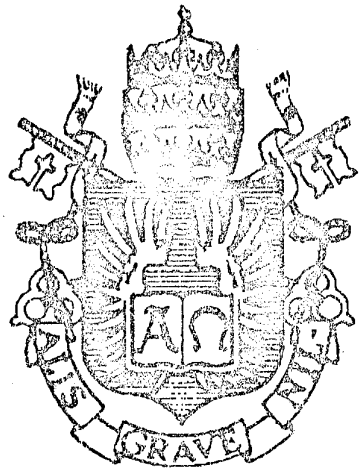

PUC

Monografias em Ciência da Computação
nº 19/92

On Formal Program Construction within an Extended Calculus for Binary Relations

Paulo A. S. Veloso
Armando M. Haeberer
Gabriel Baum

Departamento de Informática

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

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 19/92
Editor: Carlos J. P. Lucena Maio, 1992

On Formal Program Construction within an Extended Calculus for Binary Relations *

Paulo A. S. Veloso
Armando M. Haeberer
Gabriel Baum #

Depto. Informática, Universidad Nacional de La Plata, Argentina

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

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
22454970 — 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
techrep@inf.puc-rio.br (for publications only)

ABSTRACT

Research reported herein is part of an on-going effort in using relational formalisms for formal program construction. This paper emphasizes three main points: first, the distinction among specification and programming languages and derivation formalisms (as illustrated by the role of intersection); second, that the construction of a theory of the application, and its expression, involved in formal program development can, and perhaps should, be distributed and intertwined along the process; third, how derivation insights and rationales can be captured within a formalism with the goal of reusing them and automating their application.

Our formalism is based on an extended version of Tarski's calculus of binary relations, amounting to the addition of relativized identities and a couple of new operations, rendered natural because the universe is now structured. This extended calculus has the expressive power of first-order logic, which makes it appropriate for expressing, and reasoning about, programs, as well as strategies and design decisions. This gives a truly coherent framework, based on the single unifying concept of input-output relations over structured universes, covering the entire derivation spectrum, from specifications to programs, both of which viewed as relational terms.

Key words:

Formal program construction, formal methods, program derivation calculi, calculus of binary relations, specification languages, programming languages, derivation formalisms, derivation methods and strategies.

RESUMO

Os resultados aqui apresentados são parte de um esforço continuado no uso de formalismos relacionais para construção formal de programas. Este trabalho enfatiza três pontos principais: primeiro, a distinção entre linguagens de especificação e de programação e formalismos de derivação (conforme ilustrado pelo papel da interseção); segundo, que a construção de uma teoria da aplicação, e sua expressão, ocorrendo no desenvolvimento formal de programas, pode, e talvez deva, ser distribuída ao longo do processo; terceiro, como idéias e estratégias de derivação podem ser captadas em um formalismo, a fim de reutilizá-las e automatizar sua aplicação.
Nosso formalismo se baseia em uma versão estendida do cálculo de relações binárias de Tarski, envolvendo o acréscimo de identidades relativizadas e algumas novas operações, que são naturais uma vez que o universo agora é estruturado. Nosso cálculo estendido tem o poder expressivo da lógica de primeira ordem, o que o torna apropriado para expressar programas, estratégias e decisões de projeto, bem como para raciocinar sobre eles. Isso fornece um arcabouço realmente coerente, baseado no conceito unificador de relações de entrada-saída sobre universos estruturados, cobrindo todo o espectro de derivação, desde especificações até programas, ambos encarados como termos relacionais.

Palavras chave:

Construção formal de programas, métodos formais, cálculos de derivação de programas, cálculo de relações binárias, linguagens de especificação, linguagens de programação, formalismos de derivação, métodos e estratégias de derivação.

CONTENTS

On Formal Program Construction within an Extended Calculus of Binary Relations

Paulo A. S.Veloso, Armando M. Haeberer,
Depto. Informática, Pont. Univ. Católica do Rio de Janeiro
Rua Mq. de S. Vicente 225; Rio de Janeiro, RJ 22453; Brasil

Gabriel Baum
Depto. Informática, Universidad Nacional de La Plata
La Plata, Prov. de Buenos Aires, República Argentina

1. INTRODUCTION

Research reported herein is part of an on-going effort in using relational formalisms for formal program construction. Our formalism is based on an extended version of Tarski's calculus of binary relations. This extension amounts to adding relativized identities and to structuring the universe, which makes natural the addition of a couple of new operations.

Our extended calculus has been shown to have the expressive power of first-order logic. It is also appropriate for expressing, and reasoning about, programs, as well as strategies and design decisions. So, our framework, based on the single unifying concept of binary relations over structured universes, is a truly coherent tool covering the entire derivation spectrum, from specifications to programs, which are viewed as relational terms.

This paper emphasizes three main points. First, the distinction between specification and programming languages, on the one hand, and between these and derivation formalisms, on the other hand. Second, that formal program development involves the construction of a theory of the application, and its expression within a reasonable formalism, and both can, and perhaps should, be distributed and intertwined along the process. Third, how derivation insights and rationales can be captured within our formalism with the goal of providing machine support for clerical symbolic manipulations.

As regards the first point, it is sometimes suggested that operations, like intersection of relations should not be present in a programming language, because it tends to yield extremely inefficient programs. But, should it be eliminated from specification and derivation languages, which have distinct objectives, as well? We shall argue that the

answer is no. As an illustration, we show how to derive an efficient program from a specification, used as an argument for banning intersection [Hoa86].

We illustrate the second point with some derivations, which show how the processes of a formal specification and program construction help each other. We also develop formal machinery for these purposes. Our formalism is especially appropriate for this task, since we can represent sets by relativized identities and also internalize relations by coding input-output pairs as objects of the universe.

As for the third point, let us notice that formalisms offer the advantage of precision and trustworthy manipulations. But, and perhaps more important, they also enable the clear separation between the creative parts and mere manipulations. Moreover, carying out such manipulations by hand is both boring and error prone. So, we indicate how insights and rationales can be captured within our formalism with the goal of reusing them and automating their application.

2. SPECIFICATIONS, PROGRAMS AND DERIVATION

It is important to bear in mind what is involved in deriving a program from a specification. In a nutshell, the purpose of a derivation is transforming a, possibly non-executable, specification into an executable program, ideally an efficient one. We shall now expand this by some simple remarks, which are meant to be introductory and to be complemented by further, and more substantial, observations later on.

2.1. PROGRAM DERIVATION

As a running example to illustrate and clarify some ideas, we shall often use the case of deriving a program for an *application* like palindrome or list reversal. These are problems about lists, and perhaps Booleans; so these data types are given as underlying the specification of the task; let us call them *basic data types*. The data type lists comes equipped with some basic *sorts*, *operations* and *predicates*. We shall call them (*application*) *specification primitives*, because, being already well understood, they are available as building blocks for specifying other sorts, operations and predicates. It is useful, however, to draw a distinction connected to the idea of hidden operations [Gut78]. Some specification primitives, like **head** and **tail**, are assumed to be already implemented; let us call them (*application*) *programming primitives,* because they are available as building blocks for writing executable programs. Thus, for instance, an operation occurs for checking whether an element occurs in a list might be a specification primitive, but not a programming primitive. Let us use *target data types* for the restriction of the basic data types to their programming primitives.

Of course, the starting point of the derivation process is a specification of the problem to be solved. In specifying such a task, one relies on a *specification formalism*, which provides some basic general *specification constructs*. For instance, first-order logic provides propositional connectives and quantifiers. The specialization of the specification formalism to the language of the basic data types for the problem at hand is the *application specification language*. For our running example, this may be the first-order language of lists with Booleans. The *problem specification* consists of formulas in this application specification language.

At the other end of the spectrum one has an executable program. As in he case of the specification, this *program* is a syntactic object of the *target language*, the latter being the specialization of a general *programming language*, which provides some basic general programming constructs (like conditional constructs), to the given *target data types*. More concretely, a reasonable target language for our running example may be Pascal, with its usual features, where one may freely invoke procedures for the programming primitives for lists: **head**, **tail**, etc.

Now, what is involved in deriving a program from a specification? Clearly, one has to transform the given problem specification into a program. Recall that the problem specification is a syntactic object built from specification primitives by means of specification constructs. Thus, one must express the specification constructs and primitives occurring in the problem specification in terms of the available programming constructs and primitives. For instance, if occurs and the existential quantifier are part of the problem specification, then one must express occurs by means of a program in terms of programming primitives for lists, as well as the existential quantifier in terms of programming constructs, the latter being indicated in 3.2.3.

2.2. Programming and Specification Languages

The preceding discussion has paved the way for some general remarks concerning specification and programming languages. A programming language is required to be executable, preferably in an efficient manner. This is why non-algorithmic operations are not expected to be part of such a language. Also, Hoare and Jifeng He [Hoa86] exclude intersection from their programming language mainly for reasons of efficiency. We shall argue, though, that intersection is a nice construct to have in a specification formalism (see 4.2 and 5.1). A specification language, or formalism, is not required to be executable. Its prime features should rather be expressiveness and ease of expression. Roughly speaking, the more restricted (but still universal) a programming language is, the more likely it is to attain its goals of efficient execution; whereas a very broad specification language (say, in the spirit of CIP's wide-spectrum language) is more likely to achieve its specification, and derivation, goals.

Here our specification formalism and programming language will both be based on an extended version of Tarski's calculus of binary relations (reviewed in 3.1). This extension amounts to considering structured universes, which makes it natural to add a couple of new operations on binary relations, as well as relativized identities (see 3.2). Our specification formalism will consist of relation-theoretic terms, and has been shown [Hae91; Vel91] to posses the expressive power of first-order logic (see 3.2.3). We obtain our programming language by restricting the terms to those contaning only algorithmic operations. So, our framework, based on the single unifying concept of binary relations over structured universes, supports a truly coherent wide-spectrum language, as we shall illustrate throughout the paper.

2.3. Specification of Problems and Programs

Let us now briefly comment on some views on specifying application problems.

First of all, since we have the full expressive power of first-order logic at our disposal [Hae91; Vel91], we can express in extended relational terms any application originally given by a first-order presentation.

But, an application specification is a theory about the application, expressed in its specification language. It often happens that one knows more about the application at the end of the derivation process than at the beginning. This occurs because, by tackling the derivation, one comes to grips with aspects of the application that were not envisioned at the outset. Thus, it may be wise, and profitable, to start the process with the initial, albeit incomplete, knowledge of the application embodied into some axioms, and let the derivation process guide one in enriching this specification. We shall illustrate how this can be done within our relational framework in 4.1 and 6.1.

Typically, the specification of the input-output behavior of a program consists of a a pair of formulas: $\phi(x)$ and $\psi(x, y)$ with variables x ranging over inputs and y over outputs, and the latter can always be written as $\psi(x, y) \wedge \theta(y)$. Now, by means of relativized identities, we can immediately represent this behavior by a term consisting of the composition of three relations, corresponding to the formulas ϕ, ψ and θ (saee 5.2 and 7). Furthermore, this specification can be regarded as a *generate-and-test* procedure. Of course, in order to derive more reasonable, and efficient, algorithms, some strategies are useful.

2.4. Formal Program Derivation

We have already mentioned that our framework, based on a single unifying concept, supports a truly coherent wide-spectrum language. We shall now give some indications concerning the appropriateness of our relational formalism for deriving and reasoning about programs.

Now, a good formalism for derivation purposes, in addition to being based on a coherent wide-spectrum language, should also be able to express, and manipulate, strategies and design decisions.

As mentioned, the specification of the input-output behavior of a program can be represented by a term, which can be regarded as a generate-and-test procedure. In view of the expressiveness of our relational formalism [Hae91; Vel91], one can be sure that any reasoning performed within first-order logic can easily be translated into our relational framework. Thus, we can freely employ usual logical reasoning, as illustrated in 6.1.

Also, some general problem-solving and algorithm design strategies can easily be expressed, and manipulated, within our relational formalism. Some examples are case division, reduction, trivialization, and divide-and-conquer. In a similar fashion, one can also express design decisions, "eurekas", and ideas akin to *weakest pre-specification* [Hoa86], as we shall illustrate in sections 4 and 6.

Furthermore, our relational formalism allows one to take advantage of the flexibility afforded by nondeterministic expressions. A simple example is the decomposition of a list into two sublists, from which the original list can be retrieved by concatenation. Such a decomposition is useful in expressing the idea underlying sorting algorithms of the merge-sort family.

Finally, the algebraic character of our formalism enables one to employ two possible kinds of intuitions in reasoning about a program. On the one hand, one can rely on a programming-like operational intuition concerning input-output behavior. On the other hand, one can also resort to algebraic intuitions, like distributing, commuting, etc. We shall illustrate these features in sections 4, 6 and 7.

Of course, many formalisms tend to be like a straight jacket, in that one has to carry the burden of some "formal noise" as a price for their advantages. Our formalism does have a non negligible amount of formal noise, but it is easily overcome once one gets the knack of it, and also amenable to automation, so that one can let the system take care of these matters, as indicated in section 8.

3. RELATIONAL ALGEBRAS

In this section we will first review Tarski's *Theory and Calculus of Binary Relations* [Tar41], and then introduce our extension, together with some introductory comments on its usefulness in expressing some simple aspects of program derivation.

3.1. TARSKI'S THEORIES OF BINARY RELATIONS

We now introduce Tarski's *Elementary Theory of Binary Relations* and *Calculus of Binary Relations* [Tar41], briefly analyzing then the algebraic structure of their intended models.

