveness and Program Constructs

Paulo A. S. Veloso

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, N° 43/96 ISSN 0103-9741
Editor: Carlos J. P. Lucena December 1996

On Algorithmic Fork Relations: Effectiveness and Program Constructs *

Paulo A. S. VELOSO †

* Research partly sponsored by CNPq.

† On leave at Instituto de Matemática, Universidade Federal do Rio de Janeiro.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

ON ALGORITHMIC FORK RELATIONS: EFFECTIVENESS AND PROGRAM CONSTRUCTS

Paulo A. S. VELOSO

{e-mail: veloso@inf.puc-rio.br}

PUCRioInf MCC 43/96

Abstract. We examine the adequacy of an extended relational framework for program development. We argue that we can select, by effectiveness considerations, a certain set of symbols deserving the name "algorithmic" and we outline a programming language correspondence, which indicates that this repertoire of symbols provides adequate power for expressing programs. In previous reports, we introduced the question of adequacy of an extended relational framework for program development: we argued that the familiar apparatus of binary relations should be extended to be appropriate for programming and we examined the nature of such an extension obtained by considering relations on structured universes with new structural operations. The adequacy of such an extended apparatus hinges on having an appropriate expressive power. In this report we address the issue of adequate expressive power for programming. We offer two explanations for the selection of the algorithmic part and its computing-like nature, which jointly justify its adequacy. First, the identification of the proper repertoire is based on effectiveness, which guarantees soundness, in the sense of not leaving the effective realm. The second - more intuitive - explanation relies on the programming language correspondence, which, besides reinforcing the first explication, indicates that these algorithmic symbols provide adequate power for expressing programs.

Key words: Formal specifications, relational calculi, fork algebras, structural operations, effectiveness, programming constructs, expressive power.

Resumo. Examina-se a adequação de um ambiente relacional estendido por fork para desenvolvimento de programas. Argumenta-se que se pode selecionar, por considerações de efetividade, um certo conjunto de símbolos merecendo o nome "algorítmico", e esboça-se uma correspondência com linguagens de programação, a qual indica que este repertório de símbolos fornece poder adequado para expressar programas. Em trabalhos anteriores introduziu-se a questão da adequação de um ambiente relacional estendido por fork para desenvolvimento de programas: argumentou-se que o aparato usual de relações binárias deveria ser estendido para ser adequado para programação e examinou-se a natureza de tal extensão obtida considerando-se relações sobre um universo estruturado com novas operações estruturais. A adequação de tal aparato estendido depende de ter poder expressivo apropriado. Neste trabalho enfoca-se a questão poder expressivo adequado para programação. Oferecem-se duas explicações para a seleção da parte algorítmica e de sua natureza computacional, que conjuntamente justificam sua adequação. Primeiramente, a identificação do repertório apropriado é baseada em efetividade, o que garante a corretude, no sentido de não se sair do domínio do efetivo. A segunda explicação - mais intuitiva - repousa na correspondência com linguagens de programação, a qual, além de reforçar a anterior, indica que esses símbolos algorítmicos fornecem poder adequado para expressar programas.

Palavras chave: Especificações formais, cálculos relacionais, álgebras de fork, operações estruturais, efetividade, conceitos de programação, poder expressivo.

NOTE

Research reported herein is part of on-going project.

Collaboration with Armando M. Haeberer, Marcelo F. Frias and Gabriel Baum was instrumental in sharpening many ideas. The author would also like to thank the following for many fruitful discussions on these and related topics: Gunther Schmidt, Rudolf Berghammer, Pablo Elustondo and Juán Durán.

A preliminary version of some of these ideas was presented as part of an invited talk at PRATICA '96 (PRovAs, TIpos e CAtegorias), a workshop organised by Luís Carlos Pereira that took place at PUC-Rio, Rio de Janeiro, Brazil, in April 1996.

This report is the third one of a series of reports addressing the question of adequacy of a fork relational framework for program development. Other reports focus on other, related aspects of this question. Other reports concentrate on distinct aspects of this question, such as:

A fork relational framework for program development;

Structured universe and structural operations and constants;

Algorithmic fork relations and programs.

CONTENTS

1. INTRODUCTION

In this report we examine the adequacy of an extended relational framework for program development. We argue that we can select, by effectiveness considerations, a certain set of symbols deserving the name "algorithmic". We also outline a programming language correspondence, which indicates that these algorithmic symbols provide adequate power for expressing programs.

In two previous reports, we introduced the question of adequacy of an extended relational framework for program development. We argued that the familiar apparatus of binary relations must be extended to be appropriate for programming [Veloso '96a]. We also examined the nature of such an extension obtained by considering relations on structured universes with new structural operations [Veloso '96b].

A basic issue concerning such a programming framework concerns its adequacy for expressing programs. This adequacy relies on having an appropriate expressive power.

In this report we examine more closely the algorithmic part of this fork relational framework and identify a set of symbols that deserve being called "algorithmic". We offer two explanations for this selection, which jointly justify its adequacy. First, the identification of the proper repertoire will be based on effectiveness, which will guarantee soundness, in the sense that one does not leave the effective realm. The second explanation relies on a programming language correspondence, which, besides reinforcing the first explication, will indicate that we have adequate expressive power.

We begin by reviewing some of the material in the two previous reports. In section 2 we recall some basic operations on relations and present a series of examples, intended to illustrate how one can express programming ideas in a relational form. In section 3 we examine the proposed extension and the new structural operations as well as some of their properties.

In section 4 we examine some considerations connecting algorithms and program constructs to effectiveness, with the purpose of preparing the terrain for a classification of the extended relational symbols. This classification is presented in section 5 as a basis for the selection of the algorithmic symbols, which is explained in terms of effective properties and their preservation. We then move on to the programming language correspondence, which is presented in section 6, indicate that these algorithmic symbols provide adequate power for expressing programs. Some other aspects, of importance in the context of a framework for program development, are briefly commented upon in section 7. Finally, section 8 presents some concluding remarks and comments on related aspects.

This report is the third one of a series of reports addressing the question of adequacy of a fork relational framework for program development.

Other reports focus on other, related aspects of this question.

2. BASIC IDEAS: SETS, RELATIONS AND PROGRAMMING

We begin with some preparatory material [Veloso '96a, b]. We first recall some operations on relations. Next, we briefly illustrate how one can express programs and programming ideas within an extended relational framework, which is more adequate for programming tasks.

2.1 SETS AND RELATIONS

We now briefly recall some operations on relations and examine partial identities as tools for representing sets as relations.

A. Operations on Sets and Relations

We first recall some operations on sets and relations [Halmos '63; Tarski '41; Maddux '91; Schmidt & Ströhlein '93; Veloso '74].

Consider a set S and a (universal) subset $V \subseteq S$. We then have some operations on subsets of V as well as some distinguished subsets of V.

We have some set-theoretical, or Boolean, operations and constants.

As *Boolean operations* we will employ:

set-theoretical union $\cup$, intersection $\cap$, and

complementation (with respect to universal V: $r^{\sim} := \{s \in V / s \notin r\}$).

We also have the *Boolean constants*:

empty relation $\varnothing$ and universal subset V.

When dealing with sets of ordered pairs, i. e. relations on U, one has some more operations and constants, often called Peircean [Maddux '91].

We will use two *Peircean operations*:

relation transposition (converse) $r^{T} := \{\langle v,u\rangle \in U \times U / \langle u,v\rangle \in r\}$ and

relation composition (relative product)

$r|s := \{\langle u,w\rangle \in U \times U / \exists v \in U\ [\langle u,v\rangle \in r \,\&\, \langle v,w\rangle \in s]\}$,

We will also employ the *Peircean constant*:

identity (diagonal) relation on U $1_U := \{\langle u,v\rangle \in U \times U / u=v\}$.

Notice that these operations (except Boolean complementation $^{\sim}$) are monotonic with respect to inclusion $\subseteq$ [Schmidt & Ströhlein '93].

In the case of relations on U, the universal subset will also be a relation $V \subseteq U \times U$. One then wishes to have closure: in particular, one wishes to have 1_U, V^T and $V|V$ included in V. This means that the universal relation $V \subseteq U \times U$ must be an equivalence relation on U. For such an equivalence relation $V \subseteq U \times U$ on U, its powerset $\wp(V) = \{r \subseteq U \times U / r \subseteq V\}$ will be closed under both operations transposition ^{T} and composition $|$ (in view of their monotonicity). (See, e. g. [Jónsson & Tarski '52; Veloso '74].)

B. Partial Identities

Let us now examine partial identities [Veloso '96a; Haeberer & Veloso '91].

Partial identities are devices for representing sets as relations, they are useful tools in expressing programs and specifications in a relational manner. A partial identity behaves as a "filter" on its underlying set; as

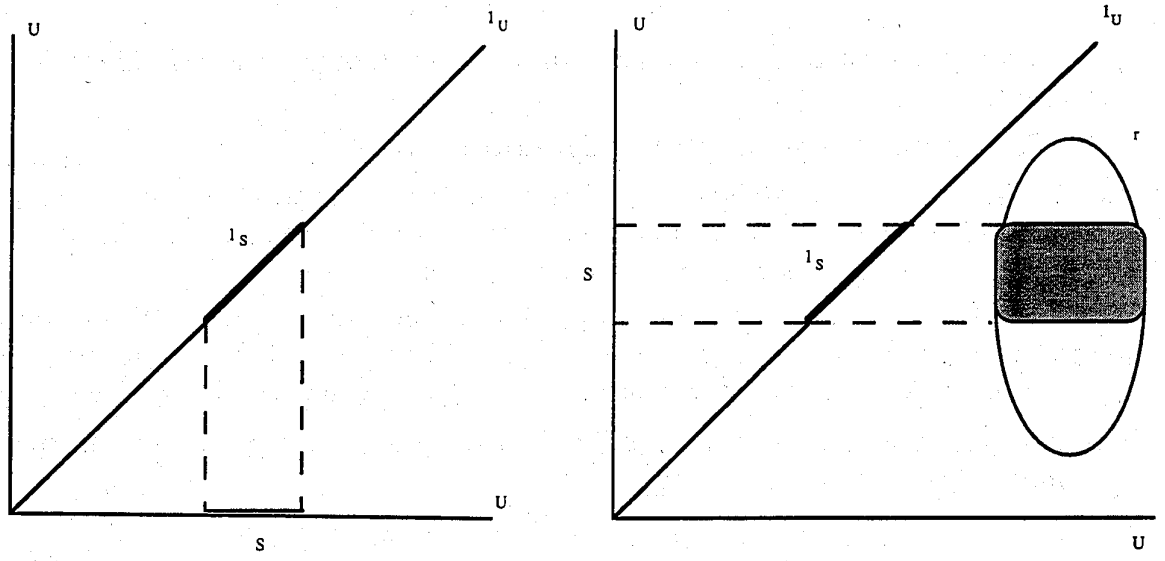
such it can also be used to obtain tests for membership as well as for restricting relations to sets .

