PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 42/96

Fork Relational Frameworks on Structured Universes

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° 42/96 ISSN 0103-9741
Editor: Carlos J. P. Lucena November 1996

Fork Relational Frameworks on Structured Universes *

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

Fork Relational Frameworks on Structured Universes

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

PUCRioInf MCC 42/96

Abstract

We examine a fork relational framework for program development. We introduce some structural operations and constants, consider some interconnections among them, which provide alternative bases for the extension, and examine some properties of this fork relational apparatus. In a previous report, we have argued that the familiar apparatus of binary relations must be extended to be adequate for programming; we have also suggested that an appropriate extension could be obtained by considering relations on structured universes together with new operations. In this report we examine more closely the nature of this fork relational framework.

Key words: Formal specifications, relational calculi, relational algebras, fork algebras, structured universe, structural operations.

Resumo

Examina-se um ambiente relacional estendido por fork para desenvolvimento de programas. São introduzidas algumas operações estruturais, consideradas algumas interconexões entre elas, as quais fornecem bases alternativas para a extensão, e examinadas algumas propriedades deste ambiente relacional estendido por fork. Em um trabalho anterior, argumentou-se que o aparato usual de relações binárias precisa ser estendido para ser adequado para programação; suggeriu-se também que uma extensão apropriada poderia ser obtida considerando-se relações sobre um universo estruturado com novas operações. Neste trabalho examina-se mais de perto a natureza deste aparato relacional estendido por fork.

Palavras chave: Especificações formais, cálculos relacionais, álgebras relacionais, álgebras de fork, universo estruturado, operações estruturais.

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 second of a series of reports addressing the question of adequacy of a fork relational framework for program development. Other reports concentrate on distinct aspects of this question, such as:
A fork relational framework for program development;
Effectiveness and programming language aspects in fork relations;
Algorithmic fork relations and programs.

CONTENTS

1. INTRODUCTION

We examine a fork relational framework for program development. We introduce some structural operations and constants, consider some alternative bases for the extension and examine some properties of this fork relational apparatus.

In a previous report, we have argued that the familiar apparatus of binary relations must be extended to be appropriate for programming; we have also suggested that an appropriate extension can be obtained by considering relations on structured universes together with new operations.

In this report we examine more closely the nature of this fork relational framework. We introduce some structural operations and constants, consider some interconnections among them, which provide alternative bases for the extension, and examine some properties of this fork relational apparatus.

We begin in section 2 with some motivation about program construction and derivation and by recalling some basic operations on relations. Next, we present of 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 also outline some desiderata for a wide-spectrum framework for program development. In section 4 we take a closer look at the nature of the proposed extension and the new operations provided. Some other aspects, of importance in the context of a framework for program development, are briefly commented upon in section 5. Finally, section 6 presents some concluding remarks and comments on related aspects.

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

2. PRELIMINARIES

We begin with some preparatory material. First we briefly present some basic ideas about program construction, which motivate a relational approach to programming. We then recall some basic operations on relations as sets of ordered pairs.

2.1 MOTIVATION: PROGRAM CONSTRUCTION AND DERIVATION

Program construction refers to the process of obtaining a program from a specification of its input-output behaviour in a methodical manner. An interesting version is program derivation, where the emphasis resides 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 appropriate 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 much 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 having to examine traces corresponding to particular inputs [Backus '78].

These considerations suggest a formalism with an algebraic flavour, based mainly on terms and equations. We would then reason about properties in an equational manner and transform expressions in an algebraic fashion. A repertoire of transformations based on theorems of the formalism can provide a sound basis for such transformations [Bauer & Wössner '82; Sintzoff '85; Partsch '90].

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

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,

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 easily one can handle it.

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.

2.2 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) $r^T:=\{<v,u> \in U \times U / <u,v> \in r\}$ 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].)