3.1.1. The Elementary Theory of Relations

Tarski develops his *Elementary Theory of Binary Relations* as an extension of first-order logic by introducing variables ranging over two sorts, namely *individuals*, denoted here by x, y, z, ... , and *relations*, denoted by italic letters *r*, *s*, *t*, ... The *atomic sentences* are of the form $r(x, y)$ (or $x\ r\ y$ -we will use both notations indistinctly - meaning "x is in relation *r* with y") and $r = s$ (where the symbol = denotes equality of relations). As usual, the *compound sentences* are built from atomic ones by means of logical connectives $\wedge$, $\vee$, $\leftrightarrow$, $\rightarrow$, $\neg$, and the quantifiers $\forall$ and $\exists$.

The symbols introduced by Tarski are, in our notation, ∞ (for the *universal relation*), 0 (for the *null relation*), 1 (for the *identity relation*) and $\wp$ (for the *diversity relation*), as *relational constants*, together with the following *operations on relations* : $^{-}$ (*complement*), $^{\smile}$ (*converse*), $+$ (*sum*), $\bullet$ (*intersection*), $\oplus$ (*relative sum*), and $;$ (*relative product*). The symbols ∞, 0, $^{-}$, $+$ and $\bullet$ are sometimes called *absolute* or *Boolean*, whereas 1, $\wp$, $^{\smile}$, $\oplus$ and $;$ are called *relative* or *Peircean*.

Finally, Tarski takes as extra-logical axioms the following 12 sentences:

EA1 $\quad \forall x \forall y(\infty(x, y))$

EA2 $\quad \forall x \forall y(\neg\, 0(x, y))$

EA3 $\quad \forall x(1(x, x))$

EA4 $\quad \forall x \forall y \forall z((r(x, y) \wedge 1(y, z) \rightarrow r(x, z)))$

EA5 $\quad \forall x \forall y(\wp(x, y) \leftrightarrow \neg\, 1(x, y))$

EA6 $\quad \forall x \forall y(\bar{r}(x, y) \leftrightarrow \neg\, r(x, y))$

EA7 $\quad \forall x \forall y(\breve{r}(x, y) \leftrightarrow r(y, x))$

EA8 $\quad \forall x \forall y(r + s(x, y) \leftrightarrow (r(x, y) \vee s(x, y)))$

EA9 $\quad \forall x \forall y(r \bullet s(x, y) \leftrightarrow (r(x, y) \wedge s(x, y)))$

EA10 $\quad \forall x \forall y(r \oplus s(x, y) \leftrightarrow (\forall z)(r(x, z) \vee s(z, y)))$

EA11 $\quad \forall x \forall y(r ; s(x, y) \leftrightarrow (\exists z)(r(x, z) \wedge s(z, y)))$

EA12 $\quad r = s \leftrightarrow \forall x \forall y(r(x, y) \leftrightarrow s(x, y))$

3.1.2. The Calculus of Binary Relations

Tarski's *Calculus of Binary Relations* may be regarde as the part of his *Elementary Theory* without any variables over individuals. In order to develop his *Calculus of Binary Relations*, Tarski derives from the axioms of his *Elementary Theory* an appropriate set of theorems whose variables are exclusively relational ones. He then

takes these theorems as the extra-logical axioms of his *Calculus of Binary Relations*. His axioms can be naturally divided into three groups as follows.

Axioms of the *Boolean* (*absolute*) symbols:

CA1 $r + s = s + r$

CA2 $r \bullet s = s \bullet r$

CA3 $(r + s) \bullet t = (r \bullet t) + (s \bullet t)$

CA4 $(r \bullet s) + t = (r + t) \bullet (s + t)$

CA5 $r + 0 = r$

CA6 $r \bullet \infty = r$

CA7 $r + \overline{r} = \infty$

CA8 $r \bullet \overline{r} = 0$

CA9 $\overline{\infty} = 0$

Axioms of the *relative* (*Peircean*) symbols:

CA10 $\breve{\breve{r}} = r$

CA11 $\breve{r} ; \breve{s} = \breve{s} ; \breve{r}$

CA12 $r ; (s ; t) = (r ; s) ; t$

CA13 $r ; 1 = r$

Axioms relating *Boolean* and *relative* symbols:

CA 14* $r ; \infty = \infty \vee \infty ; \overline{r} = \infty$

CA15 $(r ; s) \bullet \breve{t} = 0 \rightarrow (s ; t) \bullet \breve{r} = 0$

CA16 $\wp = \overline{1}$

CA17 $r \oplus s = \overline{\overline{r} ; \overline{s}}$

Shorter axiomatizations have been provided [Jón52; Chi50]. We adopt the above one in view of our interest in program derivation. Some of these results (marked with *) were later found too restrictive - in that they hold only for special classes of structures - and then dropped. This is the case of **CA**14*, as well as of Theorem C.14* below. Our development will not rely on them.

Some examples of the theorems of the *Calculus of Binary Relations* derived by Tarski are the following ones (notive that they do not mention individuals).

Theorem C.1 $((r ; s) \bullet t = 0$ iff $(s ; \breve{t}) \bullet \breve{r} = 0)$ and
$((r ; s) \bullet t = 0$ iff $(\breve{t} ; r) \bullet \breve{s} = 0)$

Theorem C.2 If $r \bullet \overline{s} = 0$ then $\breve{r} \bullet \overline{\breve{s}} = 0$

Theorem C.3 $\breve{r + s} = \breve{s} + \breve{r}$

Theorem C.4 $\breve{0} = 0$ and $\breve{\infty} = \infty$

Theorem C.5 If $s \bullet \overline{t} = 0$ then $(r ; s) \bullet \overline{r ; t} = 0$

Theorem C.6 $r ; (s + t) = (r ; s) + (r ; t)$

Theorem 2.7 $r\ ;\ 0 = 0$

Theorem C.8 If $r \bullet \bar{s} = 0$ then $(r\ ;\ t) \bullet \overline{s\ ;\ t} = 0$

Theorem C.9 $(r + s)\ ;\ t = (r\ ;\ t) + (s\ ;\ t)$

Theorem C.10 $0\ ;\ r = 0$

Theorem C.11 $\tilde{1} = 1$

Theorem C.12 $1\ ;\ r = r$

Theorem C.13 If $(\infty ;\ s) \bullet t = 0$ then $(\infty ;\ t) \bullet s = 0$

Theorem C.14* If $r \neq \infty$ then $(\infty\ ;\ \bar{r})\ ;\ \infty = \infty$

3.1.3. The Structure of Tarski's Algebras

The intended standard models of Tarski's *calculus* are *relational algebras* of the form $\Re = \langle \mathbf{R}, +, \bullet, \bar{}, \infty, 0, ;, \oplus, \breve{}, 1, \wp \rangle$, where $\mathbf{R}$ is a set of *binary relations* over a *universe* $\mathcal{U}$ closed under the Boolean and relative operations and constants. Let us consider such a set $\mathbf{R}$ of binary relations and analyze its algebraic structure under the relation $\subseteq$ (where, as usual, $r \subseteq s$ means $r + s = s$).

It is easy to see that:

i. $\subseteq$ is a *partial order*, i.e. $\langle \mathbf{R}, \subseteq \rangle$ is a *poset*;

ii. for every pair of *relations* r and s in $\mathbf{R}$, the relation $r + s$ (respectively $r \bullet s$) is the *least upper bound* (respectively *greatest lower bound*) of r and s.

Hence, $\langle \mathbf{R}, \subseteq \rangle$ is a *lattice* [Bur80, Grä71]. Thus, both $+$ and $\bullet$ are *associative*, *idempotent* and satisfy the *absorption laws* $r = r + (s \bullet t)$ and $r = r \bullet (s + t)$. It is also easy to notice that $\infty = lub(\mathbf{R})$ and $0 = glb(\mathbf{R})$.

From axioms **CA3** and **CA4** the lattice $\langle \mathbf{R}, \subseteq \rangle$ is *distributive*,. and axioms **CA5** through **CA8** imply that $\langle \mathbf{R} \subseteq \rangle$ is a *Boolean algebra*. It is quite clear that the relations of the form $a = \{\langle x, y \rangle\}$ are its *atoms*, i.e., the only elements $r \in \mathbf{R}$ that satisfy $r \subseteq a$ are 0 and a. Hence, $\langle \mathbf{R}, \subseteq \rangle$ is a *complete atomistic Boolean algebra* [McK40], in that for every $r \neq 0$ there exists an atom a such that $r \subseteq a$.

3.2. The Extended Calculus of Binary Relations

We shall now extend Tarski's *Calculus of Binary Relations*, by means of some new operations and constants, in order to obtain a *relational calculus* appropriate for program derivation.. As already noticed by Tarski [Tar41], the expressive power of his *Calculus of Binary Relations* falls short from his *Elementary Theory of Binary Relations*, as well as from first-order logic. This is partly due to the fact that his calculus does not have individual variables.

Before dealing with our extension, let us notice that we can already express some concepts, like determinism and refinement, relevant to program derivation.

A *deterministic* relation is a functional one: a, perhaps partial, function. The determinism of a relation r can be expressed in algebraic terms by any one of the (equivalent) conditions: $\breve{r}\,;\, r \subseteq 1$, or $r\,;\breve{r}\,;\, r = r$.

A simple, but typical, example of refinement is provided by the case of passing from a guarded conditional to an if_then_else command. The idea is reducing the nondeterminism, so decreasing the choice for input-output paths, without decreasing the domain, which is the intuition behind our concept of complete sub-relation.

Given relations r and s we say that r is a *complete sub-relation* of s (denoted $r \lhd s$) iff r is a subrelation of s with the same domain. More precisely:

$$r \lhd s \leftrightarrow r \subseteq s \wedge \mathcal{D}om(r) = \mathcal{D}om(s).$$

Let us now turn ot our extension. It may be naturally regarded as consisting of two steps. The first one is the introduction of relativized identities, in addition to the overall identity 1 on the entire universe $\mathcal{U}$, and the second one amounts to the introduction of two new operations, made reasonable because we shall have structured universes, with tree-like objects, rather than mere points.

3.2.1. Relativized Identities

An important motivation for the introduction of relativized identities is expressing pre-conditions, invariants, and the like, as well as having the advantges of many-sorted languages.

A simple-minded special case comes from reexamining our preceding considerations. We have mentioned that, in the case of palindrome, one would naturally have several basic sorts: C (for the elements of the lists), $\mathcal{L}^*$ (for the lists with elements of C), etc. Now, relation *rev* is intended to reverse lists; so we would like to restrict it to $\mathcal{L}^*$, for it is there that it means reversal. For this purpose, it would be convenient to be able to have $\mathcal{L}^*$ represented as a binary relation. A quite convenient representation is by means of the identity over lists:

$$1_{\mathcal{L}^*} = \{\langle x, x\rangle : x \in \mathcal{L}^*\}.$$

We can express the pre-restriction of *rev* to lists by means of $1_{\mathcal{L}^*}\,;\, rev$.

In general, a basic sort X will be represented by its relativized identity 1_X, and the *pre-restriction* of a relation r to X can be expressed by $1_X\,;\, r$. Notice that indeed $1_X\,;\, r = \{\langle x, y\rangle : \langle x, y\rangle \in r \wedge x \in X\}$ and $\mathcal{D}om(1_X\,;\, r) = X \cap \mathcal{D}om(r)$. Dually, we can have *post-restrictions* $r\,;\, 1_Y$.

Another example of the usability of these relativized identities comes from viewing the set of lists as partitioned into $\mathcal{L}^1$ (with the lists with length 0 and 1) and $\mathcal{L}^* - \mathcal{L}^1$ (with the longer lists). If we use relativized identities $1_{\mathcal{L}^*}$ and $1_{\mathcal{L}^*-\mathcal{L}^1}$, then we can

express this sort decomposition by means of $1_{\mathcal{L}^*} = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*\text{-}\mathcal{L}^1}$. If we now wish to pre-restrict *rev* to lists of length up to one, this is expressed by $1_{\mathcal{L}^1}$; *rev*.

3.2.2. Coping with the lack of Individual Variables

From the viewpoint of reasoning about programs, the absence of program, as well as individual, variables can be very advantageous [Bac78]. But, it has a somewhat undesirable side effect. For instance, consider the problem of checking whether a list is a palindrome. A very straightforward way of doing this is as follows: make two copies of the input list, reverse one of them, and compare the result with the other one, let untouched. For this purpose, it would be convenient to have a copying operation.

In a relational framework, the desired copying operation should be regarded as a relation. It should receive an object x as input, and output two copies of x. But, what is "two copies of x" supposed to mean? We do not want a ternary relation, for we wish to retain the input-output character. One way of getting the best from both worlds, so to speak, is to make the output a pair of the form [x, x]. Thus, we shall have this copying relation 2, which is described by

$$2 = \{\langle u, [u, u]\rangle : u \in \mathcal{U}\}.$$

Let us, now, explain in more precise terms what we have done. Our universe $\mathcal{U}$ is assumed to contain some set B of basic sorts. In the case of list reversal, sorting and palindrome, one would naturally have $B = \mathcal{C} \cup \mathcal{L}^* \cup \mathcal{B}ool$, consisting of elements (atoms), lists of these atoms and Boolean values. We also wish our universe to contain pairs of its elements: $\mathcal{U}$ is supposed to closed under the pair-formation operation $*$ with $u * v = [u, v]$. Thus, our universe $\mathcal{U}$ is assumed to contain all *trees* over some set B of basic sorts. In this sense, we now have a structured universe, rather than a mere set of *points*. Notice that operation $*$ is non-associative (which will be important later on for some distributivity properties), that is why we talk of *trees*, rather than *strings*).

The structured, tree-like, character of our universe $\mathcal{U}$ renders natural the introduction of some new *structural operations* on binary relations, namely fork and direct product.

We call *fork* of *relations* p and q the relation

$$p \nabla q = \{\langle u, v * w\rangle : \langle u, v\rangle \in p \wedge \langle u, w\rangle \in q\}.$$

and we define the *direct product* of *relations* p and q as

$$p \otimes q = \{\langle u * v, w * z\rangle : \langle u, w\rangle \in p \wedge \langle v, z\rangle \in q\}$$

where $*$ is the above structuring operation on the universe $\mathcal{U}$.