Given a subset $S \subseteq U$, by the *partial identity* on S we mean the binary relation $1_S := \{<u,u> \in U \times U / u \in S\}$ on U. From the partial identity 1_S one can recover the set S it represents, since $S = \{u \in U / <u,u> \in 1_S\}$.

Partial identity 1_S behaves as a "filter" on S: for $u \in S$ it behaves identically ($<u,v> \in 1_S$ iff v=u), whereas for $u \notin S$ it provides no output.

As a test for non-membership we can use the complement with respect to the identity: $\hat{}1_S := 1_U \cap 1_S\tilde{}$, for $\hat{}1_S = \{<u,u> \in U \times U / u \notin S\}$.

These partial identities can be used to express, in relational terms, restrictions of a relation $r \subseteq U \times U$ to a set $S \subseteq U$ by composition: pre composition $1_S | r = \{<u,v> \in r / u \in S\}$, and post composition $r | 1_S = \{<u,v> \in r / v \in S\}$.



2.2 Expressing Programming Ideas with (Extended) Relations

We now briefly indicate how one can express programs and programming ideas in relational terms. We illustrate the expression of programming ideas by means of (extended) relations with five simple examples: a program, an input-output specification, a simple unsorted data type specification, a programming method and a many-sorted data type presentation [Veloso '96a, b].

A. Expressing Programs as Relations

As example of a program, consider a program for computing the double of a natural number by relying on successor, predecessor and zero.

A (recursive) formulation in the usual manner can be as follows:

$$D(n) = \begin{cases} 0 & \text{if } n = 0 \\ SSDP(n) & \text{otherwise} \end{cases}$$

The test for zero can be expressed by the partial identity $1_{zr} = \{<0,0>\}$ and the case division under test for zero can be expressed by means of 1_{zr} and its complement with respect to 1_N: $1_{nzr} = 1_{zr}\tilde{} \cap 1_N$.

We thus have the following formulation of binary relation $D \subseteq \mathbf{N} \times \mathbf{N}$ in terms of its input-output pairs:

$$n \xrightarrow{D} m \;\Leftrightarrow\; \left\{ \begin{array}{c} n \xrightarrow{1_{zr}} m \\ \text{or} \\ n \xrightarrow[P]{} i \xrightarrow[D]{} j \xrightarrow[S\,|S]{} m \end{array} \right\}$$

We can thus express binary relation $D \subseteq \mathbf{N} \times \mathbf{N}$ by means of a relational term: $D = 1_{zr} \cup (1_{nzr} | P | D | S | S)$.

$$D = \left(\begin{array}{c} 1_{zr} \\ \cup \\ (1_{zr}^{\sim} \cap 1_{\mathbf{N}}) \,|P\,|D\,|S\,|S \end{array} \right)$$

This example illustrates how one can express (some simple) programs in relational terms.

B. Relational input-output Specifications

As a simple example of an input-output specification, consider palindromes, namely words that read the same forwards or backwards. So, a palindrome is a word that is equal to its reversal.

Here, we use a binary relation $Rev \subseteq W \times W$ on the universe W of words, where $\langle u,v \rangle \in Rev$ iff v is the reversal of u. The set Pal of palindromes can be described by a "filter" for palindromes $1_{Pal} = \{\langle w,w \rangle \in W \times W / w \in Pal\}$.

We can describe the behaviour of a "filter" for palindromes by means of its input-output pairs as follows:

$$u \xrightarrow{1_{Pal}} v \;\Leftrightarrow\; \left[\begin{array}{c} u \xrightarrow{Rev} v \\ \text{and} \\ u \xrightarrow[1_W]{} v \end{array} \right] \qquad 1_{Pal} = \left(\begin{array}{c} Rev \\ \cap \\ 1_W \end{array} \right)$$

We can thus express binary relation $1_{Pal} \subseteq W \times W$ as $1_{Pal} = Rev \cap 1_W$.

Now, assume that we have constant relations Tr and Fl, matching every word to, respectively, true and false. We can then obtain, from the above expression for 1_{Pal}, a relational term for the characteristic function of the set Pal of palindromes. Namely, $\chi_{Pal} = (1_{Pal} | Tr) \cup [(1_{Pal}^{\sim} \cap 1_W) | Fl]$.

$$\chi_{Pal} = \left(\begin{array}{c} 1_{Pal} \,|Tr \\ \cup \\ (1_{Pal}^{\sim} \cap 1_W) \,|Fl \end{array} \right)$$

This example illustrates how one can express input-output specifications by means of relational terms. (From such relational input-output specifications one can derive, by algebraic manipulations, some (recursive and iterative) programs for palindromes [Haeberer & Veloso '91].)

C. Relational Data Type Presentations

For a simple example, consider the data type of the natural numbers with operations successor s and predecessor p as well as constant zero.

To express properties of this data type in a relational manner, we proceed in a manner similar to the one implicit in the previous example. Namely, for the operations we use their graphs: relations S and P. For the constant 0 we use the constant relation $Z:=\mathbf{N}\times\{0\}$. From this constant relation Z we can obtain the partial identity $1_{zr}=\{<0,0>\}$. So, we also have the partial identity $1_{nzr}=1_{zr}^{\sim}\cap 1_{\mathbf{N}}^{\sim}$ for the non-zero naturals.

With these ideas, we can express properties of this data type. For instance, consider some axioms and their expressions in relational terms.

Injectivity of s: $S|S^T\subseteq 1_{\mathbf{N}}$,

$\forall x\neg s(s(x)=x$: $S|S\cap 1_{\mathbf{N}}=\emptyset$,

$\forall y[y=0\vee\exists xy=s(x)]$: $1_{nzr}\subseteq S^T|S$ or $1_{\mathbf{N}}\subseteq 1_{zr}\cup(S^T|S)$

$$y \xrightarrow{S^T} x \xrightarrow{S} y \;:\; 1_{nzr}\subseteq S^T|S,$$

$\forall y[\neg y=0\rightarrow s(p(y))=y]$: $1_{nzr}|P|S\subseteq 1_{\mathbf{N}}$

$$y \xrightarrow{1_{nzr}} y \xrightarrow{P} x \xrightarrow{S} y \;:\; 1_{nzr}|P|S\subseteq 1_{\mathbf{N}}.$$

This example illustrates how one can present unsorted data types by means of relational specifications.

D. Expressing Programming Methods

As example of a programming method, we consider divide-and-conquer.

Mergesort is a sorting algorithm based on this method [Aho et al. '75]. This algorithm is based on the following idea. Given an input sequence:

if it is simple ($lg(s)\leq 1$), then its sorted version is directly obtained;

otherwise, split s into its two halves, recursively apply the algorithm and recombine (by merging) the results.

This process of recursively splitting, until reaching simple instances, and recombinations is the idea underlying the problem-solving method divide-and-conquer [Horowitz & Sahni '78]. It can be presented as follows:

$$D\_C(d) \;=\; \begin{cases} Drct(d) & \text{if } Smpl(d) \\ Rcmbn\left(\begin{matrix} D\_C(Splt_1(d)) \\ D\_C(Splt_2(d)) \end{matrix}\right) & \text{otherwise} \end{cases}$$

Its behaviour in terms of input-output pairs can be described as follows:

$$\begin{cases} \text{if } Smpl(d)\text{: } d \xrightarrow{D\_C} r \;\Leftrightarrow\; d \xrightarrow{Drct} r \\ \text{otherwise: } d \xrightarrow[D\_C]{} r \;\Leftrightarrow\; d \xrightarrow{Splt} \begin{pmatrix} d_1 \xrightarrow{D\_C} r_1 \\ d_2 \xrightarrow{D\_C} r_2 \end{pmatrix} \xrightarrow{Rcmbn} r \end{cases}$$

We can notice two related features of this description:

process Splt produces from an input d a pair $<d_1,d_2>$ of outputs;

D_C is applied in parallel to each component of $<d_1,d_2>$ to produce $<r_1,r_2>$.

If we use $\|$ for the parallel execution on components, we can represent this situation as:

$$d \xrightarrow{\text{Splt}} \begin{pmatrix} d_1 \\ , \\ d_2 \end{pmatrix} \qquad \begin{pmatrix} d_1 \\ , \\ d_2 \end{pmatrix} \begin{array}{l} \xrightarrow{D\_C} \\ \\ \xrightarrow{D\_C} \end{array} \begin{pmatrix} r_1 \\ , \\ r_2 \end{pmatrix} \Leftrightarrow \begin{pmatrix} d_1 \\ , \\ d_2 \end{pmatrix} \begin{pmatrix} D\_C \\ \| \\ D\_C \end{pmatrix} \begin{pmatrix} r_1 \\ , \\ r_2 \end{pmatrix}$$

With the parallel product $/\!/$ operation, we can write an expression for binary relation D_C:

$$D\_C = \left(\begin{array}{c} 1_{Smpl} \,|Drct \\ \cup \\ (1_{Smpl}{}^{\sim} \cap 1_U) \,|Splt| \begin{pmatrix} D\_C \\ /\!/ \\ D\_C \end{pmatrix} |Rcmbn \end{array} \right)$$

This example illustrates how one can express programming methods by means of extended relational terms. Of course, for expressing some programs, such as mergesort and related sorting algorithms [Darlington '78; Broy '83], such extension will also be used.