B. Partial Identities

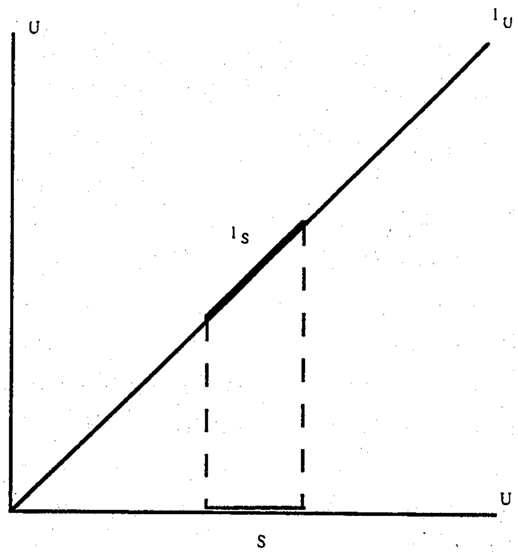
Let us now examine partial identities [Haeberer & Veloso '91].

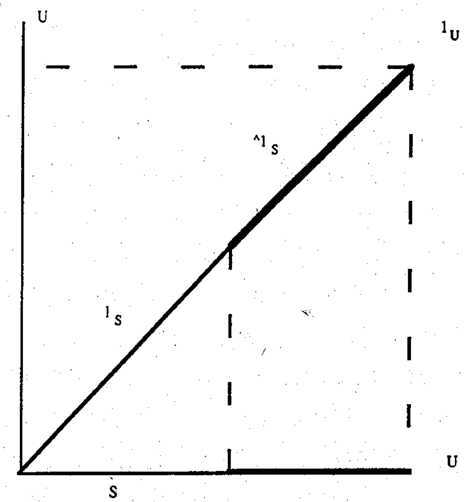
Partial identities are devices for representing sets as relations, they are useful tools in expressing programs and specifications in a relational manner. A partial identity behaves as a "filter" on its underlying set; as such it can also be used to obtain tests for membership as well as for restricting 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 $S=\{u\in U/<u,u>\in 1_S\}$.

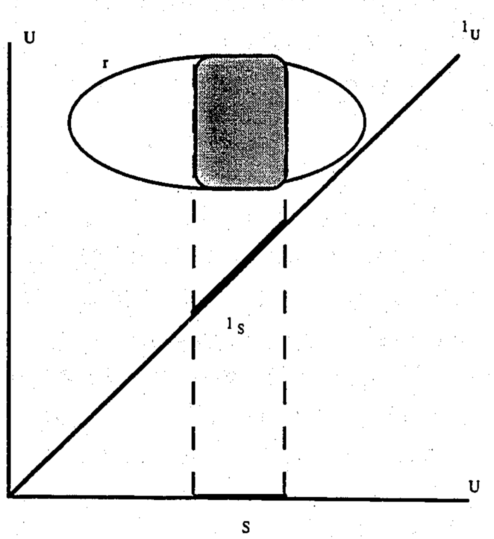
Partial identity 1_S behaves as a "filter" on S: for $u\in S$ it behaves identically ($<u,v>\in 1_S$ iff $v=u$), whereas for $u\notin S$ it provides no output.

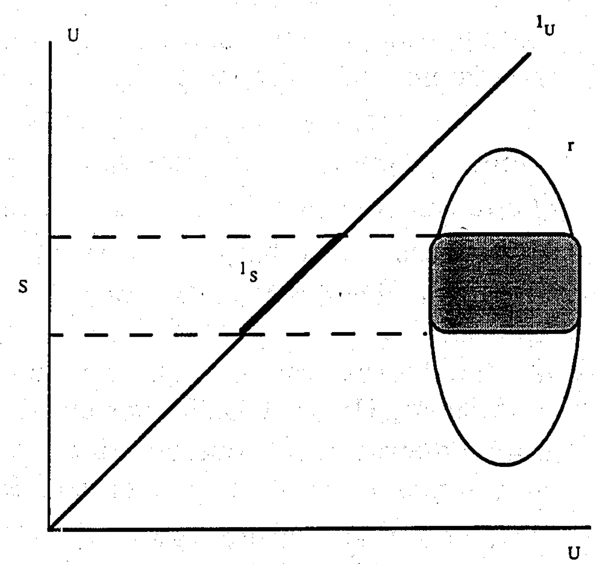
As a test for non-membership we can use the complement with respect to the identity: $^\wedge 1_S := 1_U \cap 1_S^{\sim}$, for $^\wedge 1_S=\{<u,u>\in U\times U/u\notin S\}$.