These new operations enable us to describe some interesting behaviors, especially form the programming viewpoint. In particular, recalling that 1 is the identity on $\mathcal{U}$, we can now define our copying relation by means of $2 = 1 \nabla 1$. Also, since we now have structured objects, it is natural to have operations to "unpack" them: relations with behavior

$$\Pi_1 = \{\langle u * v, u\rangle : u, v \in \mathcal{U}\} \text{ and } \Pi_2 = \{\langle u * v, v\rangle : u, v \in \mathcal{U}\}$$

These *projections* , which decompose objects into their components, are convenient shorthand for some special relations, in that they can be defined as follows

$$\Pi_1 = \widetilde{1 \nabla \infty} \text{ and } \Pi_2 = \widetilde{\infty \nabla 1}$$

3.2.3. Algebras of Relations over Structured Universes

An intended standard model of our *Extended Calculus of Binary Relations* involves a set $\mathbf{S}$ of*binary relations* over a *structureduniverse* $\mathcal{U}$ closed under the absolute, relative and structural operations and constants. In more detail, we start with a base set B of *basic sorts*,. and close it under the non-associative pair-formation operation $*$, by including *trees* of elements of B; this gives us our universe $\mathcal{U}$. More precisely, the universe $\mathcal{U} = B^*$ is the *free groupoid* generated by the base set B. Thus, given a set B of *basic sorts*, our intended standard models are *structured relational algebras* of the form $\mathfrak{S} = \langle \mathbf{S}, +, \bullet, \bar{}, \infty, 0, ;, \oplus, \breve{}, 1, \wp, (1_X)_{X \in B}, \nabla, \otimes\rangle$. Since $\mathfrak{S}$ is a special kind of algebra of binary relations, its algebraic structure under the relation $\subseteq$ is that of a *Boolean algebra.*

Some properties of the structural operations on relations are

(i) distributivity of fork and direct product over sum:

$$p \nabla (r + s) = (p \nabla r) + (q \nabla s)$$

$$p \otimes (r + s) = (p \otimes r) + (q \otimes s)$$

(ii) distributivity of converse over direct product:

$$\widetilde{r \otimes s} = \breve{r} \otimes \breve{s}$$

(iii) distributivity of fork and direct product:

$$(p \nabla q) ; (r \otimes s) = (p ; r) \nabla (q ; s)$$ [1]

(iv) distributivity of deterministic relation over fork:

$$t ; (p \nabla q) = ((t ; p) \nabla (t ; p)); \text{ whenever } \quad t ; \tilde{t} ; t = t$$

So, we now have binary relations over structured universes, consiting of objects with an internal tree-like structure, rather than simple points. This is then our extension to Tarski's *Calculus of Binary Relations* .

In our *Extended Calculus of Binary Relations* we can express, in an algebraic manner, several interesting properties of relations, as well as input-output behavior of programs. We have already seen how to express deterministim. Termination can also be simply expressed, for r *terminates over* X iff $X \subseteq \mathcal{D}om(r)$ iff $1_X \subseteq r ; \tilde{r}$. Also, projections are handy for rearrangement purposes: we can express the commutativity of union by $un = (\Pi_2 \nabla \Pi_1) ; un$, or, more conspicuously, by

$$(\Pi_1 \nabla \Pi_2) ; un = (\Pi_2 \nabla \Pi_1) ; un$$

As mentioned before, our *Extended Calculus of Binary Relations* has the expressive power of first-order logic, in the following sense. Given any first-order formula φ with variables partitioned into input variables and output variables, one can effectively construct a term $\mathcal{T}(\varphi)$ of the *Extended Calculus of Binary Relations* that defines the same input-output relation. More precisely, we have the next result [Hae91; Vel91].

Theorem *Given a first-order language $\mathcal{L}$, there exists a function $\mathcal{T}:\Phi(\mathcal{L}) \rightarrow \Im(\mathcal{L})$, such that for every structure $\mathfrak{C}$ for $\mathcal{L}$, $1_{\varphi\mathfrak{C}} = \mathfrak{C}[\mathcal{T}(\varphi)]$, for every formula φ of $\mathcal{L}$, where $1_{\varphi\mathfrak{C}}$ is the relativized identity over the set $\varphi^{\mathfrak{C}}$ defined by formula φ and $\mathfrak{C}[\mathcal{T}(\varphi)]$ is the binary relation denotating term $\mathcal{T}(\varphi)$ on structure $\mathfrak{C}$.*

A formal proof of this theorem and a more detailed discussion of expressiveness of our extended calculus of binary relations can be found in [Vel91]. We shall here just indicate the main ideas. It is clear that one can simulate the propositional connectives by means of the Boolean operations. For instance, assuming, inductively, that we already have relational terms $\mathcal{T}(\psi)$ and $\mathcal{T}(\theta)$ corresponding to formulas ψ and θ, respectively,

---

[1] This is one place where the non-associativity of the pair-forming operation $*$ is convenient.

then $\mathcal{T}(\psi \vee \theta) = \mathcal{T}(\psi) + \mathcal{T}(\theta)$ and $\mathcal{T}(\psi \wedge \theta) = \mathcal{T}(\psi) \bullet \mathcal{T}(\theta)$. The crucial remaining step is the simulation of the quantifiers, which we can do by $\mathcal{T}(\exists y\ \theta(x, y)) = \tilde{\tilde{\Pi}}_1 ; \mathcal{T}(\theta) ; \Pi_1$. It is interesting to notice that this expression is of the form *generate and test*, where *generate* is $\tilde{\tilde{\Pi}}_1 = 1 \nabla \infty$ and *test* is $\mathcal{T}(\theta)$, Π_1 being the *extraction function*. The rearrangement of variables can be expressed by a suitable combination of projections, forks and direct products (see Sections 7.4 and 8.2).

4. SOME MOTIVATING DERIVATIONS

In this section, we shall outline derivations of programs for reversing and sorting lists. These introductory examples are intended to play a twofold role: illustrating the use of our *Extended Calculus of Binary Relations* as well as paving the way for some more general considerations.

Before going on with the illustrative examples, let us introduce the basic data type *list*. We will deal with lists of elements from a given set C. So, if we denote by Λ the null list, the domain of lists is $\mathcal{L}^* = \bigcup_{i \in N} \mathcal{L}^i$, where $\mathcal{L}^i$ is the set of all lists of length between 0 and i, with $\mathcal{L}^0 = \{\Lambda\}$. Its specification and programming primitives are

$hd = \{\langle x, c\rangle : x \in \mathcal{L}^* - \mathcal{L}^0 \wedge c \in C \wedge c = \mathbf{head}(x)\}$
$tl = \{\langle x, y\rangle : x \in \mathcal{L}^* - \mathcal{L}^0 \wedge y = \mathbf{tail}(x)\}$
$cns = \{\langle [c, x], z\rangle : c \in C \wedge x \in \mathcal{L}^* \wedge z = \mathbf{cons}(c, x)\}$
$init = \{\langle x, y\rangle : x \in \mathcal{L}^* - \mathcal{L}^0 \wedge y = \mathbf{initial}(x)\}$
$lst = \{\langle x, c\rangle : x \in \mathcal{L}^* - \mathcal{L}^0 \wedge c \in C \wedge c = \mathbf{last}(x)\}$
$app = \{\langle [x, c], y\rangle : x \in \mathcal{L}^* \wedge c \in C \wedge y = \mathbf{append}(x, c)\}$
$lhd = \{\langle x, y\rangle : x \in \mathcal{L}^* - \mathcal{L}^0 \wedge y = \mathbf{cons}(\mathbf{head}(x), \Lambda)\}$
$llst = \{\langle x, y\rangle : x \in \mathcal{L}^* - \mathcal{L}^0 \wedge y = \mathbf{cons}(\mathbf{last}(x), \Lambda)\}$
$md = \langle x, y\rangle : x \in \mathcal{L}^* - \mathcal{L}^1 \wedge y = \mathbf{initial}(\mathbf{tail}(x))\}$

Here, **cons**, **head** and **tail** are the usual list operations (acting at the beginning of a list), whereas **append**, **last** and **initial** are their counterparts acting at the end of a list. One should bear in mind ther distinction between the list structure on one hand and the underlying universe structure on the other: [c, x], for instance, is a structured object of our universe, rather than a list beginning with c and followed by x, the latter being **cons**(c, x).

4.1. LIST REVERSAL

Let us examine a possible derivation of a program for reversing lists.

We shall start from its definition in the elementary theory of relations, i.e., $\forall x \forall y (rev(x, y) \leftrightarrow x \in \mathcal{L}^* \wedge y = \hat{x})$, where $\hat{x}$ denotes the reversal of x.

We will use the *knowledge* of what *rev* should be in order to *construct* a theory for it within the calculus of relations, instead of *deriving* a translation into a relational specification directly from the above definition of *rev*.

Let us start with the trivialization of *rev* into two cases by partitioning the list domain $\mathcal{L}^*$ into $\mathcal{L}^1$ and $\mathcal{L}^* - \mathcal{L}^1$, which can be expressed by the equation

$$1_{\mathcal{L}^*} ; rev = (1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1}) ; rev \tag{1}$$

By distributing ; over + according to theorem C.6, (1) becomes

$$1_{\mathcal{L}^*} ; rev = 1_{\mathcal{L}^1} ; rev + 1_{\mathcal{L}^*-\mathcal{L}^1} ; rev \tag{2}$$

The rationale for this decomposition was the realization that the reversal of any null or unitary list is itself. This is expressed as $1_{\mathcal{L}^1} ; rev = 1_{\mathcal{L}^1}$. Thus

$$rev = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; rev \tag{3}$$

So, we are left with the reversal of a non-trivial list: $rev_1 = 1_{\mathcal{L}^*-\mathcal{L}^1} ; rev$. This is probably a good point to introduce a *eureka*. We can try to take advantage of the inductive structure of the domain $\mathcal{L}^*$. But, instead of tackling an inductive solution in a *divide-and-conquer* fashion, we can actually derive it by reasoning as follows.

We can imagine the last "part" as a kind of *join*, such as the concatenation of the *lhd*, the middle part and the *llst* of a decomposed list whose reversed parts we already have. Hence, we can write the following equation:

$$x ; erk = rev_1 \tag{4}$$

So the *eureka* should be something like

$$erk = \widetilde{\left((lhd \nabla md) \nabla llst\right)} \tag{5}$$

We will usually write this kind of expression in the form $erk = \left(\left(lhd \,\tilde{\nabla}\, md\right) \tilde{\nabla}\, llst\right)$.

The idea behind (4) is that we are looking for a, yet to be determined, relation x which, multiplied on its right by *erk*, yields rev_1. Now, by multiplying both sides of (4) by the converse of *erk*, we have

$$x \; ; erk \; ; \widetilde{erk} = rev_1 \; ; \widetilde{erk} \tag{6}$$

Since $\mathcal{R}an(\widetilde{erk}) = \mathcal{D}om(erk)$ and $\widetilde{erk}$ is deterministic, we have $erk \; ; \widetilde{erk} = 1_{\mathcal{D}om(erk)}$. So, $x = rev_1 ; \widetilde{erk}$ is a solution for (4), which, in view of (5), we can write as

$$x = 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \; ; rev \; ; (hd \nabla \; md \nabla \; tl) \tag{7}$$

Now, since *rev* is deterministic, we know that it distributes over fork (see property (iv) in 3.2.3). Thus, we can replace in (7) $rev \; ; (hd \nabla \; md \nabla \; tl)$ by $(rev \; ; hd) \nabla \; (rev \; ; md) \nabla \; (rev \; ; tl)$ to obtain

$$x = 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \; ; [\; (rev \; ; hd) \nabla \; (rev \; ; md) \nabla \; (rev \; ; tl) \;) \tag{8}$$

At this point it is heuristically simpler than at the very beginning of the process to express the following three axioms of our theory of *rev*

$$rev \; ; hd = tl$$

$$rev \; ; tl = hd$$

$$rev \; ; md = md \; ; rev$$

Notice that an earlier specification and derivation of *rev* would probably lead one to choose something like $rev \; ; hd = tl$ and $rev \; ; tl = tl \; ; rev$ as (part of) the theory of *rev*. This would be heuristically less useful for the case at hand, in view of the structuring principle embodied in the eureka.

Continuing with our derivation, we can rewrite (8) as

$$x = 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \; ; (\; tl \nabla \; (md \; ; rev) \nabla \; hd \;)$$

which can be *unfolded* into (4), yielding

$$rev_1 = 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \; ; (\; tl \nabla \; (md \; ; rev) \nabla \; hd \;) \; ; erk \tag{9}$$

Finally, by *unfolding* (9) into (3), we obtain:

$$rev = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \; ; (\; tl \nabla \; (md \; ; rev) \nabla \; hd \;) \; ; erk \tag{10}$$

This is a final expression for *rev*.

For the sake of clarity we shall use sometimes resort to a two-dimensional notationfor our relational terms. In this notation (10) will have the following aspect

$$rev = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} \; ; \begin{pmatrix} \mathit{lst} \\ \nabla \\ \mathit{hd} \; ; \; rev \\ \nabla \\ \mathit{lhd} \end{pmatrix} ; \mathit{erk}$$

Now, by definition, whenever $x \in \mathcal{L}^{n+1}$ with $n \geq 1$:

if $\langle x, z \rangle \in \mathit{hd}$ then, $z \in \mathcal{L}^{n-1} \subseteq \mathcal{L}^n$ (i. e. $1_{\mathcal{L}^{n+1}} \; ; \mathit{hd} = \mathit{hd} \; ; 1_{\mathcal{L}^n}$) ,

if $\langle x, z \rangle \in \mathit{lhd}$ then, $z \in \mathcal{L}^1 \subseteq \mathcal{L}^n$, and

if $\langle x, z \rangle \in \mathit{lst}$ then, $z \in \mathcal{L}^1 \subseteq \mathcal{L}^n$.