We mention that (some) program derivation strategies can be formulated in a similar manner [Haeberer & Veloso '91]. Their application relies on a pattern matching procedure, similar to the one involved in applying an algebraic law, say the binomial expansion, to a particular instance.

E. Relational Presentations for many-sorted Structures

Data types used in programming often involve several sorts.

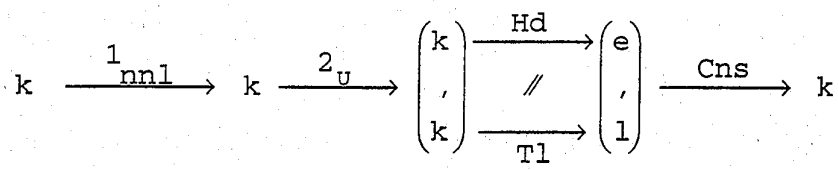
For a simple example of a many-sorted structure, consider the data type Lists of Elements. It has sorts Lst and Elm, operations head, tail and cons, as well as unary predicate null, say. To express properties of this many-sorted data type in a relational manner, we proceed as follows.

The universe U is to consist of the sorts Lst and Elm as well as of their ordered pairs. For the sorts we use partial identities 1_{Lst} and 1_{Elm}. For the operations we use their graphs: relations Hd, Tl and Cns. For the predicate null we use the partial identity $1_{nl}=\{<k,k>/k\in \text{null}\}$ (so, we have partial identity $1_{nnl}:=1_{Lst}\cap 1_{nl}{}^{\sim}$ to represent the non-null lists).

With these ideas, we can express a (reachability) axiom such as $(\forall k{:}Lst)[null(k)\vee(\exists e{:}Elm)(\exists l{:}Lst)k=cons(e,l)]$ relationally by one of the alternative formulations: $1_{nnl}\subseteq(Cns^T)|Cns$ or $1_{Lst}\subseteq 1_{nl}\cup[(Cns^T)|Cns]$.

Also, a property like $(\forall k{:}Lst)[\neg null(k)\rightarrow cons(head(k),tail(k))=k]$ can be expressed relationally by $1_{nnl}|2_U|(Hd/\!/Tl)|Cns\subseteq 1_{Lst}$, where the doubling

relation $2_U := \{<u,<u,u>> \in U \times U / u \in U\}$ outputs two copies of the input.



This example illustrates how one can describe a many-sorted data type by a relational presentation.

3. STRUCTURED UNIVERSE AND EXTENDED RELATIONS

We have indicated that the relational framework should be extended in order to express programming ideas. We now take a closer look at this extended relational framework [Veloso '96b].

As mentioned, the universe should have ordered pairs of its elements. Thus, we consider the universe U to be closed under cartesian product: $U \times U \subseteq U$. As a consequence, we no longer have an unstructured set of elements. Instead, we will be dealing with a universe with underlying structure, having objects such as $<u,v>$, $<u,<v,w>>$, $<<u,v>,w>$, and so forth. We use the name *structured universe* for such a set U closed under cartesian product: $U \times U \subseteq U$.

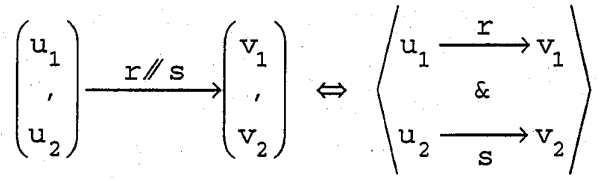
3.1 STRUCTURAL RELATIONS AND OPERATIONS

We shall be dealing with relations on a structured universe; so we can have some more operations and constants, which we will call structural. We shall now examine these structural operations and constants as well as some interconnections among them.

A. Extended Operations and Constants

Of the new structural operations and constants we have already encountered the parallel product // and the duplication 2_U.

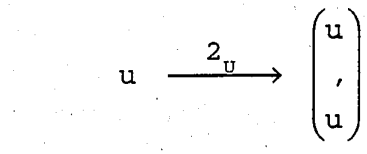
Operation *parallel product* // corresponds to parallel execution, as such $r//s := \{<<u_1,u_2>,<v_1,v_2>> \in U \times U / <u_1,v_1> \in r \,\&\, <u_2,v_2> \in s\}$.



Notice that parallel product operation // is monotonic with respect to inclusion $\subseteq$ and preserves functionality of relations.

Constant *duplication* 2_U produces two copies of the input.

$2_U := \{<u,<v,w>> \in U \times U / u=v=w\}$



Notice that constant relation 2_U is a total functional relation.

With this constant 2_U we can construct an *equality sieve* 2_U^T.

Equality sieve $2_U{}^T$ $\begin{pmatrix} u \\ , \\ v \end{pmatrix} \xrightarrow{2_U{}^T} w \quad \Leftrightarrow \quad u = v = w$

Another natural operation on relations comes from the idea of feeding a common input to two relations. This new operation, called *fork*, produces relation $r\angle s:=\{<u,<v,w>>\in U\times U/<u,v>\in r\&<u,w>\in s\}$.

Notice that fork operation $\angle$ is monotonic with respect to inclusion $\subseteq$ and preserves functionality of relations; also $Dom[r\angle s]=Dom[r]\cap Dom[s]$.

Fork $\angle$ can be defined from $/\!/$ and 2_U, since $r\angle s=2_U|(r/\!/s)$.

$$u \longrightarrow \angle\left\langle \begin{matrix} \xrightarrow{r} \\ \\ \xrightarrow[s]{} \end{matrix} \begin{pmatrix} v \\ , \\ w \end{pmatrix} \right. \quad \Leftrightarrow \quad u \xrightarrow{2_U} \begin{pmatrix} u \\ , \\ u \end{pmatrix} \begin{matrix} \xrightarrow{r} \\ /\!/ \\ \xrightarrow[s]{} \end{matrix} \begin{pmatrix} v \\ , \\ w \end{pmatrix}$$

B. Alternative Bases for the Extension

In general, the universal relation V is an equivalence relation V on U. We then wish to have $V/\!/V$ and $V\angle V$ included in V. We thus require V to be closed under pair formation: $<x,<x,y>>\in V$ whenever $<x,y>\in V$.

We use the names *closed equivalence* for an equivalence relation $V\subseteq U\times U$ on U that is closed under pair formation ($<x,<x,y>>\in V$ whenever $<x,y>\in V$), and *structural equivalence* for a structured universe U equipped with a closed equivalence relation $V\subseteq U\times U$ on U.

For such a closed equivalence V, its powerset $\wp(V)=\{r\subseteq U\times U/r\subseteq V\}$ is closed under both operations parallel product $/\!/$ and fork $\angle$ (in view of their monotonicity). We then have both $2_U, 2_U{}^T\subseteq V$.

Clearly, duplication constant 2_U can be derived from fork $\angle$, via $2_U=1_U\angle 1_U$.

With the operation fork $\angle$ we can construct new constant relations: the *left* and *right projections* $\pi_V:=(1_U\angle V)^T$ and $\rho_V:=(V\angle 1_U)^T$.

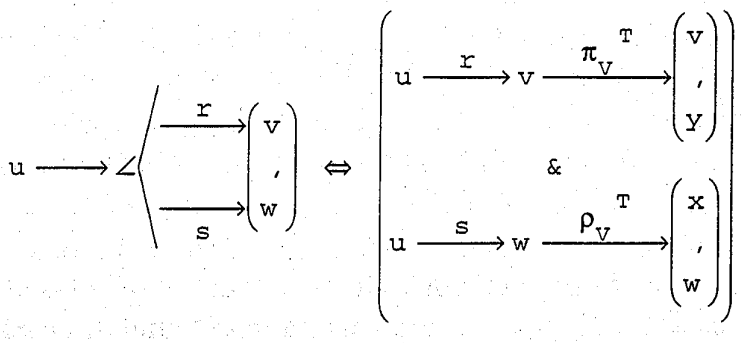
$$\pi_V{}^T : x \longrightarrow \angle\left\langle \begin{matrix} \xrightarrow{1_U} \\ \\ \xrightarrow[V]{} \end{matrix} \begin{pmatrix} x \\ , \\ y \end{pmatrix} \right. \quad \& \quad \rho_V{}^T : y \longrightarrow \angle\left\langle \begin{matrix} \xrightarrow{V} \\ \\ \xrightarrow[1_U]{} \end{matrix} \begin{pmatrix} x \\ , \\ y \end{pmatrix} \right.$$

We notice that the constant projections $\pi_V=\{<<x,y>,x>\in U\times U/<x,y>\in V\}$ and $\rho_V=\{<<x,y>,y>\in U\times U/<x,y>\in V\}$ are partial functions with domain $U\times U\subseteq U$.

We can also notice that from fork $\angle$ and the projections $\pi_V=(1_U\angle V)^T$ and $\rho_V=(V\angle 1_U)^T$ one can recover parallel product $/\!/$, via $r/\!/s=(\pi_V|r)\angle(\rho_V|s)$.

$$\begin{pmatrix} u_1 \\ , \\ u_2 \end{pmatrix} \begin{matrix} \xrightarrow{r} \\ /\!/ \\ \xrightarrow[s]{} \end{matrix} \begin{pmatrix} v_1 \\ , \\ v_2 \end{pmatrix} \quad \Leftrightarrow \quad \begin{pmatrix} u_1 \\ , \\ u_2 \end{pmatrix} \longrightarrow \angle\left\langle \begin{matrix} \xrightarrow{\pi_V} u_1 \xrightarrow{r} \\ \\ \xrightarrow[\rho_V]{} u_2 \xrightarrow[s]{} \end{matrix} \begin{pmatrix} v_1 \\ , \\ v_2 \end{pmatrix} \right.$$

Also, notice that from the transposed projections $\pi_V{}^T=1_U\angle V$ and $\rho_V{}^T=V\angle 1_U$ one can recover fork $\angle$, via $r\angle s=(r|\pi_V{}^T)\cap(s|\rho_V{}^T)$.