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





3. RELATIONS AND PROGRAMMING

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

3.1 Expressing Programming Ideas

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.

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: $1_{nzr}=1_{zr}^{\sim}\cap 1_N$. 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 $D\subseteq \mathbf{N}\times\mathbf{N}$ 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: $D=1_{zr}\cup(1_{nzr}|P|D|S|S)$.

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

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 $<u,v>\in Rev$ iff v is the reversal of u.

The set Pal of palindromes can 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 $Pal=\{w\in W/<w,w>\in 1_{Pal}\}$.

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].

C. Relational Data Type Presentations

A specification describes properties of the objects involved. We have seen an example of a relational specification for a program. Similarly, one can also specify data types by presenting their properties in a relational style.

For a simple example, consider the data type of the natural numbers used in the previous example of a program for 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 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 $Z:=\mathbf{N}\times\{0\}=\{<n,0>\in\mathbf{N}\times\mathbf{N}/n\in\mathbf{N}\}$. From this constant relation Z we can obtain the partial identity $1_{zr}=\{<0,0>\}$. So, we also have the partial identity $1_{nzr}=1_{zr}{}^{\sim}\cap 1_N{}^{\sim}$ to represent 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:	$S|S^T\subseteq 1_N$,
$\forall x\neg s(s(x)=x$:	$S|S\cap 1_N=\varnothing$,
$\forall y[y=0\vee\exists x y=s(x)]$:	$1_{nzr}\subseteq S^T|S$ or $1_N\subseteq 1_{zr}\cup(S^T|S)$

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

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

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

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

D. 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:

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

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

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].

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:

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

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

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

We can notice two related features of 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:

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

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:

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

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 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.

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.

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 $1_{nl}=\{<k,k>/k\in null\}$ (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 $(\forall k{:}Lst)[null(k)\vee(\exists e{:}Elm)(\exists l{:}Lst)k=cons(e,l)]$ in relational terms by one of the alternative formulations: $1_{nnl} \subseteq (Cns^T)|Cns$ or $1_{Lst} \subseteq 1_{nl} \cup [(Cns^T)|Cns]$.

$$k \xrightarrow{Cns^T} \begin{pmatrix} e \\ , \\ l \end{pmatrix} \xrightarrow{Cns} k \; : \; 1_{nnl} \subseteq (Cns^T)\,|Cns$$

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

$$k \xrightarrow{1_{nnl}} k \xrightarrow{2_U} \begin{pmatrix} k \\ , \\ k \end{pmatrix} \begin{matrix} \xrightarrow{Hd} \\ /\!\!/ \\ \xrightarrow[Tl]{} \end{matrix} \begin{pmatrix} e \\ , \\ l \end{pmatrix} \xrightarrow{Cns} k$$

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

3.2 PROGRAMMING WITH RELATIONS

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, 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.

4. STRUCTURED UNIVERSE AND EXTENDED RELATIONS

The examples and comments in the preceding section suggest that the relational framework should be extended in order to express programming ideas. We shall now take a closer look at such extended relational framework.

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 shall use the name *structured universe* for such a set U closed under cartesian product: $U \times U \subseteq U$.

4.1 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 introduce them and examine some connections among them.

A. Extended Operations and Constants

As mentioned, we can have new structural operations and constants. We have already encountered the parallel product operation ⫽ and the duplication constant 2_U.

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

$$\begin{pmatrix} u_1 \\ , \\ u_2 \end{pmatrix} \xrightarrow{r /\!\!/ s} \begin{pmatrix} v_1 \\ , \\ v_2 \end{pmatrix} \Leftrightarrow \left\langle \begin{matrix} u_1 \xrightarrow{r} v_1 \\ \& \\ u_2 \xrightarrow[s]{} v_2 \end{matrix} \right\rangle$$

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.

$$2_U := \{<u,<v,w>> \in U \times U / u=v=w\} \qquad u \xrightarrow{2_U} \begin{pmatrix} u \\ , \\ u \end{pmatrix}$$

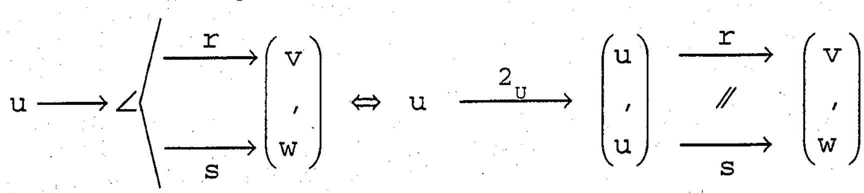
Notice that constant relation 2_U is a total functional relation.