Thus, equation (10) expresses a recursive algorithm for reversing lists. The termination of this algorithm hinges on the fact that successive applications of *hd* eventually reduce any list to one with length at most 1. This can be expressed in the infinitary version of our formalism as

$$1_{\mathcal{L}^1} \; ; \; \tilde{\tilde{\tilde{\mathit{hd}}}} \; ;^* = 1_{\mathcal{L}^*} \tag{11}$$

where $p;^*$ is the *closure* $\sum_{n \in N} p;^n$, with $p;^n$ being the product of p by itself n times (i.e. $p;^0 = p$ and $p;^{n+1} = p \; ; \; p;^n$) and $\sum_{n \in N} p = \bigcup_{n \in N} p$.

In other words, equations (10) and (11) together express the following inductive argument.

Basis:

If $x \in \mathcal{L}^1$ then the reversal of x is x itself.

Inductive step:

If $x \in \mathcal{L}^{n+1}$, with $n \geq 1$, then we have two possible cases:

i) $x \in \mathcal{L}^n$, then, the problem is solved by inductive hypothesis,

ii) $x \notin \mathcal{L}^n$, then, we can

decompose x into three lists: **cons(head**(x), Λ), **cons(last**(x), Λ), **initial(tail**(x)), each of which belonging to $\mathcal{L}^n$, obtain their reversals (by inductive hypothesis) and join appropriately these reversed lists.

We still have some further comments about the derivation we have developed. First, after introducing the *eureka*, the derivation of a term for *rev* is simple calculation. Also, the requirement of equality in equation (4) is usually too strong, refinement being enough: we can require simply that $x ; erk \lhd rev_1$ (i. e. $x ; erk \subseteq rev_1$ and $\mathcal{D}om(x ; erk) = \mathcal{D}om(rev_1)$ as introduced in 3.2). It is also worth pointing out that one solution of the equation $x ; erk \lhd rev_1$, turns out to be the *weakest pre-specification* introduced by Hoare and Jifeng He [Hoa86]. Our solution for $x ; erk = rev_1$ amounts to their *strongest pre-specification*.

A *Begriffsschrift*-like diagram for (10) appears in figure 1.

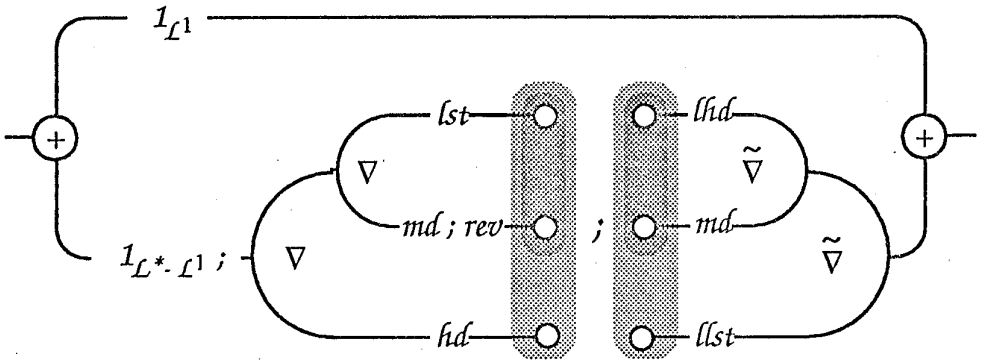

Figure 1: *Begriffsschrift*-like diagram for *rev*

Notice that the rightmost part of the preceding diagram, i.e. *erk*, can be detached when *rev* is used, as in this case, as part of a larger algorithm as was the case in [Hae91], where it is used in derving palindrome..

4.2. List Sorting

It is sometimes suggested that the operation • (intersection of relations) should not be present in a (relational) programming language because its usage tends to yield extremely inefficient programs. But, should it also be eliminated from (relational) specification and derivation languages?

As an example, assume that one wishes to derive a sorting program, call it *sort*, and that one already has available the following two (nondeterministic) programs:

to_ord, which assigns to a given list an arbitrary ordered list;

perm, which assigns to a given list an arbitrary permutation of it.

Then, one can write

$$sort = to\_ord \bullet perm \tag{12}$$

which is, at the very least, an extremely inefficient program for sorting [Hoa86].

This is the example is used by Hoare and Jifeng He [Hoa86] in presenting their case against the inclusion of intersection $\circ$ in a programming language.

But, let us ask ourselves: should we also eliminate $\circ$ from a program specification and calculation languages? In other words, $sort = to\_ord \circ perm$ is clearly an undesirable program, but, is it an undesirable specification as well? A relational, as opposed to operational, reading of (12) yields: *a pair* $\langle x, y \rangle$ *of lists is in the relation sort iff* y *is ordered and* y *is a permutation of* x. This expression is a relational instance of the *axiom of separation* , and we certainly would not wish to prevent the use of such a powerful specification tool simply by eliminating $\circ$ from our language.

Our point is twofold. First, $sort = to\_ord \circ perm$ is, to be sure, a poor program, but it is a fairly reasonable specification for a sorting algorithm. Second, we can construct from this specification a good (i.e., efficient) program within our relational calculus. Let us proceed to indicate how this can be done.

Let 1_{ord} be the identity over ordered lists. Then we can write

$$to\_ord = 1_{\mathcal{L}^*} \; ; \infty \; ; \; 1_{ord}$$

We can now eliminate $\bullet$ form (1), by applying a general result (see Corollary 5.2 below in 5.1) to obtain

$$sort = to\_ord \bullet perm = 1_{\mathcal{L}^*} \; ; \; perm \; ; \; 1_{ord} \tag{13}$$

Now, consider a program *merge* that merges two lists into one. Then, *perm* has the following property

$$perm = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} \; ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} \; ; \left(perm \otimes perm\right) ; \; merge$$

By unfolding the latter expression for *perm* into (13) we obtain

$$\begin{aligned} sort &= 1_{\mathcal{L}^*} \; ; \left(1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} \; ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} \; ; \left(perm \otimes perm\right) ; \; merge\right) ; \; 1_{ord} \\ &= 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} \; ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} \; ; \left(perm \otimes perm\right) ; \; merge \; ; \; 1_{ord} \\ &= 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} \; ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} \; ; \left(perm \otimes perm\right) ; \left(1_{\mathcal{L}^*} \otimes 1_{\mathcal{L}^*}\right) ; \; merge \; ; \; 1_{ord} \end{aligned}$$

Since $1_{\mathcal{L}^*} + 1_{ord} = 1_{\mathcal{L}^*}$, we can now write

$$\begin{aligned} sort = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} \; ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} \; ; \left(perm \otimes perm\right) ; \\ \left[\left(1_{\mathcal{L}^*} \otimes 1_{\mathcal{L}^*}\right) + \left(1_{ord} \otimes 1_{ord}\right)\right] ; \; merge \; ; \; 1_{ord} \end{aligned}$$

and, by distributing + over ; and commuting both terms, we can arrive at

$$sort = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} ; (perm \otimes perm) ; \left(1_{\mathcal{L}^*} \otimes 1_{\mathcal{L}^*}\right) ; merge ; 1_{ord} +$$
$$1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} ; (perm \otimes perm) ; \left(1_{ord} \otimes 1_{ord}\right) ; merge ; 1_{ord}$$

The latter is an expression of the form $sort = sort + \chi$. But, $\chi \lhd sort$, as defined in 3.2.1, so we can use the following refinement

$$sort = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} ; (perm \otimes perm) ; \left(1_{ord} \otimes 1_{ord}\right) ; merge ; 1_{ord}$$
$$= 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} ; \left(\left(1_{\mathcal{L}^*} ; perm ; 1_{ord}\right) \otimes \left(1_{\mathcal{L}^*} ; perm ; 1_{ord}\right)\right)$$
$$; \left(1_{ord} \otimes 1_{ord}\right) ; merge ; 1_{ord}$$

Now, by folding now (13) into this expression, we get

$$sort = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} ; (sort \otimes sort) ; \left(1_{ord} \otimes 1_{ord}\right) ; merge ; 1_{ord}$$

Finally, by calling $merge\_ord = \left(1_{ord} \otimes 1_{ord}\right) ; merge ; 1_{ord}$, we obtain

$$sort = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; \tilde{c}\tilde{o}\tilde{n}\tilde{c} ; (sort \otimes sort) ; merge\_ord$$

which is a typical *merge-sort* algorithm.

Thus, from a specification, which, as a program, is quite inefficient, we were able to derive a quite reasonable program in terms of efficiency. The morale of this development is that there is a huge difference between choosing the repertoire of constructs for a *programming language* and for a *formal program construction language.*

5. SETS AND RELATIONS

The previous section has paved the way for some theoretical considerations of interest to program derivation, especially the use of partial identities and intersection. We shall examine them in this section, where we first clarify the role of intersection and then use it in specifying sets, after briefly considering the algebra of partial identities.

5.1. THE ROLE OF INTERSECTION

In view of the remarks in 4.2 concerning the role of intersection in programming, specification and derivation languages, it is generally convenient to be able to rewrite

intersection in terms of more efficient constructs. A special, but quite frequent, case is dealt with in the next result.

Proposition 5.1 *Let r and s be relations such that $r = 1_\partial ; t ; 1_\rho$. Then*

$$r \bullet s = 1_\partial ; s ; 1_\rho \text{ iff } (1_\partial ; s ; 1_\rho) \subseteq (1_\partial ; t ; 1_\rho).$$

Proof. We have

$$r \bullet s = (1_\partial ; t ; 1_\rho) \bullet s = (1_\partial ; t ; 1_\rho) \bullet (1_\partial ; s ; 1_\rho)$$

$$\text{and } (1_\partial ; t ; 1_\rho) \bullet (1_\partial ; s ; 1_\rho) = (1_\partial ; t ; 1_\rho)$$

$$\text{iff } (1_\partial ; s ; 1_\rho) \subseteq (1_\partial ; t ; 1_\rho)$$

QED

A further special case of intersection elimination is delat with in the next corollary, which deals with "rectangular" relations.

Corollary 5.2 *Given a relation r, the following are equivalent:*

a) $r = 1_\partial ; \infty ; 1_\rho$

b) $r \bullet s = 1_\partial ; s ; 1_\rho$ *for every relation s*

c) $r = r ; \infty ; r$

Proof.

(a $\Rightarrow$ b) Clearly $(1_\partial ; s ; 1_\rho) \subseteq (1_\partial ; \infty ; 1_\rho)$

(b $\Rightarrow$ c) Take $s = \infty$

(c $\Rightarrow$ a) $r ; \infty ; r = 1_\partial ; \infty ; 1_\rho ; \infty ; 1_\partial ; \infty ; 1_\rho = 1_\partial ; \infty ; 1_\rho$

QED

More generally, one can always eliminate intersection, because it can be defined in terms of fork and equality filter, as shown in the next result.

Theorem 5.3 *Given relations r and s, we have*

a) $r \bullet s = (r \nabla s) ; \tilde{2}$ *and*

b) $r \nabla s = (r \nabla \infty) \bullet (\infty \nabla s) = (r ; \tilde{\Pi}_1) \bullet (s ; \tilde{\Pi}_2)$.

Proof.

a) For any u, v $\in \mathcal{U}$, we have from the definitions u $(r \nabla s)$; $\tilde{2}$ v iff u $(r \nabla s)$ w $*$ z and w $*$ z $\tilde{2}$ v, for some w, z $\in \mathcal{U}$, iff u r w, u s z and w = v = z, for some w, z $\in \mathcal{U}$, iff u r v and u s v iff u $r \bullet s$ v.

b) For any u, v, w $\in \mathcal{U}$, we have from the definitions
u $(r \nabla \infty) \bullet (\infty \nabla s)$ v $*$ w iff u $(r \nabla \infty)$ v $*$ w and
u $(\infty \nabla s)$ v $*$ w iff u r v v, u ∞ v and u s v iff u $(r \nabla s)$ v $*$ w.

QED

The preceding theorem also shows that fork can be defined in terms of intersection and projections, thereby establishing a tight connection between intersection and fork. Thus, if one wishes to eliminate intersection from the specification language, then one must eliminate fork as well, since one wishes to retain the equality filter. This would decrease the expressive power of the specification language, in that one cannot have existential quantifiers (see the definition of fork in 3.2.2).

5.2 THE ALGEBRA OF FILTERS

Our calculus is based on binary relations of input-output pairs as a single unifying concept. But, we often have to deal with sets, rather than relations. In order to handle sets in this framework, we represent them by binary relations. For this purpose, several resonable alternatives are available, for instance by resorting to the partiality of the representing relation, one of them being the conditions of [Hoa86]. For our goals of program derivation, however, another representation seems to be more convenient, namely representing a set by the identity over it.

By a *filter* we mean any subrelation of the identity relation 1. So, a filter is a reltivized identity: it acts identically on certain objects and filters out other objects, thereby representing its domain. Thus, subsets of the universe can be represented by their filters. In other words, we have a bijection I between the power-set $\mathcal{P}(\mathcal{U})$ of $\mathcal{U}$ and the set $\mathcal{F}(\mathcal{U})$ of filters over $\mathcal{U}$, defined by assigning to $A \subseteq \mathcal{U}$ the filter 1_A.

It is interesting to notice that the set of filters forms a Boolean algebra (having as least element the null relation 0 and as largest element the identity relation 1), which is isomorphic to the power-set of $\mathcal{U}$ under the above bijection. Indeed, the following rules are useful for calculating with filters

$1_{A \cup B} = 1_A + 1_B$, $1_{A \cap B} = 1_A \bullet 1_B$, $1_{\overline{A}} = 1 - 1_A$
and $1_\infty = 1$, $1_0 = 0$.
Moreover, for the relative operations we have
$1_A ; 1_B = 1_A \bullet 1_B$, and $\tilde{1}_A = 1_A$.

