ISSN 0103-9741

Monografias em Ciência da Computação
nº 41/96

On Fork Relations for Program Development

Paulo A. S. Veloso

Departamento de Informática

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, N° 41/96 ISSN 0103-9741
Editor: Carlos J. P. Lucena November 1996

On Fork Relations for Program Development *

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 FORK RELATIONS FOR PROGRAM DEVELOPMENT

Paulo A. S. VELOSO
{e-mail: veloso@inf.puc-rio.br}

PUCRioInf MCC 41/96

Abstract

We introduce the question of adequacy of a fork relational framework for program development. We first argue that the familiar apparatus of binary relations must be extended to achieve this aim. Then we suggest that an appropriate extension can be obtained by considering relations on structured universes together with new operations.

Key words: Formal specifications, program development, program derivation, relational calculi, relational algebras, fork algebras.

Resumo

Introduz-se a questão da adequação de um ambiente relacional estendido por fork para desenvolvimento de programas. Inicialmente, argumenta-se que o aparato usual de relações binárias precisa ser estendido a fim de atingir esta objetivo. Em seguida, sugere-se que uma extensão apropriada pode ser obtida considerando-se relações sobre um universo estruturado com novas operações.

Palavras chave: Especificações formais, desenvolvimento de programas, derivação de programas, cálculos relacionais, álgebras relacionais, álgebras de fork.

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 first one of a series of papers addressing the question of adequacy of a fork relational framework for program development. Subsequent papers will concentrate on other aspects of this question, such as:
Structured universe and structural operations and constants;
Effectiveness and programming language aspects in fork relations;
Algorithmic fork relations and programs.

CONTENTS

1. INTRODUCTION	1
2. MOTIVATION: PROGRAM CONSTRUCTION AND DERIVATION	1
3. SETS AND RELATIONS	2
3.1 Operations on Sets and Relations	2
3.2 Partial Identities	3
4. EXPRESSING PROGRAMMING IDEAS	4
4.1 Expressing Programs as Relations	5
4.2 Relational input-output Specifications	5
4.3 Relational Data Type Presentations	6
4.4 Expressing Programming Methods	7
4.5 Relational Presentations for many-sorted Structures	8
5. A RELATIONAL PROGRAMMING FRAMEWORK	9
6. OTHER ASPECTS	10
7. CONCLUSIONS	12
REFERENCES	13

1. INTRODUCTION

In this paper we introduce the question of adequacy of a fork relational framework for program development. We first argue that the familiar apparatus of binary relations must be extended to achieve this aim. Then we suggest that an appropriate extension can be obtained by considering relations on structured universes together with new operations.

We begin in section 2 with some motivation about program construction and derivation. In section 3 we then recall some basic operations on sets and relations and briefly review the role of partial identities for representing sets as relations. We then present in section 4 a series of examples, intended to illustrate how one can express programming ideas in a relational form and to indicate the need for an extension. We outline some desiderata for a wide-spectrum framework for program development in section 5. Some other aspects, of importance in the context of a framework for program development, are briefly commented upon in section 6. Finally, section 7 presents some concluding remarks and comments on related aspects.

This paper is the first one of a series of papers addressing the question of adequacy of a fork relational framework for program development. Subsequent papers will concentrate on other aspects of this question.

2. MOTIVATION: PROGRAM CONSTRUCTION AND DERIVATION

First we briefly present some basic ideas about program construction, which will motivate a relational approach to programming.

Program construction refers to the process of obtaining a program from a specification of its input-output behaviour in a methodical manner. An interesting variation is program derivation, where the emphasis is on obtaining the programs by formal manipulations on specifications, one often says that the program is to be calculated from its specification [Darlington '78; Broy & Pepper '81; Partsch '90].

For the purposes of program derivation, it is of interest to have a wide-spectrum formalism, supporting intermediate versions of specifications and programs as well as the manipulations transforming them [Burstall & Darlington '77; Bauer & Wössner '82; Sintzoff '85; Partsch '90].

Such a formalism will be appropriate for these purposes provided it presents some features such as adequate expressive, deductive and transformational powers. It should support:

- expression of behavioural specifications and programs,
- reasoning about their properties,
- transformations on specifications and programs.

These features will be greatly enhanced if one can manipulate and reason about its expressions, specifications or programs, with having to resort to individuals. For instance, one would like to manipulate programs without consideration of traces corresponding to particular inputs [Backus '78].