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

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

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 ∠ is monotonic with respect to inclusion ⊆ and preserves functionality of relations; also Dom[r∠s]=Dom[r]∩Dom[s].
This fork operation ∠ can be defined from the above two structural operations, since $r \angle s = 2_U | (r /\!\!/ s)$.



B. Alternative Bases for the Extension

In general, the universal relation V may be an equivalence relation V⊆U×U on U. We then wish to have both V⁄⁄V and V∠V included in V. For this purpose we require V to be closed under pair formation: <x,<x,y>>∈V whenever <x,y>∈V. We shall use the names:

closed equivalence for an equivalence relation V⊆U×U on U that is closed under pair formation (<x,<x,y>>∈V whenever <x,y>∈V), and

structural equivalence for a structured universe U equipped with a closed equivalence relation V⊆U×U on U.

For such a closed equivalence V, its powerset ℘(V)={r⊆U×U/r⊆V} is closed under both operations parallel product ⁄⁄ and fork ∠ (in view of their monotonicity).

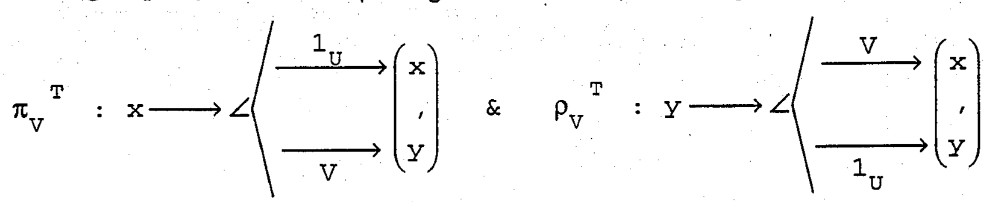
We then have both $2_U, 2_U^T \subseteq V$. So, we can express these constants as:

duplication $2_U = \{\langle u,\langle v,w\rangle\rangle \in V / u=v=w\}$ and

equality sieve $2_U^T = \{\langle\langle v,w\rangle,u\rangle \in V / u=v=w\}$.

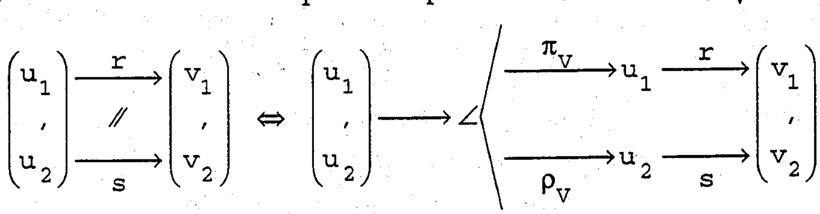
Clearly, duplication constant 2_U can be derived from operation fork ∠, via $2_U = 1_U \angle 1_U$.

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

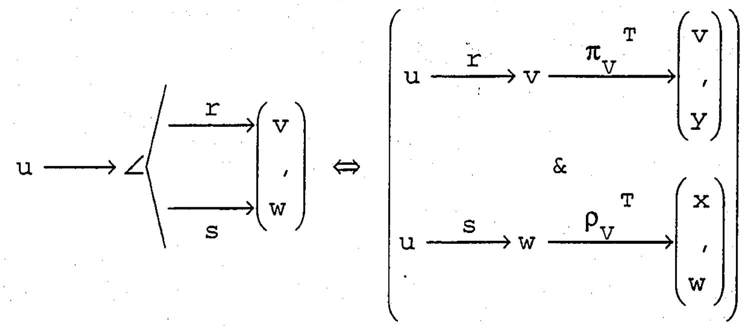


We notice that the constant projections $\pi_V = \{\langle\langle x,y\rangle,x\rangle \in U\times U / \langle x,y\rangle \in V\}$ and $\rho_V = \{\langle\langle x,y\rangle,y\rangle \in U\times U / \langle x,y\rangle \in V\}$ are partial functions with domain U×U⊆U.