5.3. SPECIFYING SETS

As mentioned in 4.2, the axiom of separation is a very powerful and usual manner of specifying a subset of a given set. We shall now examine how we can take advantage of this specification tool and how we can derive programs for testing membership in sets so defined.

The situation is as follows. Given a set A and a formula φ, the Axiom of Separation yields the set $B = \{u \in A : \varphi(u)\}$, consisting of those elements u of A that satisfy $\varphi(u)$. We wish to specify this subset B of A. A usual manner of specifying a subset is by means of a membership test that receives as input an element of A, and outputs **T** or **F** according to whether or not the input belongs to B. Another manner, which is very convenient for derivations, especially in intermediate steps, is by means of a filter.

5.3.1 Specifying Sets by Filters

Given a set S, by the *filter over* S we mean the filter 1_S, which, on iput u, outputs u itself iff u belongs to S, otherwise no output is produced. In this sense, the filter over set S does represent it as a binary relation.

We wish to derive a specifation for the filter over the set

$$B = \{x : x \in A \wedge \varphi(x)\}$$

from the, assumed available, following two filters:

$$1_A = \{[u,u] \in \mathcal{U} \otimes \mathcal{U} : u \in A\}$$
$$1_\varphi = \{[u,u] \in \mathcal{U} \otimes \mathcal{U} : \varphi(u)\}$$

We clearly have, for any object x of our universe:

$$x\, 1_B\, x \leftrightarrow x \in B \leftrightarrow \underbrace{x \in A}_{\updownarrow\; x\,1_B\,x} \wedge \underbrace{\varphi(x)}_{\updownarrow\; x\,1_\varphi\,x}$$

Hence

$$1_B = 1_A \bullet 1_\varphi$$

5.3.2 Specifying Sets by Tests

By the *characteristic predicate* for B as a subset of A we mean the predicate that tests which members of A belong to B. We shall denote this predicate by B_A. So:

$$B_A(x) = \begin{cases} \mathbf{T} \text{ if } x \in B \\ \mathbf{F} \text{ if } x \notin B \wedge x \in A \end{cases}$$

Let us define the following relations, which will be useful for several purposes,

$$\mathit{true} = \{\langle u, \mathbf{b}\rangle : u \in \mathcal{U} \wedge \mathbf{b} = \mathbf{T}\}$$

$$\mathit{false} = \{\langle u, \mathbf{b}\rangle : u \in \mathcal{U} \wedge \mathbf{b} = \mathbf{F}\}$$

Thus, we can write

$$B_A = 1_B \ ; \ \mathit{true} + 1_{A-B} \ ; \ \mathit{false}$$

But, dually

$$B - A = \{x : x \in A \wedge x \notin B\} = \{x : x \in A \wedge \neg\varphi(x)\}$$

and

$$1_{A-B} = 1_A \bullet 1_{\neg\varphi}.$$

Thus, we obtain, with the alternative notation $\dot{1}_B$ for $B_A(x)$,

$$\dot{1}_B = 1_B \ ; \ \mathit{true} + 1_{B-A} \ ; \ \mathit{false}$$

5.3.3 Specifying Sets by Fixpoints

Another convenient way of specifying a set is by means of a function whose set of fixpoints is exactly the desired set. An example illustrating the usefulness of such specifications is provided in 6.1 by the derivation of palindrome, where the specification already happens to provide such a function.

Without loss of generality, we restrict ourselves to considering formulas. Given a formula φ, we wish to define a function $\Phi : \mathcal{U} \to \mathcal{U}$ such that $\Phi(u) = u$ iff $\varphi(u)$ holds. We shall now see in general how such a function can be specified from a given (specification for) formula φ.

First, notice the following property of our structured universe: for every $u \in \mathcal{U}$, $u \neq [u, u]$ Thus, if we define $\Phi : \mathcal{U} \to \mathcal{U}$ by

$$\Phi(u) = \begin{cases} u & \text{if } \phi(u) \text{ holds} \\ [u, u] & \text{if } \phi(u) \text{ does not hold} \end{cases}$$

we see that if has the required property: $\Phi(u) = u$ iff $\varphi(u)$ holds.

It remains to see how a specification for function Φ can be obtained form one for formula φ.

But, recal that the characteristic predicate for φ outputs T iff $\varphi(u)$ holds. Then, we see that we have

$$\Phi = \left(1 \nabla \dot{1}_{\varphi} \; ; \; 1_{\mathsf{T}}\right) ; \; \Pi_1 + \left(1 \nabla \dot{1}_{\varphi} \; ; \; 1_{\mathsf{F}}\right) ; \; \Pi_1 \; ; \; 2$$

where 1_{T} and 1_{F} are the filters over the Boolean values.

This is a specification for our function Φ in terms of the characteristic predicate for φ, and a specification for the latter can be obtained from 1_{φ} and $1_{\neg\varphi}$, as already seen.

We can now employ such a function $\Phi : \mathcal{U} \to \mathcal{U}$, with $\Phi(u) = u$ iff $\varphi(u)$ holds, in order to provide alternative specifications for the filter and the characteristic predicate of a subset. For

$$\begin{aligned} B &= \{u : u \in A \wedge \varphi(u)\} \\ &= \{u : u \in A \wedge \Phi(u) = u\}, \text{ then} \\ 1_B &= 1_A \bullet \Phi \\ 1_{A-B} &= 1_A \bullet \overline{\Phi} = 1_A \bullet \left(\Phi \; ; \; \overline{1}\right) \end{aligned}$$

We can now can eliminate $\bullet$ by means of Theorem 5.3, and obtain

$$\begin{aligned} 1_B &= 1_A \bullet \Phi = 2 \; ; \left(1_A \otimes \Phi\right) ; \tilde{2} \text{ and} \qquad (14) \\ 1_{A-B} &= 1_A \bullet \left(\Phi \; ; \; \overline{1}\right) = 2 \; ; \left(1_A \otimes \left(\Phi \; ; \; \overline{1}\right)\right) ; \tilde{2} \text{ but,} \\ B_A &= 1_B \; ; \; true + 1_{A-B} \; ; \; false, \text{ so,} \\ B_A &= 1_A \; ; \; 2 \; ; \left(1_A \otimes \Phi\right) ; \underbrace{\left(\tilde{2} \; ; \; true + \tilde{\tilde{\tilde{1}}} \nabla \tilde{1} ; \; false\right)}_{\ddot{1}} \end{aligned}$$

since

$$\begin{aligned} 2 \; ; \left(1_A \otimes \left(\Phi \; ; \; \overline{1}\right)\right) ; \tilde{2} &= 1_A \; ; \; 2 \; ; \left(1 \otimes \left(\Phi \; ; \; \overline{1}\right)\right) ; \tilde{2} \\ &= 1_A \; ; \; 2 \; ; \; (1 \otimes \Phi) \; ; \underbrace{\left(1 \otimes \overline{1}\right) ; \tilde{2}}_{(a)} \end{aligned}$$

$$\tilde{\tilde{(a)}} = \tilde{\tilde{\tilde{\tilde{\tilde{\tilde{\left(1 \otimes \overline{1}\right)}}}}}} ; \tilde{2} = 1 \nabla 1 \; ; \left(1 \otimes \overline{1}\right) = \left(1 \; ; 1 \otimes 1 \; ; \overline{1}\right) = 1 \nabla \overline{1}$$

$$(a) = \tilde{\tilde{\tilde{\tilde{1}}}} \nabla \tilde{\tilde{\tilde{\tilde{1}}}}$$

6. SOME DERIVATION RATIONALES

In this section we wish to employ the machinery presented so far to illustrate how one can express and use some rationales in program derivation. We do this by means of an illustrative example and the analysis of a simple strategy, namely trivialization.

6.1. AN EXAMPLE: PALINDROME

We shall now derive a program for testing whether a given list is a palindrome: a program for the characteristic predicate of the palindromes.

The specification of palindrome as a subset of the set of lists is

$pal = \{x : x \in \mathcal{L}^* \wedge x = \hat{x}\}$, where $\hat{x}$ denotes the *reversal* of x

and its *characteristic predicate* is

$$\dot{pal}(x) = \begin{cases} \mathbf{T} \text{ if } x \in pal \\ \mathbf{F} \text{ if } x \in \mathcal{L}^* \wedge x \notin pal \end{cases}$$

Now, let *rev* be the relation defined, as in 4.1, by $x\,rev\,y \leftrightarrow y = \hat{x}$.

Then we can write

$$1_{pal} = 1_{\mathcal{L}^*} \bullet rev$$

As we already know from 5.3.2, we can derive from here the following expressions for $\dot{pal}(x)$,

$$\dot{pal} = 1_{pal} \;;\; true + 1_{\mathcal{L}^* - pal} \;;\; false$$

$$1_{\mathcal{L}^* - pal} = 1_{\mathcal{L}^*} \bullet \overline{rev}$$

$$\dot{pal} = \left(1_{\mathcal{L}^*} \bullet rev\right) \;;\; true + \left(1_{\mathcal{L}^*} \bullet \overline{rev}\right) \;;\; false$$

But, *rev* is functional (i.e., deterministic) and total from lists to lists (i.e., it has all lists in its domain and range, which can be written algebraically as $rev = 1_{\mathcal{L}^*}; rev; 1_{\mathcal{L}^*}$). Thus,we can write $\overline{rev} = \overline{1_{\mathcal{L}^*}; rev; 1_{\mathcal{L}^*}} = 1_{\mathcal{L}^*}; rev \;;\; \overline{1} \;;\; 1_{\mathcal{L}^*}$, then

$$1_{\mathcal{L}^* - pal} = 1_{\mathcal{L}^*} \bullet \left(rev \;;\; \overline{1} \;;\; 1_{\mathcal{L}^*}\right) = 1_{\mathcal{L}^*} \bullet \left(rev \;;\; \overline{1}\right)$$

So, in view of (14), we can now write

$$\dot{pal} = 1_{\mathcal{L}^*} \;;\; 2 \;;\; (1 \otimes rev) \;;\; \ddot{1},$$

which, by using the decomposition $1_{\mathcal{L}^*} = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1}$, we can trivialize as follows

$$\dot{pal} = 1_{\mathcal{L}^1} \;;\; true + \underbrace{1_{\mathcal{L}^*-\mathcal{L}^1} \;;\; 2 \;;\; (1 \otimes rev) \;;\; \ddot{1}}_{(a)}$$

Let us now continue with the derivation of part (a) as follows

$$\begin{aligned} 1_{\mathcal{L}^*-\mathcal{L}^1} \;;\; 2 \;;\; (1 \otimes rev) \;;\; \ddot{1} &= 1_{\mathcal{L}^*-\mathcal{L}^1} \;;\; (1 \nabla rev) \;;\; \ddot{1} \\ &= \left[\left(1_{\mathcal{L}^*-\mathcal{L}^1} \;;\; 1 \right) \nabla \left(1_{\mathcal{L}^*-\mathcal{L}^1} \;;\; rev \right) \right] \;;\; \ddot{1} \\ &= \left[1_{\mathcal{L}^*-\mathcal{L}^1} \nabla \underbrace{\left(1_{\mathcal{L}^*-\mathcal{L}^1} \;;\; rev \right)}_{(b)} \right] \;;\; \ddot{1} \end{aligned} \tag{15}$$

In order to continue with part (b), we introduce the *eureka*

$$1_{\mathcal{L}^*-\mathcal{L}^1} = \underbrace{(hd \nabla md \nabla lst)}_{dcmp} \;;\; \underbrace{\tilde{(hd \nabla md \nabla lst)}}_{rcmb} \tag{16}$$

A graphic representation, as in the diagram in figure 2, can clarify the goal envisioned in our *eureka*.

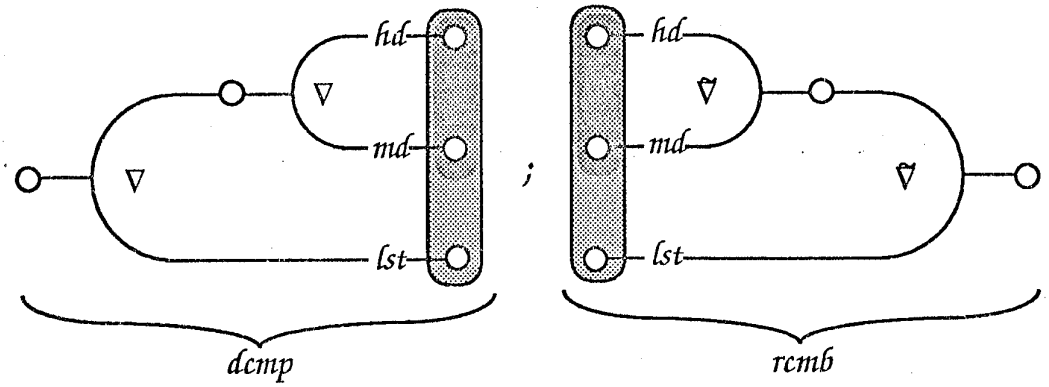

Figure 2: The goal of the eureka.

Now, we can proceed with (b), by writing

$$1_{\mathcal{L}^*-\mathcal{L}^1} \;;\; rev = (hd \nabla md \nabla lst) \;;\; \tilde{(hd \nabla md \nabla lst)} \;;\; rev \tag{17}$$

A resonable subgoal at this point would be moving *rev* to the right over *rcmb*. For this purpose, we need properties of *rev* allowing its *promotion* to the left of *rcmb*. The required properties are clearly similar to the axioms of our theory of *rev* in 4.1: $1_{\mathcal{L}^*\text{-}\mathcal{L}^1} ; rev ; hd = 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} ; lst$, $1_{\mathcal{L}^*\text{-}\mathcal{L}^1} ; rev ; lst = 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} ; hd$ and $1_{\mathcal{L}^*\text{-}\mathcal{L}^1} ; md ; rev = 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} ; rev ; md$.