These considerations suggest a formalism with an algebraic flavour, based mainly on terms and equations between them. We would then reason about properties in an equational manner and transform expressions in an algebraic fashion.

A good candidate for such a wide-spectrum formalism that comes to mind is a calculus of binary relations. The idea is that both specifications and programs can be naturally viewed as binary relations of input-output pairs, and the transformations can be guided by properties of the operations on relations.

Indeed, relational approaches to programming ideas have received considerable attention for quite some time [Codd '72; Mili '83; Berghammer & Zierer '86; Backhouse et al. '90; Berghammer '91; Möller '91; Schmidt & Ströhlein '93].

Within a wide-spectrum framework based on a calculus of relations, the process of program derivation would take the following form:

- one expresses behavioural specifications as relational terms,
- one transforms such relational terms into terms describing programs,

A repertoire of transformations based on theorems of the calculus provides a sound basis for such transformations [Bauer & Wössner '82; Sintzoff '85; Partsch '90].

Also, in selecting which transformations to apply, one can use as guidance several kinds of intuitions. One can use intermediate goals motivated by programming considerations, such as decreasing nondeterminism or increasing efficiency. Problem-solving ideas, such as reduction and divide-and-conquer, can also be brought into play. In addition, one has some algebraic manipulations, such as factorisation, distribution and commutation.

A basic issue about such a formalism concerns its adequacy for its alleged purpose. Adequacy, in turn, has - at least - two aspects. On the one hand, we have formal adequacy. This has to do with limitations in principle, and can be settled by soundness and completeness results. On the other hand, we have less formal and more pragmatic aspects of adequacy, involving issues such as is intuitive appeal and how easy it is to handle.

In the case of a framework for programming, one basic issue is its adequacy for expressing programs. This is the issue which motivates this paper.

3. SETS AND RELATIONS

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

3.1 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 [Tarski '41; Maddux '91].

We will use two *Peircean operations*:

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

relation composition (relative product)

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

We will also employ the *Peircean constant*:

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

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

The following operations are monotonic with respect to inclusion $\subseteq$:

Boolean union $\cup$ and intersection $\cap$, and

Peircean transposition r^T and composition $|$.

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].)

3.2 Partial Identities

Let us now examine partial identities [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 .

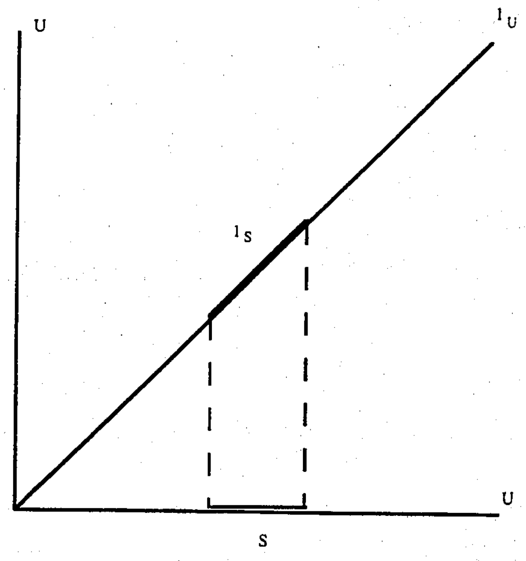
Given a subset $S\subseteq U$, by the *partial identity* on S we mean the binary relation $1_S:=\{<u,u>\in U\times U/u\in S\}$ on U. From the partial identity 1_S one can recover the set S it represents, since $S=\{u\in U/<u,u>\in 1_S\}$.

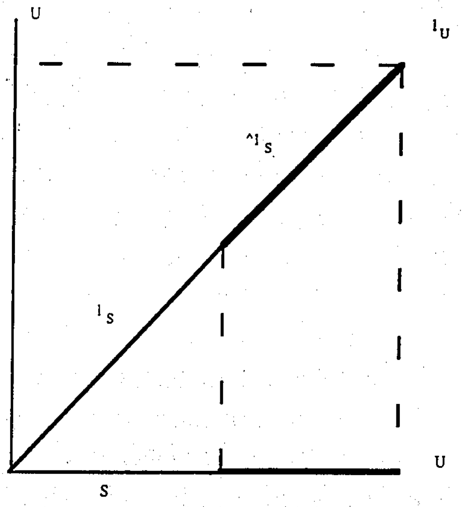
Partial identity 1_S behaves as a "filter" on S:

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: $^{\wedge}1_S:=1_U\cap 1_S^{\sim}$, for $^{\wedge}1_S=\{<u,u>\in U\times U/u\notin S\}$.