Hence, the sets $\{\angle\}$, $\{/\!/,2_U\}$ and $\{\pi_V,\rho_V\}$ are all relationally interderivable, and either one of them can be used as a basis for our extension.

Other interesting examples of interdefinability are provided by set-theoretical intersection and by Peircean transposition of relations.

Boolean operation $\cap$ can be recovered from fork $\angle$ and equality sieve $2_U{}^T$, since $r\cap s=(r\angle s)|2_U{}^T$.

Peircean operation ^{T} can be defined from intersection $\cap$, parallel product $/\!/$, projections and their transpositions, since $r^T=\{[\pi_V{}^T|(1_U/\!/r)]\cap\rho_V{}^T\}|\pi_V$.

Thus, on such a structural equivalence we can have, in addition to the Boolean and Peircean apparatus, some *structural* operations and constants:

structural operations parallel product $/\!/$ and fork $\angle$;

structural constants duplication 2_U, left and right projections π_V and ρ_V.

As mentioned, as bases for our extension we can use fork $\angle$ alone, the left and right projections π_V and ρ_V, or both $/\!/$ and 2_U.

3.2 Properties and Derived Operations

We will now consider some simple properties of this extended relational framework and some useful derived operations. We will first examine some relational properties of this extended apparatus. Then, we will consider some derived operations and take a closer look at the representation of subsets by partial identities.

A. Properties of the Extended Operations and Constants

We now notice some properties of this extended relational apparatus concerning totality and functionality as well as their preservation.

With respect to totality, we notice that many constants are total but most operations fail to preserve this property.

The constants (but the projections π_V and ρ_V) are total relations:
the constant relations identity $1_U=\{<u,u>\in U\times U/u\in U\}$, universal V and duplication $2_U=\{<u,<u,u>>\in U\times U/u\in U\}$ are total relations (included in V).

The operations union $\cup$ and fork $\angle$ preserve totality of relations:
the family of total relations (included in V) is closed under the operations but Boolean $\cap$ and $\tilde{}$, Peircean | and ^{T}, and structural $/\!/$.

As for functionality, we can see that most constants are functional and some operations preserve this property.

The constants (but universal V) are functional relations:
the constants empty relation $\varnothing$, identity $1_U=\{<u,u>\in U\times U/u\in U\}$, duplication $2_U=\{<u,<u,u>>\in U\times U/u\in U\}$ (and equality sieve $2_U{}^T=\{<<u,u>,u>\in U\times U/u\in U\}$), as well as the projections $\pi_V=\{<<x,y>,x>\in U\times U/<x,y>\in V\}$ and $\rho_V=\{<<x,y>,y>\in U\times U/<x,y>\in V\}$ are functional relations (included in V).

Operations $\cap$, $|$, $/\!\!/$ and $\angle$ preserve functionality of relations:
the family of functional relations (included in V) is closed under the operations except union $\cup$, complementation $\tilde{}$ and transposition ^{T}.

B. Derived Operations and Partial Identities

Of course, one can introduce other operations by definition. We examine a few examples of derived operations that are often found useful.

An example of derived operation is the operation restricted union. Given relations p and q on U, we define the binary operation *restricted union over* p and q ${}_p\oplus_q$ by $r\,{}_p\oplus_q s:=(p|r)\cup(q|s)$.

Complementation with respect to the identity is an important derived operation. Unary operation *identity complementation* $\hat{}$ is defined by $\hat{}t:=1_U\cap t\tilde{}$; so $\hat{}t=\{<u,u>\in U\times U/<u,u>\notin t\}$.

Another important example of derived operation is the domain representation. Unary operation *domain* $\_$ is defined by $\underline{r}:=(r|r^T)\cap 1_U$. Notice that $\underline{r}\subseteq 1_U$ and $\underline{r}=\{<u,u>\in U\times U/u\in Dom[r]\}$.

We now comment on the usage of partial identities for representing sets.

First, notice that the family $\wp(1_U):=\{p\subseteq U\times U/p\subseteq 1_U\}$ of *partial identities* consists of functional relations. We now explicitly introduce a transformation for representing subsets of U as partial identities, namely $1\_:\wp(U)\rightarrow\wp(1_U)$ given by the assignment $S\mapsto 1_S:=\{<u,u>\in U\times U/u\in S\}$.

The family $\wp(1_U)$ of partial identities forms a Boolean algebra (with identity complementation $\hat{}p=1_U\cap p\tilde{}$) that is isomorphic to the field of subsets $\wp(U)$ (under the assignments $S\mapsto 1_S$ and $p\mapsto Dom[p]$).

Also, the family $\wp(1_U)$ of partial identities is closed under transposition ^{T}, composition $|$ and parallel product $/\!\!/$ (as well as domain $\_$.)

{Because $1_S{}^T=1_S$, $1_S|1_T=1_{S\cap T}=1_S\cap 1_T$ and $1_S/\!\!/1_T=1_{S\times T}$.}

By resorting to partial identities, one can obtain interesting versions of restricted union, namely guarded and branching unions.

The *guarded union* is simply the union restricted to partial identities: for partial identities $p,q\in\wp(1_U)$ $r\,{}_p[]_q s:=r\,{}_p\oplus_q s$.

Given a partial identity $t\in\wp(1_U)$, the binary operation *branching union over* t $\vee_t$ is defined by $r\vee_t s:=r\,{}_t\oplus_{\hat{}t}$; so $r\vee_t s=(t|r)\cup(\hat{}t|s)$.

Notice that one can recover Boolean union from guarded union as a restricted union over the diagonal 1_U. Also, branching union preserves

functionality: if r and s are functional then so is $r \vee_t s$ for each $t \subseteq 1_U$.

4. PROGRAMMING AND EFFECTIVENESS

We have already indicated the desirability of suitably extending the relational approach for expressing programming ideas [Veloso '96a]. Also, the preceding section indicates that such extended relational framework provides new operations and constants [Veloso '96b].

We wish to see that we have an adequate framework for programming. We shall argue that a certain set of extended relational operations and constants can be regarded as forming a programming language.

It turns out that we can single out a certain set of terms, built from some of the operations and constants, deserving the name "algorithmic". We shall offer two justifications for the selection of such a subset of operations and constants (and explanations for their computing-like character): one in terms of effectiveness (preservation of effective enumerability) and a programming view (programming language correspondence).

For this purpose, we first examine some considerations connecting algorithms and program constructs to effectiveness. These considerations are based on effective enumerability and the preservation of effective enumerability. They hinge on the idea that programs on concrete data types have effective behaviours, which is to be preserved by program constructs.

4.1 CONCRETE DATA TYPES AND EFFECTIVENESS

We first consider effectiveness properties of concrete data types.

The basic predicates and operations of a concrete data type are computable and total. So, these predicates and the graphs of these operations are effectively decidable subsets of their domains of arguments and results.

Consider, for instance, the data type of naturals.

A basic operation like successor $s:\mathbf{N} \rightarrow \mathbf{N}$ is a total computable function, thus its graph is an effectively decidable subset $S:=\{<m,n> \in \mathbf{N} \times \mathbf{N} / s(m)=n\}$ of $\mathbf{N} \times \mathbf{N}$, so it is effectively enumerable.

A basic predicate such as less than $<$, being computable and total, has as its extension an effectively decidable subset $Lt:=\{<m,n> \in \mathbf{N} \times \mathbf{N} / m<n\}$ of $\mathbf{N} \times \mathbf{N}$. Similarly, the extension of the equality test is an effectively decidable subset $Eq:=\{<m,n> \in \mathbf{N} \times \mathbf{N} / m=n\}$ of $\mathbf{N} \times \mathbf{N}$.

Now, the tests used in program constructs, like **if_then_else_**, are Boolean combinations of these basic tests, so they are effectively decidable subsets of their domains of arguments, as well. For instance, a test like $x<y \wedge \neg y=z$ corresponds to an effectively decidable subset of $\mathbf{N} \times \mathbf{N} \times \mathbf{N}$.

4.2 PROGRAMS, PROGRAM CONSTRUCTS AND EFFECTIVENESS

We now examine effectiveness in the context of programs and program constructs, first the deterministic and then the nondeterministic case.

We first recall some simple remarks on effectiveness [Rogers '67; Shoenfield '67; Manna '74].

On the natural numbers, a partial function is computable iff its graph is an effectively enumerable set of input-output pairs. In other words, the partial computable functions are those functional relations whose graphs are effectively enumerable relations. Also, a set of naturals is effectively decidable iff it and its complement are both effectively enumerable.

Now, let us consider programs and their input-output behaviours.

By the *input-output behaviour* of program (segment) p we mean the set $\langle p \rangle$ consisting of its input-output pairs $\langle u,z \rangle$. This concept of input-output behaviour is quite familiar for the case of a deterministic program (segment), and it is naturally used also for possibly nondeterministic programs.

One usually considers (deterministic) programs as descriptions of effective procedures for computing partial functions. So, a binary relation is the input-output behaviour of a deterministic program (segment) iff it is functional and effectively enumerable. In this sense, a deterministic program (segment) p may be regarded as a description of an effectively enumerable relation: its (functional) input-output behaviour $\langle p \rangle$.

For possibly nondeterministic programs we take this idea as basic: program segments as descriptions of effectively enumerable input-output relations.