But, if we take a closer look at figure 2, we immediately see that we may have a problem in indicating the correct connections. In other words,we wish to combine the three required properties so that we can write

$$\left(\left(1 \otimes 1_{\mathcal{L}^*\text{-}\mathcal{L}^1}\right) \otimes 1\right) ; recmb ; rev = \tag{18}$$
$$\left(\left(1 \otimes 1_{\mathcal{L}^*\text{-}\mathcal{L}^1}\right) \otimes 1\right) ; \left((\Pi_{lst} \nabla (\Pi_{md} ; rev)) \nabla \Pi_{hd}\right) ; recmb$$

But, from the nesting of parentheses in the term involving only identites, we see that the *extraction functions* Π_{lst}, Π_{md} and Π_{hd} involve certain amount of *deparenthesizing* To see how they can be constructed, let us analyze the diagram in figure 3.

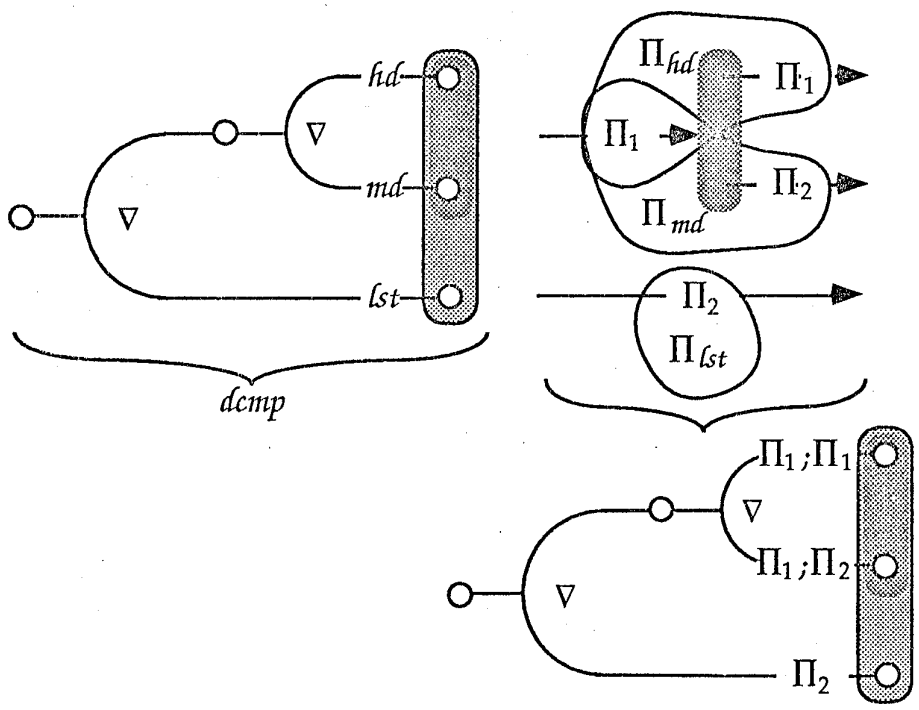

Figure 3: Connecting term for the eureka.

From this diagram it is easy to see that the required expressions for the extraction functions are as follows (we shall reexamine this construction in a general setting later on in 8.2):

$\Pi_{hd} = \Pi_1 \; ; \; \Pi_1$
$\Pi_{md} = \Pi_1 \; ; \; \Pi_2$
$\Pi_{lst} = \Pi_2$

Hence, the explicit form of (18) will be

$$\left(\left(1 \otimes 1_{\mathcal{L}^* - \mathcal{L}^1}\right) \otimes 1\right) \; ; \; rcmb \; ; \; rev = \tag{19}$$
$$\left(\left(1 \otimes 1_{\mathcal{L}^* - \mathcal{L}^1}\right) \otimes 1\right) \; ; \; \left(\left(\Pi_2 \nabla \left((\Pi_1 \; ; \; \Pi_2) \; ; \; rev\right)\right) \nabla (\Pi_1 \; ; \; \Pi_1)\right) \; ; \; rcmb$$

We should recognize here that is not very comfortable to be forced to deal with this *LISP-like* parenthetization. So, we will assume the nesting of the terms over identities in (18) (see also the diagram in figure 3) as canonical, in which case we will leave implicit the parentheses in fork and direct product terms. Thus, we write (18) in the simpler form

$$\left(1 \otimes 1_{\mathcal{L}^* - \mathcal{L}^1} \otimes 1\right) \; ; \; recmb \; ; \; rev = \tag{20}$$
$$\left(1 \otimes 1_{\mathcal{L}^* - \mathcal{L}^1} \otimes 1\right) \; ; \; \left(\Pi_{lst} \nabla \Pi_{md} \; ; \; rev \nabla \Pi_{hd}\right) \; ; \; recmb$$

Now, by unfolding (19) into (17) and (17) and (16) into (15), we obtain

$$1_{\mathcal{L}^* - \mathcal{L}^1} \; ; \; 2 \; ; \; (1 \otimes rev) \; ; \; \ddot{1} = \left[\left((hd \nabla md \nabla lst) \; ; \; \overset{\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim}{(hd \nabla md \nabla lst)}\right) \nabla \right.$$
$$\left.\left((hd \nabla md \nabla lst) \; ; \; (\Pi_{lst} \nabla \Pi_{md} \; ; \; rev \nabla \Pi_{hd}) \; ; \; \overset{\sim\sim\sim\sim\sim\sim\sim\sim\sim\sim}{(hd \nabla md \nabla lst)}\right)\right] \; ; \; \ddot{1}$$

But the very goal of the extraction functions is the following, easily verified, property:

$$(hd \nabla md \nabla lst) \; ; \; (\Pi_{lst} \nabla \Pi_{md} \; ; \; rev \nabla \Pi_{hd}) = (lst \nabla md \; ; \; rev \nabla hd)$$

So,

$$1_{\mathcal{L}^* - \mathcal{L}^1} \; ; \; 2 \; ; \; (1 \otimes rev) \; ; \; \ddot{1} =$$
$$\left((hd \nabla md \nabla lst) \nabla (lst \nabla md \; ; \; rev \nabla hd)\right) \; ; \; (rcmb \otimes rcmb) \; ; \; \ddot{1}$$

Now, let us call, $eql = \left(1_{\mathcal{L}^* - \mathcal{L}^1} \otimes 1_{\mathcal{L}^* - \mathcal{L}^1}\right) \; ; \; \ddot{1}$ and $smc = \left(1_C \otimes 1_C\right) \; ; \; \ddot{1}$. Then, the property of lists: for all $a_1, a_2 \in C \wedge x_1, x_2 \in \mathcal{L}^*$

$$x_1\ eql\ x_2 \wedge a_1\ smc\ a_2 \leftrightarrow\ cons(a_1, x_1) = cons(a_2, x_2)$$

can be written in a relational form as:

$$(smc \otimes eql)\ ;\ and = 2\ ;\ t\ ;\ (cns \otimes cns)\ ;\ eql$$

where t is the *connecting term* for taking care of *formal noise* introduced above. As it is shown in the diagram in figure 4, it is very easy to generalize this property to match our case. In this diagram we use the symbol ● for elements of C, the symbol ■ for elements of $\mathcal{L}^*$–$\mathcal{L}^1$ and the symbol ◆ for the sort *Boolean*.

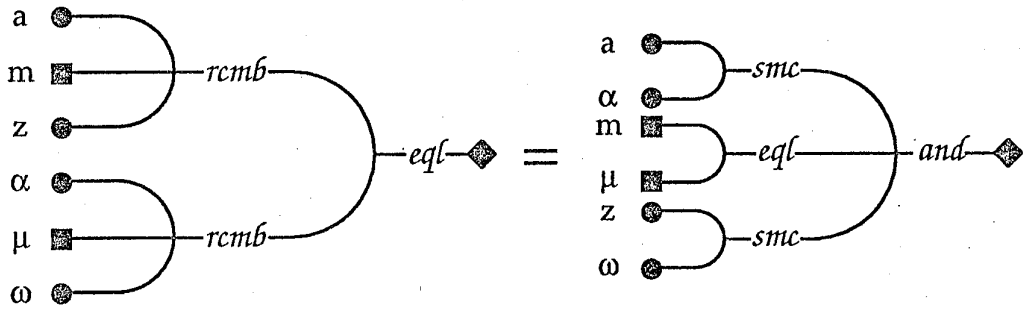

Figure 4: Equality of decomposed lists.

If we use, for the sake of clarity, the two-dimensional notation, we can write this property as follows

$$\begin{pmatrix} rcmb \\ \otimes \\ rcmb \end{pmatrix};\ eql = \begin{bmatrix} (\Pi_1\ ;\ \Pi_h \nabla \Pi_1\ ;\ \Pi_h)\ ;\ smc \\ \nabla \\ (\Pi_1\ ;\ \Pi_m \nabla \Pi_1\ ;\ \Pi_m)\ ;\ eql \\ \nabla \\ (\Pi_1\ ;\ \Pi_l \nabla \Pi_1\ ;\ \Pi_l)\ ;\ smc \end{bmatrix};\ and$$

Now, for any pair of relations p and q, we have

$$(p \nabla q)\ ;\ \begin{pmatrix} rcmb \\ \otimes \\ rcmb \end{pmatrix};\ eql = \begin{bmatrix} (p\ ;\ \Pi_h \nabla q\ ;\ \Pi_h)\ ;\ smc \\ \nabla \\ (p\ ;\ \Pi_m \nabla q\ ;\ \Pi_m)\ ;\ eql \\ \nabla \\ (p\ ;\ \Pi_l \nabla q\ ;\ \Pi_l)\ ;\ smc \end{bmatrix};\ and$$

But, notice that $dcmp\ ;\ \Pi_h = hd$, $dcmp\ ;\ \Pi_m = md$ and $dcmp\ ;\ \Pi_l = lst$. Thus

$$1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \;;\; \dot{pal} = \begin{bmatrix} (hd \nabla lst) \;;\; smc \\ \nabla \\ (md \nabla md \;;\; rev) \;;\; eql \\ \nabla \\ (lst \nabla hd) \;;\; smc \end{bmatrix} \;;\; and$$

Now, it is easy to see (as we shall examine more generally in 8.2) that $lst \nabla hd = (hd \nabla lst) \;;\; (\Pi_2 \nabla \Pi_1)$, and from the general property of *filters over equality* $(\Pi_2 \nabla \Pi_1) \;;\; \ddot{I} = (\Pi_1 \nabla \Pi_2) \;;\; \ddot{I} = \ddot{I}$ (see 7.3 below), we have $(\Pi_2 \nabla \Pi_1) \;;\; smc = (\Pi_1 \nabla \Pi_2) \;;\; smc = smc$. So

$$1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \;;\; \dot{pal} = \begin{bmatrix} (hd \nabla lst) \;;\; smc \\ \nabla \\ (md \nabla md \;;\; rev) \;;\; eql \\ \nabla \\ (hd \nabla lst) \;;\; smc \end{bmatrix} \;;\; and$$

Now, by applying, the property of a∧(b∧b)=a∧b (which we write relationally as $(p \nabla q \nabla q) \;;\; and = (p \nabla q) \;;\; and$), we can write

$$1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \;;\; \dot{pal} = \begin{bmatrix} (hd \nabla lst) \;;\; smc \\ \nabla \\ md \;;\; \underbrace{(1 \nabla rev) \;;\; eql}_{\dot{pal}} \end{bmatrix} \;;\; and$$

Therefore, we obtain

$$\dot{pal} = 1_{\mathcal{L}^1} \;;\; true + 1_{\mathcal{L}^*\text{-}\mathcal{L}^1} \;;\; \begin{bmatrix} (hd \nabla lst) \;;\; smc \\ \nabla \\ md \;;\; \dot{pal} \end{bmatrix} \;;\; and$$

as an expression for a (recursive) program for palindrome.

6.2. TRIVIALIZATION

The idea behind trivialization is restricting the problem to a set of inputs where one already knows its behavior. We shal now briefly examine this idea in general in the context of deriving a characteristic predicate for a subset B of a given set A. In view of the above results in 5.3.3, we may assume that the condition for membership of an element of A in B is given by a function Φ.

Consider a set $P \subseteq A$, where $1_P \; ; \; \Phi = \tau$ (known and acceptable). In other words, we already have an acceptable program for the pre-restriction to the subset P.

Then, we can decompose

$$1_A = 1_P + 1_{P-\mathcal{A}}$$

$$B_A = 1_P \; ; \; B_A + 1_{P-\mathcal{A}} \; ; \; B_A$$

$$1_P \; ; \; B_A = 1_P \; ; \; \left(1_A \bullet \Phi\right) \; ; \; true + 1_P \; ; \; \left(1_A \bullet \overline{\Phi}\right) \; ; \; false$$

But, as in 5.3.3, we have

$$1_P \; ; \; \left(1_A \bullet \Phi\right) = \left(1_P \; ; \; \Phi\right) \bullet 1_A = \tau \bullet 1_A$$

$$1_P \; ; \; \left(1_A \bullet \overline{\Phi}\right) = \left(\underbrace{1_P \; ; \; \Phi}_{\tau} \; ; \; \overline{1} \right) \bullet 1_A = \left(\tau \; ; \; \overline{1}\right) \bullet 1_A$$

Hence, we can write

$$1_P \; ; \; B_A = \left(\tau \bullet 1_A\right) \; ; \; true + \left(\left(\tau \; ; \; \overline{1}\right) \bullet 1_A\right) \; ; \; false$$

Finally, by reusing (14), we can arrive at

$$1_P \; ; \; B_A = 1_A \; ; \; \left(1 \nabla \tau\right) \; ; \; \ddot{1}$$

In particular, in case $1_P \; ; \; \left(1_A \bullet \overline{\Phi}\right) = 0$, we have

$$1_P \; ; \; B_A = \left(\tau \bullet 1_A\right) \; ; \; true.$$

7. INTERNALIZATION OF RELATIONS

A test over a relation is meant to check whether an input-output pair belongs to the relation. In other words, it is supposed to receive x and y as inputs, and output **T** or **F**

accordingly. In view of our universe strucutre, we can code up the input-output pair as [x,y]. In addition, we can internalize a relation by means of its filter, which turns out to be quite useful for derivation purposes. A filter (relativized identity), as already seen, represents :the set of objects that exhibit a given property. As auxiliary tools, as well as alternative internalizations, we also introduce semi-filters over relations.