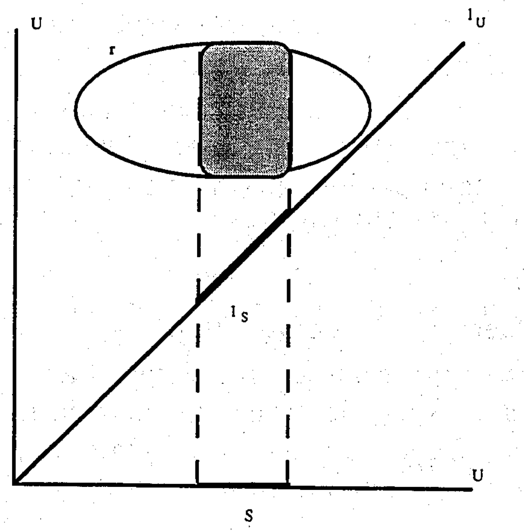


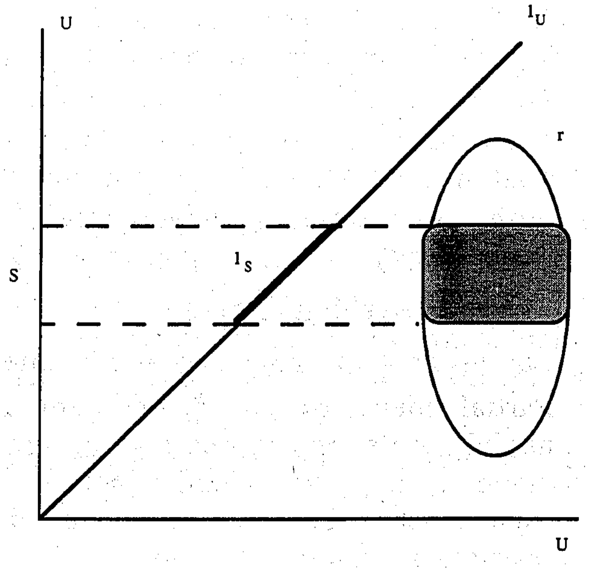
For instance, for the set $E \subseteq N$ of even naturals, we have test for evenness $1_E = \{<n,n> \in N \times N / n \in E\}$ and test for oddness $1_O = {}^\wedge 1_E = \{<n,n> \in N \times N / n \notin E\}$.

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 $r | 1_S = \{<u,v> \in r / v \in S\}$.





4. EXPRESSING PROGRAMMING IDEAS

We now illustrate how one can express programs and programming ideas by means of relations. This will suggest some extensions to the relational framework to make it more adequate for programming tasks.

We illustrate the expression of programming ideas in relational form by means of 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.

4.1 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:

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

Clearly this formulation can be immediately translated into a (recursive) program in a programming language with recursion. We wish to express it in relational terms.

First, let us examine its input-output behaviour. It can be described as follows:

$$n \xrightarrow{D} m \Leftrightarrow \begin{cases} m = 0 & \text{if } n = 0 \\ n \xrightarrow[P]{} n-1 \xrightarrow[D]{} k \xrightarrow[S|S]{} k+2 & \text{if } n \neq 0 \end{cases}$$

Now, 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: $1_{nzr}=1_{zr}^{\sim}\cap 1_N$ (since $1_{zr}^{\sim}\cap 1_N=\{<n,n>\in N\times N/n\neq 0\}$). So, one can express this behaviour in terms of its input-output pairs as follows:

$$\begin{cases} \text{if } n = 0 : n \xrightarrow{D} m \Leftrightarrow n \xrightarrow{1_{zr}} m \\ \text{if } n \neq 0 : n \xrightarrow[D]{} m \Leftrightarrow n \xrightarrow[P]{} i \xrightarrow[D]{} j \xrightarrow[S|S]{} m \end{cases}$$

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

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

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

4.2 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 [Partsch' 90].

Here, we use a binary relation $Rev \subseteq W \times W$ on the universe W of words, where $<u,v> \in Rev$ iff v is the reversal of u. So, the set of palindromes can be described as $Pal = \{w \in W / <w,w> \in Rev\}$.