We see that a nondeterministic program can be employed as a (nondeterministic) procedure for enumerating its input-output pairs. Also, the input-output behaviour of such a nondeterministic program can be regarded as the union of deterministic behaviours obtained by selecting among the various choices left open.

On the other hand, an effectively enumerable binary relation can be "executed", at least by a "British Museum" schema. (The procedure for effectively enumerating input-output pairs provides a procedure for transforming (some) inputs to corresponding outputs.) Thus, such an effectively enumerable input-output relation, being "executable", may be regarded as the input-output behaviour of a, possibly nondeterministic, program (segment).

Now, let us consider program constructs, such as sequentialisation and conditionals, as tools for constructing new program segments.

Program constructs are intended to be applied to appropriate program segments to produce other program segments. So, they should preserve effectively enumerable input-output behaviours.

For instance, consider the sequentialisation construct ;. The compound statement R ; S is a program segment constructed from program segments R and S. So, program construct _ ; _ should preserve effectively enumerable input-output behaviours.

Similarly, for a test t, **if** t **then** P **else** Q is a program segment constructed from program segments P and Q. So, for an effective decidable test t

program construct **if** t **then** _ **else** _ should preserve effectively enumerable input-output behaviours.

Thus, program constructs may be regarded as operations on programs and program segments preserving effectively enumerable input-output behaviours.

4.3 Effective Sets and Programs

We shall use in the sequel the concept of effective set. We shall say that a set U is *effective* when equality between its elements is effectively decidable relative to $U^2=U\times U$, in the sense that there exists an effective procedure for deciding whether or not u=v for every pair $\langle u,v\rangle \in U\times U$.

Notice that the cartesian product $U\times W$ of effective sets U and W is again an effective set (since the equality $\langle u',w'\rangle=\langle u'',w''\rangle$ reduces to the equalities between components u'=u'' and w'=w'').

With these observations on effectiveness, we can more clearly see why the input-output behaviour of a program (segment) on an effective set is an effectively enumerable relation of input-output pairs.

Indeed, consider a program (segment) on an effective set. It is constructed from basic tests, which are effectively decidable, and functions, which are effectively computable, by applying program constructs, which preserve effective enumerability. Thus, we can then see that its input-output behaviour is an effectively enumerable relation of input-output pairs.

Now, consider a partial function f on an effective set U. When its graph $F:=\{\langle u,v\rangle \in U\times U/f(u)=v\}$ is effectively enumerable we shall call f *effectively computable*. The reason for this name is that an effective procedure for enumerating the graph F of f provides an effective procedure for computing f.

One can effectively compute f as follows: given $u\in U$, we effectively enumerate those $\langle v,w\rangle \in F$, decide whether u=v and in such case output w.

So, such an effectively computable partial function on an effective set has an effective procedure for computing results from arguments in its domain.

5. Extended Relations and Effectiveness

We have already argued for the desirability of suitably extending the relational approach to make it more adequate for programming. We now wish to see that this extension does provide an adequate framework for programming.

We will argue that a certain set of extended relational terms can be regarded as effective procedures describing programs. For this purpose, we first make some considerations that will give some guidelines for the selection of some "algorithmic" symbols and will also provide a first explanation, in terms of effectiveness, for their computing-like nature.

5.1 Effectiveness and Extended Operations and Constants

We mentioned that we can single out certain operations and constants,

which deserve being called "algorithmic". We now examine some guidance towards the selection of such symbols. These criteria will also provide a first explanation, in terms of effectiveness, for their computing-like behaviour.

First, consider subsets of a universal relation V.

Then, the Boolean operations $\cup$, $\cap$ and $^{\sim}$ preserve effective decidability:

- if r and s are effectively decidable subsets of V, so are $r \cup s$, $r \cap s$ and $r^{\sim}$.

Also, Boolean operations $\cup$ and $\cap$ preserve effective enumerability:

- if r and s are effectively enumerable, then so are $r \cup s$ and $r \cap s$.

Clearly, the empty relation $\varnothing$ is an effectively decidable subset of V.

We now assume that the universal relation V is an equivalence relation $V \subseteq U \times U$ on U. Then, transposition ^{T} preserves effective decidability:

- if r is an effectively decidable subset of V, then so is r^{T}.

We now also assume that set U is an effective set, whose equality is effectively decidable.

Let us consider the Peircean constant identity and operation composition.

Clearly, the constant identity relation $1_U = \{<u,u> \in U \times U / u \in U\}$ is an effectively decidable subset of $U \times U$. Also, 1_U is a functional and total relation. Thus:

- identity relation $1_U = \{<u,u> \in U \times U / u \in U\}$ is
  an effectively decidable subset of $U \times U$, and
  an effectively computable total function.

Concerning operation composition, we can then see that, though not preserving effective decidability, composition | does preserve effective enumerability:

- if relations r and s are effectively enumerable, then so is r|s.

Indeed, if r and s are effectively enumerable, then one can effectively enumerate r|s by effectively enumerating those $<x,y> \in r$ and $<z,w> \in s$, deciding whether v=z and in such case outputting <x,w>.

We consider a special case where composition | does preserve effective decidability: pre composition by an effectively computable partial function. We can see that for an effectively computable partial function f:

- if s is an effectively decidable subset of V, then so is f|s.

Indeed, in this case one can effectively decide whether or not $<u,w> \in f|s$ by effectively computing v=f(u) and deciding whether or not $<u,v> \in s$.

For the structural operations, we assume that set U is a structured universe U equipped with a closed equivalence relation $V \subseteq U \times U$ on U.

Now, consider structural operations fork $\angle$ and parallel product $/\!/$.

Operations fork $\angle$ and parallel product $/\!/$ preserve effective enumerability:

- if relations r and s are effectively enumerable, then so are $r \angle s$ and $r /\!/ s$.

For effectively enumerable relations r and s, one can effectively enumerate $r \angle s$ by effectively enumerating those $<x,v> \in r$ and $<z,w> \in s$, deciding whether x=z and in such case forming the ordered pair <v,w> and

outputting <x,<v,w>>.

Now, let us consider the remaining extended constants.

We further assume that the universal equivalence relation $V \subseteq U \times U$ is effectively enumerable. So:

- universal relation V is total and effectively enumerable.

We now examine the remaining extended constants: the structural ones.

We can also see that the projections $\pi_V = \{<<x,y>,x> \in V / <x,y> \in V\}$ and $\rho_V = \{<<x,y>,y> \in V / <x,y> \in V\}$ are effectively enumerable. They are also functional relations. Thus:

- the left and right projections π_V and ρ_V are
  effectively enumerable subsets of $V \subseteq U \times U$, and
  effectively computable partial functions.

For instance, one can effectively compute π_V as follows: given $u \in U$, we effectively enumerate those $<x,y> \in V$, decide whether $u=<x,y>$ and in such case output x.

We can see that the constant duplication $2_U = \{<u,<u,u>> \in V / u \in U\}$ is effectively enumerable. Also, 2_U is a functional and total relation. So:

- duplication $2_U = \{<u,<u,u>> \in V / u \in U\}\}$ is
  an effectively enumerable subset of $V \subseteq U \times U$, and
  an effectively computable total function.

Also, the equality sieve $2_U^T = \{<<u,u>,u> \in V / u \in U\}$ is a functional relation that is effectively enumerable, as is the empty relation $\varnothing$. Thus:

- equality sieve $2_U^T = \{<<u,u>,u> \in V / u \in U\}$ is
  an effectively enumerable subset of $V \subseteq U \times U$, and
  an effectively computable partial function.

We can now examine the structural operations concerning effective computability of partial functions and effective decidability.

Operations fork $\angle$ and parallel product $/\!/$ preserve effective computability:

- if r and s are effectively computable, so are $r \angle s$ and $r /\!/ s$.

We can also see that the structural operations fork $\angle$ and parallel product $/\!/$ transform effective decidable subsets:

- if r and s are effectively decidable subsets of $U' \times W'$ and $U'' \times W''$, respectively, then
  their fork $r \angle s$ is an effectively decidable subset of $(U' \cap U'') \times (W' \times W'')$,
  $r /\!/ s$ is an effectively decidable subset of $(U' \times U'') \times (W' \times W'')$.

Consider effectively decidable subsets r and s of $U' \times W'$ and $U'' \times W''$, respectively. Then, given $<u,w> \in (U' \times U'') \times (W' \times W'')$, one can effectively decide whether $<u,w> \in r /\!/ s$ by applying the projections for effectively computing the components $u_1 = \pi_V(u) \in U'$ and $u_2 = \rho_V(u) \in U''$ of u as well as $w_1 = \pi_V(w) \in W'$ and $w_2 = \rho_V(w) \in W''$ of w, and deciding whether $<u_1, w_1> \in r$ and $<u_2, w_2> \in s$.

Finally, let us also examine partial identities for representing sets.

Given a subset S of U, consider its partial identity $1_S=\{<u,u>\in U\times U/u\in S\}$. Clearly, partial identity $1_S\subseteq 1_U$ is an effectively decidable (respectively enumerable) subset of U×U iff S is an effectively decidable (respectively enumerable) subset of U. Thus, for each subset $S\subseteq U$:

- S is an effectively decidable subset S of U iff
  its partial identity 1_S is an effectively decidable subset of U×U;
- S is an effectively enumerable subset S of U iff
  its partial identity 1_S is effectively enumerable subset of U×U
  (i. e. 1_S is a partial computable function).

In other words, the bijective transformation $\underline{1}:\wp(U)\rightarrow\wp(1_U)$ preserves effective decidability as well as effective enumerability.

5.2 Extended Relations and Effective Structural Equivalence

By an *effective structural equivalence* we mean a structural equivalence whose set U has an effectively decidable equality and whose closed equivalence relation $V\subseteq U\times U$ is effectively enumerable. With this concept of effective structural equivalence, we can conveniently summarise our considerations as follows.

All the extended constants are effectively enumerable.