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



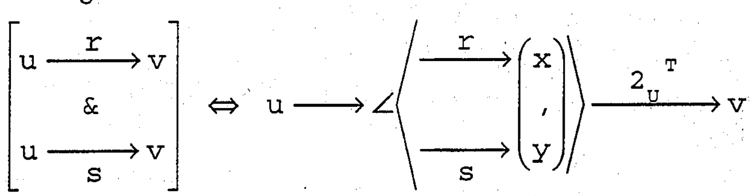
Also notice that from the transposed projections $\pi_V{}^T=1_U\angle V$ and $\rho_V{}^T=V\angle 1_U$ one can recover fork $\angle$, via $r\angle s=(r|\pi_V{}^T)\cap(s|\rho_V{}^T)$.



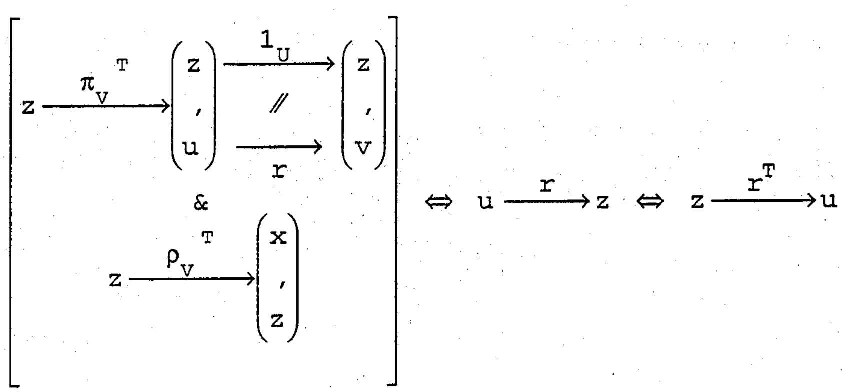
Therefore, the sets $\{\angle\}$, $\{/\!\!/,2_U\}$ and $\{\pi_V,\rho_V\}$ are all relationally interderivable, and either one can be used as a basis for our extension.

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.

4.2 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 give some more details concerning the representation of subsets by partial identities.

A. Properties of the Extended Operations and Constants

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

With respect to totality, we see 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 $^\wedge p=1_U\cap p^\sim$) 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 considering 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 $p,q\in\wp(1_U)$ $r_p[]_q s:=r_p\oplus_q s$; so we have $r_p[]_q s=\{<u,v>\in U\times U/(<u,u>\in p\wedge<u,v>\in r)\vee(<u,u>\in q\wedge<u,v>\in s)\}$.

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

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 $t\subseteq 1_U$.

5. OTHER ASPECTS

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

5.1 ADEQUACY FOR PROGRAMMING

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 adequate expressive power. This can be settled by identifying 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.

5.2 CARTESIAN PRODUCT AND PAIR CODING

We now 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 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 ${}_{V|}*: V \rightarrow U$.) The preceding considerations carry over to this more flexible approach based on the concept of pair coding.

5.3 Expressing Properties

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 φ can be (effectively) converted to a closed (partial identity) term $\varphi^{\#}$ "with the same extension". So, the partial identity of each m-ary relation definable by a first-order formula φ 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.

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.

In this sense, these extended relational languages can be regarded as relational counterparts of first-order logical languages [Veloso & Haeberer '91].

5.4 Reasoning about Properties

We shall briefly comment on reasoning about properties.

Consider a matching between a repertoire of predicates, operations and a corresponding relational constants. 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.

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. 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.

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].

As an example, consider a program (segment) s on a data type. Let Σ be the specification of this data type by first-order sentences, which can be formulated relationally as equations between closed terms Σ^*. Let $\Gamma(s)$ be the first-order specification of the program behaviour, which can be formulated relationally as equations between closed terms $\Gamma^*(s\#)$. Consider also a first-order sentence $\tau(p)$ expressing a property of program s, which has as relational formulation an equation $\tau^*(s\#)$. We then have, in first-order logic $\Sigma\cup\Gamma(s)\models\tau(s)$ iff within AFC $\tau^*(s\#)$ can be equationally derived from $\Sigma^*\cup\Gamma^*(s\#)$.