The set Pal of palindromes can also be described by a "filter" for palindromes $1_{Pal} = \{<w,w> \in W \times W / w \in Pal\}$. From filter 1_{Pal} one can recover the set Pal of palindromes, via $Pal = \{w \in W / <w,w> \in 1_{Pal}\}$.

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

$$u \xrightarrow{1_{Pal}} v \Leftrightarrow \begin{bmatrix} u \xrightarrow{Rev} v \\ \text{and} \\ u \xrightarrow[1_W]{} v \end{bmatrix} \qquad 1_{Pal} = \begin{pmatrix} Rev \\ \cap \\ 1_W \end{pmatrix}$$

We can thus formulate binary relation $1_{Pal} \subseteq W \times W$ as a relational term: $1_{Pal} = Rev \cap 1_W$.

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 the filter, a relational term for the characteristic function of the set Pal of palindromes. Namely, $\chi_{Pal} = (1_{Pal}|Tr) \cup [(1_{Pal}^{\sim} \cap 1_W)|Fl]$.

$$\chi_{Pal} = \begin{pmatrix} 1_{Pal}|Tr \\ \cup \\ (1_{Pal}^{\sim} \cap 1_W)|Fl \end{pmatrix}$$

This example illustrates how one can express input-output specifications by means of relational terms.

From such relational input-output specifications we can derive, by algebraic manipulations, some (recursive and iterative) programs [Partsch '90] for palindromes [Haeberer & Veloso '91].

4.3 Relational Data Type Presentations

A specification describes properties of the objects involved. We have seen an example of a relational specification for a program. In a similar spirit, one can also specify data types by presenting their properties in a relational style [Berghammer '91].

For a simple example, consider the data type of the natural numbers used in the previous example of a program for computing the double. It has operations successor s and predecessor p as well as constant zero.

We wish to express some properties of this data type in a relational manner. For this purpose we may proceed in a manner similar to the one implicit in the preceding example. Namely, for the operations we use their graphs: relations S and P. For the constant 0 we use the constant relation $Z := N \times \{0\} = \{<n,0> \in N \times N / n \in N\}$. From this constant relation Z we can obtain the partial identity $1_{zr} = \{<0,0>\}$ (since $1_{zr} = (Z^T|Z) \cap 1_N$). So, we also have the partial identity $1_{nzr}=1_{zr}\tilde{}\cap 1_N\tilde{}$ to represent the non-zero naturals).

We can then replace an expression like p(m)=n by $<m,n>\in P$; so p(s(n))=n becomes equivalent to $<n,n>\in S|P$. Thus, we can express functionality of relations (e . g. $P^T|P\subseteq 1_N$ for n'=n" whenever <n,n'>,<n,n">$\in$P).

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:	$S|S^T\subseteq 1_N$,
$\forall x\neg s(s(x)=x$:	$S|S\cap 1_N=\varnothing$,
$\forall y[y=0\vee\exists x y=s(x)]$:	$1_{nzr}\subseteq S^T|S$ or $1_N\subseteq 1_{zr}\cup(S^T|S)$

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

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

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

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

4.4 Expressing Programming Methods

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

We first examine a sorting algorithm based on this method, namely mergesort. A (recursive) formulation for mergesort is as follows:

$$\mathrm{srt}(s) = s \qquad \text{if } \lg(s)\leq 1$$

$$\mathrm{srt}(s) = \mathrm{merge}\begin{pmatrix} \mathrm{srt(frst(s))} \\ , \\ \mathrm{srt(scnd(s))}\end{pmatrix} \qquad \text{otherwise}$$

where frst(s) and scnd(s) provide, respectively, the first and second halves of sequence s, which are (recursively) sorted and merged into a sorted version of the original input sequence [Aho et al. '75; Broy '83].

This algorithm is based on the following idea. Given an input sequence:
if it is simple (lg(s)≤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 [Aho et al. '75; Horowitz & Sahni '78]. It can be presented as follows:

$$\mathrm{D\_C(d)} = \begin{cases} \mathrm{Drct(d)} & \text{if Smpl(d)} \\ \mathrm{Rcmbn}\begin{pmatrix}\mathrm{D\_C(Splt_1(d))} \\ \mathrm{D\_C(Splt_2(d))}\end{pmatrix} & \text{otherwise}\end{cases}$$

Now, let us examine its behaviour in terms of input-output pairs. It can be described as follows:

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

We can notice two related features of the above 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, this situation can be represented as:

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

We see that this situation involves two related points:
- the universe U should have pairs $<u_1,u_2>$ of (some of) its elements;
- we need a new binary operation $/\!\!/$ on relations
  corresponding to parallel execution on components $\|$.