- The universal relation $V\subseteq U\times U$ is effectively enumerable.
- The extended constants (but V) are effectively computable functions:
  relations $\varnothing$ and 1_U are functional effectively decidable subsets of V;
  relations 2_U, 2_U^T, π_V and ρ_V are functional and effectively enumerable.

The extended operations (except ~) preserve effective enumerability:

- the family of effectively enumerable subsets of V is closed under
  the extended operations $\cup$, $\cap$, |, ^{T}, $/\!\!/$ and $\angle$.

Operations $\cup$, $\cap$, ~ and ^{T} preserve effective decidability of subsets of V:

- the family of effectively decidable subsets of V is closed under
  the Boolean operations $\cup$, $\cap$ and ~ as well as Peircean transposition ^{T}.

The extended operations (but ~, $\cup$ and ^{T}) preserve effective computability:

- the family of effectively computable partial functions is closed under
  the extended operations $\cap$, |, $\angle$ and $/\!\!/$.

Let us use **Dcd**(W) and **Enm**(W), respectively, for the families of effectively decidable and effectively enumerable subsets of a given set W.

Concerning the representation of subsets by partial identities, the transformation $\underline{1}:\wp(U)\rightarrow\wp(1_U)$ establishes bijections

- between the families **Dcd**(U) and $\mathbf{Dcd}(1_U)\subseteq\mathbf{Dcd}(V)$, as well as
- between the families **Enm**(U) and $\mathbf{Enm}(1_U)\subseteq\mathbf{Enm}(V)$.

This means that effective properties are not lost by this representation of subsets by partial identities.

5.3 Algorithmic Symbols and Terms

We now have some guidance towards selecting our algorithmic symbols.

According to the observations on effectiveness and programming, we shall use as main criteria effective enumerability and its preservation.

We will first classify the extended relational constants and operations according to these effectiveness criteria as well as auxiliary ones. Next, we shall introduce our algorithmic symbols and a first explanation, in terms of effectiveness, for their computing-like nature.

A. Classification of Extended Constants and Operations

Based on what we have seen, we can classify the extended relational constants and operations according to some properties and their preservation.

We have considered the relational criteria of totality and functionality. For an effective structural equivalence, consisting of a set U with an effectively decidable equality and a closed equivalence relation $V \subseteq U \times U$ that is effectively enumerable, we have also considered some effectiveness criteria: effective decidability, enumerability and computability.

The transformation $\underline{1} : \wp(U) \to \wp(1_U)$ represents subsets of U as functional partial identities. It transforms subsets to partial identities as follows:

subset $D \in \mathbf{Dcd}(U)$	partial identity $1_D \in \mathbf{Dcd}(U \times U)$
subset $E \in \mathbf{Enm}(U)$	partial identity $1_E \in \mathbf{Enm}(U \times U)$

We have seen that the universal relation V is a total effectively enumerable relation.

We can classify the remaining extended relational constants, which are functional, with respect to totality and effectiveness as follows.

empty relation $\varnothing$	partial	$\mathbf{Dcd}(V)$
identity relation 1_U	total	$\mathbf{Dcd}(V)$
duplication 2_U	total	$\mathbf{Enm}(V)$
equality sieve 2_U^T	partial	$\mathbf{Enm}(V)$
projections π_V and ρ_V^T	partial	$\mathbf{Enm}(V)$

We can also classify the extended operations according to the preservation of some families of binary relations.

Operation	$\wp(1_U)$	$\mathbf{Dcd}(V)$	$\mathbf{Enm}(V)$
union $\cup$	+	+	+
intersection $\cap$	+	+	+
complementation $\tilde{}$	-	+	-
composition $\mid$	+	-	+
transposition ^{T}	+	+	+
fork $\angle$	-	?	+
parallel product $/\!/$	+	?	+

In a similar spirit, we can also classify some useful derived operations according to the preservation of some families of binary relations.

Derived operation	$\wp(1_U)$	**Dcd**(V)	**Enm**(V)
domain _.	+	-	+
identity complementation ^	+	+	-
branching union $\vee_t$	+	+ if t∈ **Dcd**(V)	+ if t,^t∈ **Enm**(V)
guarded union $r_p[]_q s$	+	+ if p,q∈ **Dcd**(V)	+ if p,q∈ **Enm**(V)

B. Algorithmic Constants, Operations and Terms

On the basis of the preceding classifications, we now introduce the algorithmic symbols.

We will call

constants $\varnothing$, 1_U, V, 2_U, 2_U^T, π_V and ρ_V the *algorithmic constants*, and

operations $\cup$, $\cap$, |, ^{T}, $/\!\!/$ and $\angle$ the *algorithmic operations*.

Notice that the algorithmic operations are monotonic with respect to inclusion $\subseteq$.

The *algorithmic symbols* will be the algorithmic operations together with the algorithmic constants. By an *algorithmic relation* we mean an extended relational term built with algorithmic constants and operations.

The reason for these names should now be clear.

Consider an effective structural equivalence, consisting of a set U with an effectively decidable equality and a closed equivalence relation $V \subseteq U \times U$ that is effectively enumerable. We then have:

- the algorithmic constants are effectively enumerable,
  (the functional ones (all but V) are effectively computable);
- the algorithmic operations preserve effective enumerability;
- the algorithmic relations are effectively enumerable.

This justification for these names also provides an explanation for their computing-like nature. With proper allowance for nondeterminism, algorithmic constants and relations can be regarded as programs, while algorithmic operations can be viewed as program constructs.

In this sense, we may regard our algorithmic symbols as, possibly nondeterministic, program schemas.

We now consider a given set $T \subseteq \wp(V)$ of relations on U.

The *algorithmic closure of* set T is the set Alg[T] consisting of the relations built from algorithmic constants and those in T by means of repeated applications of algorithmic operations.

We shall call an extended relation t *testable over* T iff both t and its complement $t^\frown$ are in the algorithmic closure of T: $t \in Tst[T]$ iff $\{t,t^\frown\} \subseteq Alg[T]$).

By an *algorithmic term over* T we mean an extended relational term built from algorithmic closure of T by algorithmic operations as well as branching unions over partial identities that are testable over T (i. e. $\vee_t$ for $t \in \wp(1_U)$ with $\{t,t^\frown\} \subseteq Alg[T]$).

Since the algorithmic operations are monotonic with respect to inclusion $\subseteq$, so are the algorithmic terms over a set.

Again, the reason for this terminology should be clear by now. Consider an effective structural equivalence.

Then, effective procedures for effectively enumerating the relations in T, provide effective procedures for:

- effectively enumerating the relations in the algorithmic closure of T,
- effectively deciding the testable relations over T,
- effectively enumerating the algorithmic terms over T.

So, if the set T consists of effectively enumerable relations, then we have:

- the relations in the algorithmic closure of T are effectively enumerable,
- the testable relations over T are effective decidable,
- the algorithmic terms over T are effectively enumerable.

In other words, if $T \subseteq \mathbf{Enm}(V)$ then $Tst[T] \subseteq \mathbf{Dcd}(V)$ and the algorithmic terms over T are in $\mathbf{Enm}(V)$.

In this sense, the algorithmic terms over a set of effectively enumerable relations may be viewed as programs, while the algorithmic terms over a set may be regarded as program schemas.

6. PROGRAMMING LANGUAGE CORRESPONDENCE

We now reconsider our algorithmic symbols and terms and examine a second explanation, in terms of a programming view, for their computing-like nature.

The programming view relies on a behavioural correspondence between these algorithmic symbols and programming language constructs. We shall consider some concepts and constructs akin tot those found in the usual programming languages [Ghezzi & Jazayeri '82].

6.1 ALGORITHMIC SYMBOLS AND PROGRAM SCHEMAS

Let us consider some concepts and constructs found in the usual programming languages [Ghezzi & Jazayeri '82]. We notice that we have a correspondence between the behaviours of some of them and some algorithmic symbols.

We first examine a correspondence between some algorithmic constants and some programming concepts. Notice that the following matchings preserve behaviours.

Constant relation	Behaviour	Concept
empty relation $\varnothing$	$\langle u,v\rangle \notin \varnothing$	abort
identity relation 1_U	$u \xrightarrow{1_U} v$	transfer v:=u
universal relation $V=U\times U$	$v \xrightarrow{U^2} w$	nondeterministic assignment w:=?
duplication 2_U	$u \xrightarrow{2_U} \begin{pmatrix} u \\ , \\ u \end{pmatrix}$	making two copies of the input

defined projections $\pi_V := (1_U \angle V)^T$ and $\rho_V := (V \angle 1_U)^T$	$z = \begin{pmatrix} x \\ , \\ y \end{pmatrix} \xrightarrow{\pi} x$ $z = \begin{pmatrix} x \\ , \\ y \end{pmatrix} \xrightarrow{\rho} y$	component selection x:=z.lft y:=z.rgt

We now examine a correspondence between some algorithmic operations and some program constructs, also preserving behaviours.

Operation	Behaviour	Construct
composition r|s	$u \xrightarrow{r} v \xrightarrow{s} w$	sequentialisation first r ; then s
parallel product r//s	$u = \begin{pmatrix} u_1 \\ , \\ u_2 \end{pmatrix} \longrightarrow \left\langle \begin{matrix} u_1 \xrightarrow{r} v_1 \\ \& \\ u_2 \xrightarrow{s} v_2 \end{matrix} \right\rangle \longrightarrow \begin{pmatrix} v_1 \\ , \\ v_2 \end{pmatrix}$	parallel processing of components
operation fork r∠s	$u \longrightarrow \left\langle \begin{matrix} u \xrightarrow{r} v \\ \& \\ u \xrightarrow{s} w \end{matrix} \right\rangle \longrightarrow \begin{pmatrix} v \\ , \\ w \end{pmatrix}$	feeding common input to both r and s

We also notice matchings between some derived algorithmic operations and some important program constructs.