7.1. SEMI-FILTERS AND FILTERS OVER RELATIONS

Given a relation r, by its *semi-filter-to-domain* we mean the relation

$$\vec{r} = \{\langle [x,y], x\rangle : x\, r\, y\} \tag{19}$$

Notice that $\mathcal{R}an(\vec{r}) = \mathcal{D}om(r)$. (20)

Given a relation r, we analogously define its *semi-filter-to-range* as the relation

$$\vec{r} = \{\langle [x,y], y\rangle : x\, r\, y\} \tag{21}$$

Notice that $\mathcal{R}an(\vec{r}) = \mathcal{R}an(r)$. (22)

Notice also that $[x,y] \in \mathcal{D}om(\vec{r}) \leftrightarrow [x,y] \in \mathcal{D}om(\vec{r}) \leftrightarrow x\, r\, y$. (23)

Let us now write relational expressions for each *semi-filter*. It is clear that

$\tilde{\vec{r}} = \{\langle x, [x,y]\rangle : x\, r\, y\}$. Thus, $\tilde{\vec{r}} = 1 \nabla r$, and $\vec{r} = \tilde{1} \nabla r$ (24)

Analogously,

$\tilde{\vec{r}} = \{\langle y, [x,y]\rangle : x\, r\, y\}$. Thus, $\tilde{\vec{r}} = r \nabla 1$, and $\vec{r} = \tilde{r} \nabla 1$ (25)

Now, given a relation r, we define its *filter* as the relation

$$\vec{r} = \{\langle [x,y], [x,y]\rangle : x\, r\, y\} \tag{26}$$

Now, we can express the filters of a relation in terms of each one of its semi-filters:

$$\vec{r} = \left(\vec{r};\ \tilde{\vec{r}}\right) \circ 1 = \left(\vec{r};\ \tilde{\vec{r}}\right) \circ 1 \tag{27}$$

7.2. TESTS OVER RELATIONS

Let us define *test* over the relation r as the relation

$$\ddot{r} = \left\{\left\langle [x,y], \mathbf{b}\right\rangle : \left(\mathbf{b} = \mathbf{T} \leftrightarrow x\, r\, y\right) \wedge \left(\mathbf{b} = \mathbf{F} \leftrightarrow x\, \bar{r}\, y\right)\right\} \tag{28}$$

Notice that, in view of (5), we can write,

$$\ddot{r} = \left\{\left\langle [x,y], \mathbf{b}\right\rangle : \left(\mathbf{b} = \mathbf{T} \leftrightarrow [x,y] \in \mathcal{D}om(\bar{r})\right) \wedge \left(\mathbf{b} = \mathbf{F} \leftrightarrow [x,y] \in \mathcal{D}om(\bar{\bar{r}})\right)\right\} \tag{29}$$

It is easy to see that, for every relation r, $\mathcal{D}om(r;\ true) = \mathcal{D}om(r;\ false) = \mathcal{D}om(r)$, $\mathcal{R}an(r;\ true) = \mathcal{R}an(true)$ and $\mathcal{R}an(r;\ false) = \mathcal{R}an(false)$. So, we can write the following, nondeterministic, relational expression for the filter

$$\ddot{r} = \left(\bar{r} + \vec{r}\right);\ true + \left(\bar{\bar{r}} + \vec{\vec{r}}\right);\ false \tag{30}$$

And, since $\bar{r} \bullet \vec{r} = 0$ and $\bar{\bar{r}} \bullet \vec{\vec{r}} = 0$, any one of the following expressions can be used as a relational definition for $\ddot{r}$

$$\ddot{r} = \bar{r};\ true + \bar{\bar{r}};\ false \tag{31}$$

$$\ddot{r} = \bar{r};\ true + \vec{\vec{r}};\ false \tag{32}$$

$$\ddot{r} = \vec{r};\ true + \bar{\bar{r}};\ false \tag{33}$$

$$\ddot{r} = \vec{r};\ true + \vec{\vec{r}};\ false \tag{34}$$

7.3. FILTERS AND TESTS OVER EQUALITY

Let us now specialize the preceding development to the quite important case of equality. We shall construct the filter and test for "equality", i.e., for the relation *1*. Since $1 = \tilde{1}$, in view of (34) we can write,

$$\bar{1} = \tilde{1}\tilde{\nabla}\tilde{1} = \vec{1} = \tilde{2} \tag{35}$$

Analogously, recalling that $\wp = \bar{1}$, we can write,

$$\bar{\wp} = \tilde{1}\tilde{\nabla}\wp \text{ and } \vec{\wp} = \tilde{\wp}\tilde{\nabla}\tilde{1} \tag{36}$$

which are evidently not equal.

In view of (30), we can write the following relational expression for the test over equality:

$$\ddot{1} = \bar{1} \; ; \; true + (\bar{\wp} + \vec{\wp}) \; ; \; false \tag{37}$$

And, since $\bar{\wp} \circ \vec{\wp} = 0$, any of the following expressions can be used as a relational definition for $\ddot{1}$,

$$\ddot{1} = \bar{1} \; ; \; true + \bar{\wp} \; ; \; false \tag{39}$$

$$\ddot{1} = \bar{1} \; ; \; true + \vec{\wp} \; ; \; false \tag{40}$$

We shall generally use

$$\ddot{1} = \tilde{2} \; ; \; true + \tilde{\tilde{1}} \tilde{\nabla} \tilde{1} ; \; false$$

Notice that any one of the above expressions gives the expected behavior of the equality test. Namely

$[x,y]\ddot{1}\mathbf{T} \leftrightarrow x = y$

$[x,y]\ddot{1}\mathbf{F} \leftrightarrow x \neq y$

7.4. RETRIEVING THE RELATION

We have seen how a relation can be internalized by means of its semi-filters and filter. Now, let us show how one can recover the relation from its internalized versions.

First, consider the case of retrieving a relation from its semi-filter to domain. We have

$$\tilde{\tilde{r}} = 1 \nabla r = 2 \; ; (1 \otimes r).$$

Thus, we are looking for some relation s such that $\bar{r}; s = r$. So, it would be natural to proceed as in the reversal example. We shall, however, illustrate how one can proceed by some simple algebraic manipulations.

$$r = \infty \bullet r \qquad \text{(by intersection elimination, Theorem 5.3)}$$

$$= 2 \; ; \begin{bmatrix} \infty \\ \otimes \\ r \end{bmatrix} \; ; \; \tilde{2} \qquad \text{(by property of identity)}$$

$$= 2 \; ; \begin{bmatrix} 1 \; ; \; \infty \\ \otimes \\ r \; ; \; 1 \end{bmatrix} \; ; \; \tilde{2} \qquad \text{(by distributivity property)}$$

$$= 2 \; ; \underbrace{\begin{bmatrix} 1 \\ \otimes \\ r \end{bmatrix}}_{\tilde{\tilde{r}}} \; ; \underbrace{\begin{bmatrix} \infty \\ \otimes \\ 1 \end{bmatrix}}_{\Pi_2} \; ; \; \tilde{2} \quad \text{(by definition)}$$

$$= \tilde{\tilde{r}} \; ; \; \Pi_2$$

Now, the case of retrieving a relation from its semi-filter to range is similar. We have

$$r = r \bullet \infty = 2 \; ; \begin{bmatrix} r \\ \otimes \\ \infty \end{bmatrix} \; ; \; \tilde{2} = 2 \; ; \begin{bmatrix} 1 \; ; \; r \\ \otimes \\ \infty \; ; \; 1 \end{bmatrix} \; ; \; \tilde{2}$$

$$= 2 \; ; \begin{bmatrix} 1 \\ \otimes \\ \infty \end{bmatrix} \; ; \begin{bmatrix} r \\ \otimes \\ 1 \end{bmatrix} \; ; \; \tilde{2} \; = \tilde{\Pi}_1 \; ; \; \vec{r}$$

Finally, the case of retrieving a relation from its filter is now quite simple. We can easily derive $r = \tilde{\Pi}_1 \; ; \; \vec{r} \; ; \; \Pi_2$.

8. OVERCOMING THE NESTING "FORMAL NOISE"

We have already mentioned that our formalism is no exception, in that it has a considerable amount of drudgery to be taken care of. In the preceding sections, we have illustrated how one can cope with it, once one gets the knack of it. We shall now indicate how a reasonable amount of this "formal noise" can be overcome in a general manner, which is amenable to automation, so that one can let the system take care of these matters. We shall now indicate the general treatment of sort information, and connecting (or rearranging) terms. Both matters have already been hinted at in the preceding developments, the former at several places and the latter during the derivation of palindrome in 6.1.

8.1 Sorts, Types and Conditions

We shall now show how one can use the input sorts of a relational term to predict some information concerning its output sorts.

Consider the following situation, similar to the one encoutered in deriving reversal in 4.1.

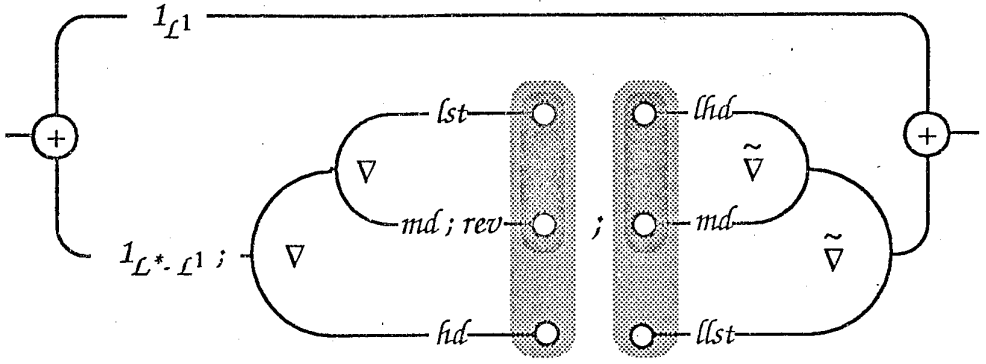

Figure 5: Program segment for list reversal.

We have the following relational expression for this program segment

$$rev = 1_{\mathcal{L}^1} + 1_{\mathcal{L}^*\cdot\mathcal{L}^1}\,;\begin{pmatrix} llst \\ \nabla \\ md\,;\,rev \\ \nabla \\ lhd \end{pmatrix};\begin{pmatrix} lhd \\ \tilde{\nabla} \\ md \\ \tilde{\nabla} \\ llst \end{pmatrix}$$

Now, consider some sort information 1_{Ξ}, say Ξ is the set of all lists containing only characters. We can pre-restrict our program segment to this set by pre-multiplying its expression by the corresponding identity filter 1_{Ξ}. We thus obtain

$$1_{\Xi}\,;\left[1_{\mathcal{L}^1} + 1_{\mathcal{L}^*\cdot\mathcal{L}^1}\,;\begin{pmatrix} llst \\ \nabla \\ md\,;\,rev \\ \nabla \\ lhd \end{pmatrix};\begin{pmatrix} lhd \\ \tilde{\nabla} \\ md \\ \tilde{\nabla} \\ llst \end{pmatrix}\right]$$

We can now propagate this pre-restriction to the right by means of the following steps.

First, we obtain

$$1_{\Xi}\,;\,1_{\mathcal{L}^1}+1_{\Xi}\,;\,1_{\mathcal{L}^*-\mathcal{L}^1}\,;\begin{pmatrix}\mathcal{lst}\\ \nabla\\ \mathcal{md}\,;\,\mathcal{rev}\\ \nabla\\ \mathcal{lhd}\end{pmatrix};\begin{pmatrix}\mathcal{lhd}\\ \tilde{\nabla}\\ \mathcal{md}\\ \tilde{\nabla}\\ \mathcal{lst}\end{pmatrix}$$

by applying the fact that $1_{\chi}\,;\,(p+q)=\left(1_{\chi}\,;\,p+1_{\chi}\,;\,q\right)$.