We shall examine them shortly. For the moment, let us just proceed with this example.
With the parallel product $/\!\!/$ operation, we can write an expression for binary relation D_C:

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

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 should 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.

4.5 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.

We wish to express some properties of this many-sorted data type in a relational manner. For this purpose we may 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 $1_{nl}=\{\langle k,k\rangle / k\in null\}$ (so, we have partial identity $1_{nnl}:=1_{Lst}\cap 1_{nl}^{\sim}$ to represent the non-null lists).

We can then replace an expression like cons(e,l)=k by $\langle\langle e,l\rangle,k\rangle\in Cns$; so head(cons(e,l))=e becomes equivalent to $\langle\langle e,l\rangle,l\rangle\in Cns|Hd$. Thus, we can provide information concerning argument and result sorts, e. g. $Hd\subseteq Lst\times Elm$ by $(Hd|Hd^T)\cap 1_U\subseteq 1_{Lst}$ and $(Hd^T|Hd)\cap 1_U\subseteq 1_{Elm}$.

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

$$\begin{tikzcd} k \arrow[r, "Cns^T"] & \begin{pmatrix} e \\ , \\ l \end{pmatrix} \arrow[r, "Cns"] & k \end{tikzcd} \quad : \; 1_{nnl} \subseteq (Cns^T)\,|Cns$$

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

$$\begin{tikzcd} k \arrow[r, "1_{nnl}"] & k \arrow[r, "2_U"] & \begin{pmatrix} k \\ , \\ k \end{pmatrix} \arrow[r, shift left=2, "Hd"] \arrow[r, shift right=2, "Tl"'] \arrow[r, phantom, "/\!\!/"] & \begin{pmatrix} e \\ , \\ l \end{pmatrix} \arrow[r, "Cns"] & k \end{tikzcd}$$

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

5. A RELATIONAL PROGRAMMING FRAMEWORK

The preceding examples and comments indicate that the relational framework, with suitable extensions, appears to be able to express (some):

- programs (as illustrated by double and mergesort);
- input-output specifications (as illustrated by palindrome);
- data type specifications (as illustrated by naturals and lists);
- programming methods (as illustrated by divide-and-conquer).

We have mentioned that (some) program derivation strategies can be formulated in a similar spirit. Furthermore, as also mentioned, 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].

These remarks stress the desirability of suitably extending the relational approach aiming at a framework appropriate for expressing programming ideas and reasoning about them. Some desiderata for this extended framework are wide expressive, deductive and transformational powers; we would like to:

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

Such a framework would support a wide-spectrum language and calculus for program derivation. Within it, we would be able to:

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

6. OTHER ASPECTS

We now comment on some other aspects, of importance in the context of a framework for program development. We shall address some of these aspects in forthcoming reports.

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. This can be settled by the identification of a set of symbols that deserve being called "algorithmic". We can 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 show that we have sufficient expressive power.

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

The structural extension was 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 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 $_{V|}*:V\rightarrow U$.)

We shall now briefly comment on expressing properties.

It is well known that information concerning domain and range, as well as functionality, injectivity and surjectivity can be expressed in relational terms. Also, the introductory examples have indicated that one can express some first-order properties in the extended relational framework.

In fact, 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 can be (effectively) converted to a closed (partial identity) term "with the same extension". Thus, the partial identity of each m-ary relation definable by a first-order formula is binary relation that is definable by a closed extended relational term.

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. We can match the repertoire of predicates, operations and constants of such an applied first-order logical language with a repertoire of relation constants, as indicated in the introductory examples.

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.

The expressivity of the extended relational languages allows one to translate back and forth between first-order formulae and closed terms. We shall now briefly comment on reasoning about properties expressed in this manner.

We can then move back and forth between first-order formulae and 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.

As a consequence of the Representation Theorem [Frias et al. '93, '95], 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. Consider a set $\Sigma \cup \{\tau\}$ of first-order sentences and corresponding equations between closed terms $\Sigma^* \cup \{\tau^*\}$. Then, sentence τ is a logical consequence of set Σ iff equation σ^* can be derived by equational reasoning within AFC from Σ^*.