The conditional constructs for selection of alternatives guarded conditional **if**_⇒_[]_⇒_**if** and deterministic conditional **if_then_else_** have the following input-output behaviours

- $\langle$ **if** p(u,u) ⇒ r(u,z) [] q(u,u) ⇒ s(u,z) **fi** $\rangle$ =
  $=\{\langle u,z\rangle \in U\times Z / (p(u,u) \wedge \langle u,z\rangle \in \langle r\rangle) \vee (q(u,u) \wedge \langle u,z\rangle \in \langle s\rangle)\}$,
- $\langle$ **if** t(u,u) **then** r(u,z) **else** s(u,z) $\rangle$ =
  $=\{\langle u,z\rangle \in U\times Z / (t(u,u) \wedge \langle u,z\rangle \in \langle r\rangle) \vee (\neg t(u,u) \wedge \langle u,z\rangle \in \langle s\rangle)\}$.

We thus have the following matchings between some derived algorithmic operations and some program constructs, which preserve behaviours

Operation	Behaviour	Construct
guarded union $r_p[]_q s$	$u \longrightarrow \left\{ \begin{matrix} u \xrightarrow{p} u \xrightarrow{r} v \\ \vee \\ u \xrightarrow{q} u \xrightarrow{s} w \end{matrix} \right\} \longrightarrow \left\{ \begin{matrix} v \\ or \\ w \end{matrix} \right\}$	nondeterministic selection of alternatives {guarded if}
branching union $r \vee_t s$	$u \longrightarrow \left\{ \begin{matrix} u \xrightarrow{t} u \xrightarrow{r} v \\ \vee \\ u \xrightarrow{\hat{}t} u \xrightarrow{s} w \end{matrix} \right\} \longrightarrow \left\{ \begin{matrix} v \\ or \\ w \end{matrix} \right\}$	deterministic selection of alternatives {**if_then_else_**}

We thus see that this programming language viewpoint indicates partial correspondences

- between some algorithmic constants and some programming concepts,
- between some algorithmic operations and some program constructs,

which preserve their behaviours.

6.2 Other Algorithmic Symbols and Programming Concepts

An algorithmic operation that does not explicitly appear in the preceding correspondence is Boolean union. But, we know that union $\cup$ can be recovered from guarded union as a restricted union over the total identity 1_U. So, $r \cup s$ corresponds to **if** *true* $\Rightarrow$ r [] *true* $\Rightarrow$ s **if**.

Some algorithmic operations - such as intersection and transposition - do not appear to have a direct programming counterpart. But, as already noticed, on effective structural equivalences their behaviours preserve effective enumerability. We can also notice other features.

First, consider the equality sieve $2_U{}^T=\{<<u,v>,w> \in V/u=v=w\}$. It is an effectively enumerable relation whose functional input-output behaviour, corresponds to a conditional program schema, as it is not difficult to see.

Now, operation $\cap$ preserves effective enumerability as well as decidability. Moreover, it can be derived from fork $\angle$ and equality sieve $2_U{}^T$ via $r \cap s=(r \angle s)|2_U{}^T$. This definition provides an execution schema for intersection of relations, albeit a not very efficient one.

Finally, consider operation ^{T}. Transposition corresponds to running the program "backwards": from output to input. This idea fits very well with the logic programming paradigm, where "execution" relies on unification matching, and thus the input-output distinction becomes somewhat blurred [Kowalski '79]. Also, the very argument showing that transposition preserves effective enumerability indicates how it can be simulated; indeed operation ^{T} can be expressed by an algorithmic term.

Dually, some programming ideas - such as program variables - do not appear to have a direct algorithmic term counterpart. But, as illustrated in the examples, we can express at least some programs by means of algorithmic terms. We shall take a closer look at this point in a forthcoming report.

7. OTHER ASPECTS

We now comment on some other aspects, of importance in the context of a framework for program development. Some of these aspects are addressed in companion reports.

7.1 Cartesian Product and Pair Coding

We first consider the role of cartesian product and pair forming.

The structural operations and constants were motivated by the idea of forming pairs and parallel application. For this reason, we have considered structured universes as sets closed under cartesian product.

This is just a simplified manner of presenting the basic ideas. We can adopt a more flexible approach based on the concept of pair coding.

The idea is that one does need actual cartesian pairs; some coding for them is enough. For instance, for the universe U=**N** of natural numbers, one might consider a Gödel-like coding $*:\mathbf{N}\times\mathbf{N}\rightarrow\mathbf{N}$, given by, say, $m*n:=2^m.(2n+1)$, coding pair $<m,n>$ of naturals by the single natural $m*n\in\mathbf{N}$.

The intuition is that, as long as one can recover the given arguments from the coded pair, one does not care about the particular coding schema adopted. It can be thought of as an internal matter left to the system.

In this sense, we can replace the ideas of pair forming and closure under cartesian product by an injective function $*:U\times U\rightarrow U$. (More generally, we can even use a coding relation $*\subseteq(U\times U)\times U$ whose restriction to $V\subseteq U\times U$ is an injective function ${}_V|*:V\rightarrow U$.)

For effectiveness considerations, we require the function ${}_V|*:V\rightarrow U$ to be effectively computable. One can then see that the previous arguments carry over to this more general case, simply by replacing expressions like "forming the ordered pair $<u,v>$" by "effectively computing $u*v$".

Thus, the preceding considerations carry over to this more flexible approach based on the concept of pair coding ${}_V|*:V\rightarrow U$.

7.2 Expressing and Reasoning about Properties

We have seen that algorithmic symbols provide an adequate framework for expressing programs and program schemas. We shall now briefly comment on expressing properties and reasoning about them.

The expressive of the extended relational language is that of first-order logical language. The expressivity theorem [Veloso & Haeberer '91; Haeberer & Veloso '91] guarantees that every first-order formula φ can be (effectively) converted to a closed (partial identity) term $\varphi\#$ "with the same extension". Thus, the partial identity of each m-ary relation definable by a first-order formula φ is binary relation that is definable by a closed extended relational term $\varphi\#$.

We clearly also have the converse: the input-output behaviour every of every closed extended relational term is also definable by a first-order formula.

In this sense, the extended relational language can be regarded as a truly relational counterpart of first-order logical language

Furthermore, this expressivity carries over to any applied first-order logical language. In this sense, these extended relational languages can be regarded as relational counterparts of first-order logical languages [Veloso & Haeberer '91].

We can match the repertoire of predicates, operations and constants of such an applied first-order logical language with a repertoire of relation constants. Given such a matching, we can translate back and forth between first-order formulae and closed terms "with the same extension".

We also have a matching between first-order sentences σ and equations σ^* between closed terms.

Moreover, these back-and-forth translations match first-order reasoning rules and axioms with equational rules and equations between extended relational terms. To accomplish a matching of deductive powers, we use as axioms a finite set of equations between extended relational terms. This finite set of equations axiomatises the so-called Algebraic Fork Calculus AFC.

The Representation Theorem guarantees that every model of AFC can be represented as an algebra of extended input-output relations [Frias et al. '93, '95]. As a consequence of the Representation Theorem, we have the soundness and completeness of this calculus: a sentence of the extended relational language is derivable within AFC iff it holds in all algebras of extended input-output relations.

We then have the desired matching of deductive powers. Thus, we can safely replace first-order reasoning by equational reasoning within our extended relational calculus. But we do not have to; whenever it is more convenient we can resort to first-order reasoning, with the assurance that it can be translated into AFC. Further, representability provides an added bonus: we can reason by means of individuals, which is often more intuitive when one wishes to think in an input-output manner (by resorting to diagrams, for instance); if the conclusion no longer involves individuals it can be derived within AFC [Veloso & Haeberer '93].

We have been considering mostly unsorted situations and languages. But, these ideas can be extended to the many-sorted case in a reasonably straightforward way. This can be done by a relational version of the reduction of many-sorted first-order logic to unsorted logic by relying on relativisation predicates. Thus, we can mimic many-sorted first-order reasoning by equational reasoning within our extended relational calculus.

7.3 Extended Relational Framework for Programming

We shall now briefly comment on the role of this extended relational framework for program development.

We have indicated that within the extended relational framework one can express programs, input-output specifications, data type specifications, programming methods. We have mentioned that program derivation strategies can be formulated in a similar spirit. Furthermore, from such relational input-output specifications one can derive, by algebraic manipulations, (possibly recursive) programs. Moreover, programs, expressed in these terms, can also be transformed in a similar fashion, say for obtaining more efficient versions [Haeberer & Veloso '91].

This extended relational framework presents wide expressive, deductive and transformational powers, which are desired in an framework for program development. Within it one can:

- express behavioural specifications and programs,
- reason about their properties (in an equational manner),

- transform specifications and programs (in an algebraic fashion).

This extended relational framework supports a wide-spectrum language and calculus for program derivation. Within it, we are able to:

- express input-output specifications, programs and programming methods by terms (constrained by equations);
- express data type specifications by equations between terms;
- employ such equations to compare and transform terms, for instance for transforming a specification of input-output behaviour into a program.

The pragmatic adequacy of this extended relational framework for program derivation has been extensively illustrated elsewhere by means of case studies [Durán & Baum '93; Frias '93; Frias et al. '93; Haeberer & Veloso '91; Vázquez & Elustondo '89; Veloso & Haeberer '93,'94], where more references and comparisons with other approaches can be found.

Related ideas have also been employed in connection with problem solving as well as with some epistemological aspects of the process of software development [Haeberer & Veloso '89, '90; Haeberer et al. '89].

8. CONCLUSIONS

In two previous reports [Veloso '96a, b], we introduced the question of adequacy of an extended relational framework for program development. We argued that the familiar apparatus of binary relations must be extended to be appropriate for programming and we examined the nature of such an extension obtained by considering relations on structured universes with new structural operations.

A basic issue concerning such a programming framework concerns its adequacy for expressing programs. The adequacy of such an extended apparatus hinges on having an appropriate expressive power.