For instance, consider a specification of input-output behaviour given by precondition $\varphi(x)$ and postcondition $\psi(x,y)$. This input-output specification can be formulated relationally by partial identities $\varphi^\#$ and $\psi^\#$, and from the latter we obtain the input-output relation $\psi^{\rightarrow}:=\pi_V{}^T|\psi^\#|\rho_V$, Now, consider an extended relational term r (expressing the input-output behaviour of program (segment) s). Partial correctness (i. e. $\varphi(x)\wedge s(x,y)\rightarrow\psi(x,y)$) is expressed by the inclusion $\varphi^\#|r\subseteq\psi^{\rightarrow}$. Termination (i. e. $\varphi(x)\rightarrow\exists y s(x,y)$) can also be handled in this framework: it is expressed by the inclusion $\varphi^\#\subseteq\underline{r}$ (as $\underline{r}=1_{Dom[r]}$). For a simple example, consider a relation r on the naturals and a total precondition $\varphi(x)$ holding for all naturals (as in the doubling relation in the introductory example). To guarantee termination in such a case, it suffices to establish the basis $1_{zr} \subseteq r | r^T$ and inductive step $\underline{r} | S \subseteq S | \underline{r}$. (See, e. g. [Veloso & Haeberer '94].)

5.5 MANY-SORTED FRAMEWORK

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 in the sense that $1_k = \{<u,u> \in U / k \in U_k\}$. For a predicate p, over sorts $s_1, \ldots, s_m$, we use the partial identity $1_p = \{<u,u> \in U_1 \times \ldots \times U_m / u \in p\}$. For an operation f, from sorts $s_1 .. s_n$ to sort s_0, we use its graph $F := \{<u,v> \in (U_1 \times \ldots \times U_n) \times U_0 / f(u) = v\}$. For a constant c, over sort s_k, we use the constant relation $C := U \times \{c\}$.

From this relational presentation one can recover the original structure.

In this relational presentation, we can express equality $=_k$ over sort s_k by the partial identity $2_U | 2_U^T | 1_k = \{<u,v> \in U_k \times U_k / u = v\}$. We also have, for each list $<s_j, \ldots, s_l>$ of sorts, the partial identity ${}_j1_l := 1_j /\!\!/ .. /\!\!/ 1_l$ describing the product ${}_jU_l := U_j \times \ldots \times U_l$, with which we can express information on domains and ranges. We can also express functionality of an operation graph F and the fact that C is a total constant function. The relational representation for terms is obtained inductively: the graph of $f(t_1, \ldots, t_n)$ being $(T_1 \angle \ldots \angle T_n) | F$.

Thus, we can see that expressivity carries over to the many-sorted setting: each formula $\varphi(x_1, \ldots, x_m)$, with $x_1 : s_1, \ldots, x_m : s_m$ being translated to partial identity $\varphi\# \subseteq 1_1 /\!\!/ .. /\!\!/ 1_m$.

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

6. CONCLUSIONS

In a previous report, we have argued that the familiar apparatus of binary relations must be extended to be appropriate for programming; we have also suggested that an appropriate extension can be obtained by considering relations on structured universes together with new operations.

In this report we have examined more closely the nature of this fork relational framework. We have introduced some structural operations and constants, considered some interconnections among them, which provide alternative bases for the extension, and examined some properties of this fork relational apparatus.

We have started in section 2 with some motivation about program construction and derivation and by recalling some basic operations on relations. In section 3, we have presented of 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. Next, we have taken a closer look at the nature of the proposed extension and the new operations provided in section 4. Some other aspects, of importance in the context of a framework for program development, have been briefly commented upon in section 5.

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 indeed provides an adequate framework for programming, because we can select an algorithmic part of the extension, which turns out to have the appropriate expressive power.

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 report is the second one of a series of reports addressing the question of adequacy of a fork relational framework for program development. Other reports focus 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.

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 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.

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., 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.

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.

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.

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].

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.