In this sense, the Algebraic Fork Calculus can be regarded as relational counterpart in equational form of first-order logic. So, the Algebraic Fork Calculus may be said to provide a finitary equational algebraic formulation of first-order logic. It may be regarded as being a first-order analogue of what Boolean algebras are for sentential logic.

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.

It is well-known that one can faithfully reduce many-sorted first-order languages to their unsorted versions, by employing relativisation predicates that are intended to characterise the sorts. In terms of models, the universe of the unsorted structure is regarded as the union of all sorts, which can be recovered by means of the relativisation predicates provided. (See, e. g. [Enderton '72; van Dalen '89].)

In the relational setting, we may proceed as suggested in the introductory examples. The universe U is to consist of the sorted sets U_k's as well as of their ordered pairs. For each sort s_k we use the partial identity 1_k, which characterises it. For a predicate p we use the partial identity 1_p over its extension. For an operation f we use its graph. For a constant c we use the constant relation $C{:=}U\times\{c\}$.

From this relational presentation one can recover the original structure.

In this manner, we can mimic many-sorted first-order reasoning by equational reasoning within our extended relational calculus.

7. CONCLUSIONS

We have presented an extended relational framework for program development. This extension is motivated by the need to express some simple programming ideas. We have, accordingly, introduced the question of its adequacy for programming.

We have first argued that the familiar apparatus of binary relations must be extended to be adequate for program development. We have then suggested that an appropriate extension can be obtained by considering relations on structured universes together with new operations.

We have begun in section 2 with some motivation about program construction and derivation. In section 3 we have recalled some basic operations on sets and relations and briefly reviewed the role of partial identities for representing sets as relations. We have then presented in section 4 a series of examples, intended to illustrate how one can express programming ideas in a relational form and to indicate the need for an extension. We have also outlined some desiderata for a wide-spectrum framework for program development in section 5. Some other aspects, of importance in the context of a framework for program development, have been briefly commented upon in section 6.

The adequacy of this extended relational framework for program derivation, with respect to more pragmatic aspects, 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].

We can also argue that this extension does provide an adequate framework for programming, because we can select an algorithmic part of the extension, which turns out to have the appropriate expressive power. This involves the establishment of two inclusions. On the one hand, our algorithmic part expresses only computing-like behaviours. On the other hand, every program, even a nondeterministic one, can be expressed in this algorithmic part.

An intuitive explanation for the computing-like nature of the algorithmic part can be provided. Namely, a programming language correspondence substantiates the feeling of adequacy for programming while giving a more pragmatic support for it.

As a criterion for the selection of the algorithmic symbols, we can consider a classification based mainly on effective properties and their preservation. This classification also provides a theoretical explanation, relying on effectiveness, for their computing-like nature: all of them present or preserve effectively enumerable behaviours.

This paper is the first one of a series of papers addressing the question of adequacy of a fork relational framework for program development. Subsequent papers will concentrate on other 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.

Backhouse, R. C., Bruin P. and Malcom, G. (1990) A Relational Theory of Types. 41st Meeting of the IFIP Working Group 2.1 "Programming Languages and Calculi", Document 638-BUR-5, Chester.

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. Assoc. Comput. Mach.* **21**, 613-641.

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

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 Schmidt, G. (1989) Symmetric quotients and domain constructors. *Inform. Process. Letters*, 33, 163-168.

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 Guiho, G. (eds) *Automatic Program Construction*, Reidel, Dordrecht.

Broy, M. and Pepper, P. (1981) Program development as a formal activity. *IEEE Trans. Software Engin.*, **7**(1), 14-22.

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.

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., Baum, G. A., Haeberer, A. M. and Veloso, P. A. S. (1995) Fork algebras are representable. *Bull. of Sect. of Logic*, Univ. Lodz, **24**(2), 64-75.

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.

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.

Halmos, P. R. (1963) *Lectures on Boolean Algebra*. D. van Nostrand, Princeton, NJ.

Harel, D. (1979) *First-order Dynamic Logic*. Springer-Verlag, Berlin.

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.

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.

Mili, A. (1983) A relational approach to the design of deterministic programs. *Acta Informatica* 20, 315-329.

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

Partsch, H. A. (1990) *Specification and Transformation of Programs*. Springer-Verlag, Berlin.

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.

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. 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.