In this report we have examined the adequacy of an extended relational framework for program development. We have argued that we can select, by effectiveness considerations, a certain set of symbols deserving the name "algorithmic". We have then outlined a programming language correspondence, which indicates that these algorithmic symbols provide adequate power for expressing programs.

This provides two explanations for the selection of the algorithmic part and its computing-like nature, which jointly justify its adequacy. First, the identification of the proper repertoire is based on effectiveness, which guarantees soundness, in the sense that one does not leave the effective realm. The second - more intuitive - explanation relies on the programming language correspondence, which, besides reinforcing the first explication, indicates that we have adequate expressive power.

We have begun by reviewing some of the material of the two previous reports. In section 2 we have recalled some basic operations on relations and presented of series of examples, intended to illustrate how one can express programming ideas in a relational form. In section 3 we have examined the proposed extension and the new structural operations as

well as some of their properties.

In section 4 we have examined some considerations connecting algorithms and program constructs to effectiveness, mainly for preparing the terrain for a classification of the extended relational symbols. This classification has been presented in section 5 as a basis for the selection of the algorithmic symbols, which has been explained in terms of effective properties and their preservation. We have then moved on to the programming language correspondence, which has been presented in section 6, indicating that these algorithmic symbols provide adequate power for expressing programs. Some other aspects, of importance in the context of a framework for program development, have been briefly commented upon in section 7.

This report is the third one of a series of reports addressing the question of adequacy of a fork relational framework for program development. Other reports focus on other, related aspects of this question.

REFERENCES

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1975) *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

Bauer, F. L. and Wössner, H. (1982) *Algorithmic Language and Program Development*. Springer-Verlag, Berlin.

Baum, G. A., Haeberer, A. M. and Veloso, P. A. S. (1991) On the representability of the ∇ abstract relational algebra. *IGPL Newsletter*, Interest Group on Propositional and Predicate Logic, **1**(3), 3-4.

Berghammer, R. (1991) Relational specification of data types and programs. Univ. Bundeswehr, Res. Rept., München.

Berghammer, R., Haeberer, A. M., Schmidt, G. and Veloso, P. A. S. (1993) Comparing two different approaches to products in abstract relation algebras. In Nivat, M., Rattray, C., Rus, T. and Scollo, G. (eds.) *Algebraic Methodology and Software Technology (AMAST' 93)*, Springer-Verlag, London, 169-176.

Berghammer, R. and Zierer, H. (1986) Relational algebraic semantics of deterministic and nondeterministic programs. *Theor. Computer Sci.*, 43, 123-147.

Broy, M. (1983) Program construction by transformations: a family of sorting programs. In Breuman, A. W. and Guino, G. (eds) *Automatic Program Construction*, Reidel, Dordrecht.

Burstall, R. M. and Darlington, J. (1977) A transformational system for developing recursive programs. *J. Assoc. Comput. Mach.* **24**(1), 44-67.

Codd, E. F. (1972) Relational completeness of data base sublanguages. In *Data Base Systems*. Courant Computer Science Symposium, vol 6.

Darlington, J. (1978) A synthesis of several sorting algorithms. *Acta Informatica*, **11**(1), 1-30.

Durán, J. E. and Baum, G. A. (1993) Construcción formal de programas a partir de especificaciones en un cálculo de relaciones binarias extendido. PUC-Rio, Dept. Informática, Res. Rept. MCC 5/93, Rio de Janeiro.

Ebbinghaus, H. D., Flum, J. and Thomas, W. (1984) *Mathematical Logic.* Springer-Verlag, Berlin.

Enderton, H. B. (1972) *A Mathematical Introduction to Logic.* Academic Press, New York.

Elustondo, P. M., Veloso, P. A. S., Haeberer, A. M. and Vázquez, L. A. (1989) Program development in the algebraic theory of problems. *18 Jornadas Argentinas de Informática e Investigación Operativa*, Buenos Aires, 2.2-2.32.

Frias, M. (1993) The ∇-extended relation algebra as a deductive and object-oriented database language. Univ. Buenos Aires, Fac. Ciencias Exactas y Naturales, Res. Rept., Buenos Aires.

Frias, M., Aguayo, N. and Novak, B. (1993) Development of graph algorithms with the ∇-extended relation algebra. *XIX Conf. Latinoamericana de Informática*, Buenos Aires, 529-554.

Frias, M. F., Baum, G. A., Haeberer, A. M. and Veloso, P. A. S. (1993) A representation theorem for fork algebras. PUC-Rio, Dept. Informática, Res. Rept. MCC 29/93, Rio de Janeiro.

Frias, M. F., Haeberer, A. M. and Veloso, P. A. S. (1995) A finite axiomatization for fork algebras. *Bull. of Sect. of Logic*, Univ. Lodz, **24**(4), 193-200.

Ghezzi, C. and Jazayeri, M. (1982) *Programming Languages Concepts.* Wiley, New York.

Goldblatt, R. (1982) *Axiomatising the Logic of Computer Programming.* Springer-Verlag, Berlin.

Haeberer, A.M., Baum, G. A. and Schmidt, G. (1993) On the smooth calculation of relational recursive expressions out of first-order non-constructive specifications involving quantifiers. *Intern. Conf. Formal Methods on Programming and its Applications.* Springer-Verlag, Berlin.

Haeberer, A. M. and Veloso, P. A. S. (1989) On the inevitability of program testing: a formal analysis. In Gonnet, G. H. (ed.) *IX Conf. Intern. Soc. Chilena de Ciencia de la Computación, vol. I: Trabajos de Investigación*, Santiago, Chile, 208-240.

Haeberer, A. M. and Veloso, P. A. S. (1990) Why software development is inherently non-monotonic: a formal justification. In Trappl, R. (ed.) *Cybernetics and Systems Research*, World Scientific Publ. Corp., London, 51-58.

Haeberer, A. M. and Veloso, P. A. S. (1991) Partial relations for program derivation: adequacy, inevitability and expressiveness. In Smith, D. (ed.) *Proc. IFIP TC2 Working Conf. Constructing Programs from Specifications*, Pacific Grove, CA, 310-352 {revised version in Möller, B. (ed.) *Constructing Programs from Specifications*, North-Holland, Amsterdam, 319-371}.

Haeberer, A. M., Veloso, P. A. S. and Baum, G. A.(1989) *Formalización del Proceso de Desarrollo de Software*. Kapelusz, Buenos Aires.

Horowitz, E. and Sahni, S. (1978) *Fundamentals of Computer Algorithms*. Comp. Sci. Press, Potomac.

Jónsson, B. and Tarski, A. (1952) Boolean algebras with operators: part II. *Amer. J. Math*, **74**, 127-162.

Kowalski, R. (1979) *Logic for Problem Solving*. North Holland, New York.

Maddux, R. D. (1991) The origins of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, **L**(3/4), 421-455.

Manna, Z. (1974) *The Mathematical Theory of Computation*. McGraw-Hill, New York.

Möller B. (1991) Relations as a program development language. In Smith, D. (ed.) *Proc. IFIP TC2 Working Conf. Constructing Programs from Specifications*, Pacific Grove, CA, 353-376

Rogers Jr., H. (1967) *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York.

Schmidt, G. and Ströhlein, T. (1993) *Relations and Graphs: Discrete Mathematics for Computer Science*. Springer-Verlag, Berlin.

Sintzoff, M. (1985) Desiderata for a design calculus. Univ. Louvain, Unité d'Informatique, Memo, Louvain.

Shoenfield, J. R. (1967) *Mathematical Logic*. Addison-Wesley, Reading.

Tarski, A. (1941) On the calculus of relations. *J. Symb. Logic*, **6**(3), 73-89 [MR **3**(5), 130-131, May 1942].

Tarski, A. and Givant, S. (1987) *A Formalization of Set Theory without Variables*. Amer. Math. Soc. {Colloquium Publ. vol. 41}, Providence, RI.

van Dalen, D. (1989) *Logic and Structure* (2nd edn, 3rd prt). Springer-Verlag, Berlin.

Vargas, D. C. and Haeberer, A. M. (1989) Formal theories of problems. PUC-Rio, Dept. Informática, Res. Rept. MCC 6/89, Rio de Janeiro.

Vázquez, L. A. and Elustondo, P. (1989) Towards program construction in the algebraic theory of problems. *18 Jornadas Argentinas de Informática e Investigación Operativa*, Buenos Aires.

Veloso, P. A. S. (1974) The history of an error in the theory of algebras of relations. MA thesis, Univ. California, Berkeley.

Veloso, P. A. S. (1996a) On fork relations for program developmenT. PUC-Rio, Dept. Informática, Res. Rept., November 1996, Rio de Janeiro.

Veloso, P. A. S. (1996b) Fork relational frameworks on structured universes. PUC-Rio, Dept. Informática, Res. Rept., November 1996, Rio de Janeiro.

Veloso, P. A. S. and Haeberer, A. M. (1989) Software development: a problem-theoretic analysis and model. In Shriver, B. D. (ed.) *22nd Hawaii Intern. Conf. System Science, vol. II: Software Track*, Kona, HI, 200-209.

Veloso, P. A. S. and Haeberer, A. M. (1991) A finitary relational algebra for classical first-order logic. *Bull. of Sect. of Logic*, Univ. Lodz, **20**(2), 52-62.

Veloso, P. A. S. and Haeberer, A. M. (1993) On fork algebras and program derivation. PUC-Rio, Dept. Informática, Res. Rept. MCC 29/93, Rio de Janeiro.

Veloso, P. A. S. and Haeberer, A. M. (1994) On fork algebras and reasoning about programs. PUC-Rio, Dept. Informática, Res. Rept. MCC 01/94, Rio de Janeiro.

Veloso, P. A. S., Haeberer, A. M. and Baum, G. A. (1992) On formal program construction within an extended calculus of binary relations. PUC-Rio, Dept. Informática, Res. Rept. MCC 19/92, Rio de Janeiro.