Now, since $1_{\chi}\,;\,1_{\mathcal{L}^1}=1_{\mathcal{L}^1}\,;\,1_{\chi}$ and $1_{\chi}\,;\,1_{\mathcal{L}^*-\mathcal{L}^1}=1_{\mathcal{L}^*-\mathcal{L}^1}\,;\,1_{\chi}$, we obtain

$$1_{\mathcal{L}^1}\,;\,1_{\Xi}+1_{\mathcal{L}^*-\mathcal{L}^1}\,;\,1_{\Xi}\,;\begin{pmatrix}\mathcal{lst}\\ \nabla\\ \mathcal{md}\,;\,\mathcal{rev}\\ \nabla\\ \mathcal{lhd}\end{pmatrix};\begin{pmatrix}\mathcal{lhd}\\ \tilde{\nabla}\\ \mathcal{md}\\ \tilde{\nabla}\\ \mathcal{lst}\end{pmatrix}$$

Similarly, we can replace

$$1_{\Xi}\,;\begin{pmatrix}\mathcal{lst}\\ \nabla\\ \mathcal{md}\,;\,\mathcal{rev}\\ \nabla\\ \mathcal{lhd}\end{pmatrix};\begin{pmatrix}\mathcal{lhd}\\ \tilde{\nabla}\\ \mathcal{md}\\ \tilde{\nabla}\\ \mathcal{lst}\end{pmatrix}\quad\text{by}\quad\begin{pmatrix}1_{\Xi}\,;\,\mathcal{lst}\\ \nabla\\ 1_{\Xi}\,;\,\mathcal{md}\,;\,\mathcal{rev}\\ \nabla\\ 1_{\Xi}\,;\,\mathcal{lhd}\end{pmatrix};\begin{pmatrix}\mathcal{lhd}\\ \tilde{\nabla}\\ \mathcal{md}\\ \tilde{\nabla}\\ \mathcal{lst}\end{pmatrix}$$

Now, since $\mathcal{lhd}$, $\mathcal{md}$ and $\mathcal{lst}$ are assumed to be known, we can get some information about their outputs over set Ξ. In our specific example, we can write $1_{\Xi}\,;\,\mathcal{lst}=\mathcal{lst}\,;\,1_{\Xi}$, $1_{\Xi}\,;\,\mathcal{md}=\mathcal{md}\,;\,1_{\Xi}$ and $1_{\Xi}\,;\,\mathcal{lhd}=\mathcal{lhd}\,;\,1_{\Xi}$. Hence we can now replace

$$\begin{pmatrix}1_{\Xi}\,;\,\mathcal{lst}\\ \nabla\\ 1_{\Xi}\,;\,\mathcal{md}\,;\,\mathcal{rev}\\ \nabla\\ 1_{\Xi}\,;\,\mathcal{lhd}\end{pmatrix}\quad\text{by}\quad\begin{pmatrix}\mathcal{lst}\,;\,1_{\Xi}\\ \nabla\\ \mathcal{md}\,;\,1_{\Xi}\,;\,\mathcal{rev}\\ \nabla\\ \mathcal{lhd}\,;\,1_{\Xi}\end{pmatrix}$$

By proceeding in a similar manner, we can arrive at replacing

$$\begin{pmatrix} 1_{\Xi} ; \mathit{lst} \\ \nabla \\ 1_{\Xi} ; \mathit{md} ; \mathit{rev} \\ \nabla \\ 1_{\Xi} ; \mathit{hd} \end{pmatrix} ; \begin{pmatrix} \mathit{hd} \\ \tilde{\nabla} \\ \mathit{md} \\ \tilde{\nabla} \\ \mathit{lst} \end{pmatrix} \quad \text{by} \quad \begin{pmatrix} \mathit{lst} \\ \nabla \\ \mathit{md} ; \mathit{rev} \\ \nabla \\ \mathit{hd} \end{pmatrix} ; \begin{pmatrix} 1_{\Xi} \\ \otimes \\ 1_{\Xi} \\ \otimes \\ 1_{\Xi} \end{pmatrix} ; \begin{pmatrix} \mathit{hd} \\ \tilde{\nabla} \\ \mathit{md} \\ \tilde{\nabla} \\ \mathit{lst} \end{pmatrix}$$

Eventually, we manage to replace the original expression by

$$\left[1_{\mathcal{L}^1} + 1_{\mathcal{L}^* - \mathcal{L}^1} ; \begin{pmatrix} \mathit{lst} \\ \nabla \\ \mathit{md} ; \mathit{rev} \\ \nabla \\ \mathit{hd} \end{pmatrix} ; \begin{pmatrix} \mathit{hd} \\ \tilde{\nabla} \\ \mathit{md} \\ \tilde{\nabla} \\ \mathit{lst} \end{pmatrix} \right] ; 1_{\Xi}$$

Thus, we can propagate the given input sorts as pre-restriction to any point in the diagram and obtain some information concerning the output sorts, in this case we can actuallly determine them.

The promotion of pre-retrictions and post-restrictions over the operations can be carried out by means of the following set of rules.

Rules for promoting restrictions

	Prerestrictions	**Post-restrictions**	
$+$	$1_X ; \begin{pmatrix} p \\ + \\ q \end{pmatrix} = \begin{pmatrix} 1_X ; p \\ + \\ 1_X ; q \end{pmatrix}$	$\begin{pmatrix} p ; 1_X \\ + \\ q ; 1_X \end{pmatrix} = \begin{pmatrix} p \\ + \\ q \end{pmatrix} ; 1_X$	
$\bullet$	$(1_X \bullet 1_Y) ; \begin{pmatrix} p \\ \bullet \\ q \end{pmatrix} = \begin{pmatrix} 1_X ; p \\ \bullet \\ 1_Y ; q \end{pmatrix}$	$\begin{pmatrix} p ; 1_X \\ \bullet \\ q ; 1_Y \end{pmatrix} = \begin{pmatrix} p \\ \bullet \\ q \end{pmatrix} ; (1_X \bullet 1_Y)$	‡
0	$1_X ; 0 = 0$	$0 ; 1_X = 0$	
$-$	$1_X ; (\infty - p) = (1_X ; \infty) - (1_X ; p)$	$(\infty - p) ; 1_X = (\infty ; 1_X) - (p ; 1_X)$	
$\sim$	$1_X ; \tilde{p} = \tilde{p}$ iff $\tilde{p} = \tilde{p} ; 1_Y$		
∇	$(1_X \bullet 1_Y) ; \begin{pmatrix} p \\ \nabla \\ q \end{pmatrix} = \begin{pmatrix} 1_X ; p \\ \nabla \\ 1_Y ; q \end{pmatrix}$	$\begin{pmatrix} p ; 1_X \\ \nabla \\ q ; 1_Y \end{pmatrix} = \begin{pmatrix} p \\ \nabla \\ q \end{pmatrix} ; \begin{pmatrix} 1_X \\ \otimes \\ 1_Y \end{pmatrix}$	‡

$\otimes$	$\begin{pmatrix} 1_X \\ \otimes \\ 1_Y \end{pmatrix} ; \begin{pmatrix} p \\ \otimes \\ q \end{pmatrix} = \begin{pmatrix} 1_X ; p \\ \otimes \\ 1_Y ; q \end{pmatrix}$	$\begin{pmatrix} p ; 1_X \\ \otimes \\ q ; 1_Y \end{pmatrix} = \begin{pmatrix} p \\ \otimes \\ q \end{pmatrix} ; \begin{pmatrix} 1_X \\ \otimes \\ 1_Y \end{pmatrix}$	
$\tilde{\nabla}$	$\begin{pmatrix} 1_X ; p \\ \tilde{\nabla} \\ 1_X ; p \end{pmatrix} = \begin{pmatrix} p \\ \tilde{\nabla} \\ q \end{pmatrix} ; (1_X \bullet 1_Y)$		‡
$\tilde{2}$	$\begin{pmatrix} 1_X \\ \otimes \\ 1_X \end{pmatrix} ; \tilde{2} = 2 ; (1_X \bullet 1_Y)$		

The propagation of information about the restrictin rests on the above distributivity rules. The crucial step is propagating the pre-condition over the basic relations, when we need some commutativity property. The more information we are able to give concerning the sort of its outputs, the better. In our preceding example, the equations $1_X ; \mathit{lst} = \mathit{lst} ; 1_X$, $1_X ; \mathit{md} = \mathit{md} ; 1_X$ and $1_X ; \mathit{hd} = \mathit{hd} ; 1_X$ do provide quite accurate information about the outputs, even though there is room for some improvement.

Now, consider another pre-condition 1_Ξ, say Ξ is the set of all lists containing character a. We start from

$$1_\Xi ; \left[1_{\mathcal{L}^1} + 1_{\mathcal{L}^* - \mathcal{L}^1} ; \begin{pmatrix} \mathit{lst} \\ \nabla \\ \mathit{md} ; \mathit{rev} \\ \nabla \\ \mathit{hd} \end{pmatrix} ; \begin{pmatrix} \mathit{hd} \\ \tilde{\nabla} \\ \mathit{md} \\ \tilde{\nabla} \\ \mathit{lst} \end{pmatrix} \right]$$

and proceed as in the previous example, until we replace

---

‡ Notice that these rules do not necessarily give information concerning the domain or range of the relations involved. Thus, they do not conflict with the results in theorem 4.4 and remark 4.5 in [Hae91], which show that one cannot determine terminating pre-restrictions independently of their respective programs.

$$1_{\Xi} ; \begin{pmatrix} \mathit{lst} \\ \nabla \\ \mathit{md} ; \mathit{rev} \\ \nabla \\ \mathit{hd} \end{pmatrix} ; \begin{pmatrix} \mathit{hd} \\ \tilde{\nabla} \\ \mathit{md} \\ \tilde{\nabla} \\ \mathit{lst} \end{pmatrix} \quad \text{by} \quad \begin{pmatrix} 1_{\Xi} ; \mathit{lst} \\ \nabla \\ 1_{\Xi} ; \mathit{md} ; \mathit{rev} \\ \nabla \\ 1_{\Xi} ; \mathit{hd} \end{pmatrix} ; \begin{pmatrix} \mathit{hd} \\ \tilde{\nabla} \\ \mathit{md} \\ \tilde{\nabla} \\ \mathit{lst} \end{pmatrix}$$

Then, we can get only some information about the outputs of *hd*, *md* and *lst* over the set Ξ. We can only write $1_{\Xi} ; \mathit{lst} = 1_{Y}$, $1_{\Xi} ; \mathit{md} = 1_{Z}$ and $1_{\Xi} ; \mathit{hd} = 1_{V}$, where $1_{Y} + 1_{Z} + 1_{V} = 1_{\Xi}$. Hence, we can now replace

$$\begin{pmatrix} 1_{\Xi} ; \mathit{lst} \\ \nabla \\ 1_{\Xi} ; \mathit{md} ; \mathit{rev} \\ \nabla \\ 1_{\Xi} ; \mathit{hd} \end{pmatrix} \quad \text{by} \quad \begin{pmatrix} \mathit{lst} ; 1_{Y} \\ \nabla \\ \mathit{md} ; 1_{Z} ; \mathit{rev} \\ \nabla \\ \mathit{hd} ; 1_{V} \end{pmatrix}$$

By proceeding in a similar manner, we eventually manage to replace the original expression by

$$\left[1_{\mathcal{L}^1} + 1_{\mathcal{L}^*-\mathcal{L}^1} ; \begin{pmatrix} \mathit{lst} \\ \nabla \\ \mathit{md} ; \mathit{rev} \\ \nabla \\ \mathit{hd} \end{pmatrix} ; \begin{pmatrix} \mathit{hd} \\ \tilde{\nabla} \\ \mathit{md} \\ \tilde{\nabla} \\ \mathit{lst} \end{pmatrix} \right] ; \left(1_{M} \bullet 1_{N} \circledcirc 1_{P} \right)$$

together with some information relating 1_{M}, 1_{N}, and 1_{P}, to 1_{Ξ}.

We can now indicate in general terms when we can propagate conditions without losing information. We wish to distinguish between our two examples above. For this purpose, let us agree to call *input sort for a basic relation* r, a filter 1_{X} that commutes with r, in the sense that there exists a term $1_{c(r,1_X)}$ such that $r = r ; 1_{c(r,1_X)}$. We now call an *input sort for a relational term* t a filter 1_{X} that is an input sort for every basic relation r occurring in t. We then see that, given an an input sort 1_{X} for a relational term t, we can determine its output sort, i. e. a relational term $1_{c(t,1_X)}$ such that $1_{M}, \ldots, 1_{P}\ t = t ; 1_{c(t,1_X)}$.

In constrast, for an arbitrary pre-condition 1_X, we may have to be content with determining some post-restrictions $1_X\ 1_M, \ldots, 1_P$ together with a relational term $i(1_M, \ldots, 1_P, 1_X)$ giving some some information relating 1_M, 1_N, and 1_P, to 1_X

8.2 CONNECTING TERMS

Consider the following situation, similar to the one encoutered in deriving palindrome in 6.1.

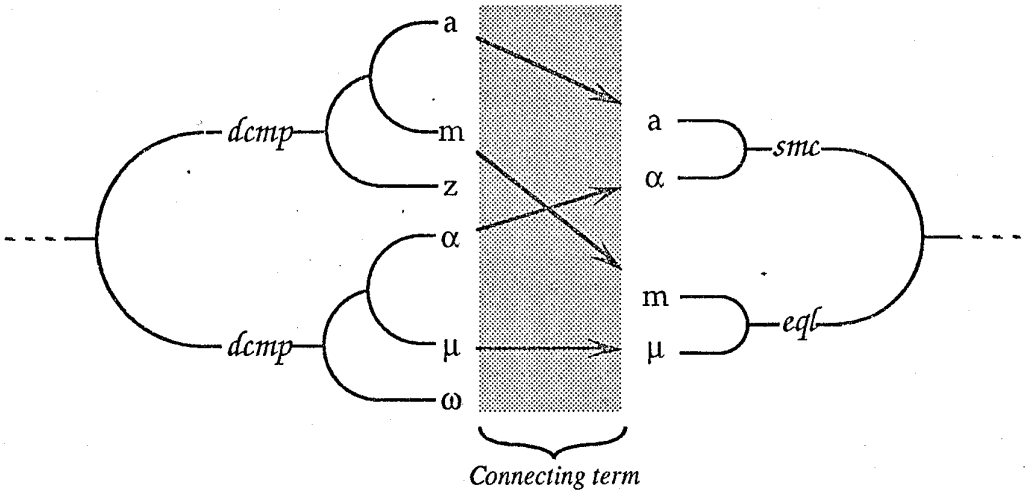

Figure 6: Connecting term for rearrangement.

Here one has a program segment, consisting of two *dcmp*'s, which outputs a composite object $s = [[[a,m],z],[[\alpha,\mu],\omega]]$. One wishes to rearrange this composite object into $t = [[a,\alpha],[m,\mu]]$ so as to feed it into the final program segment, consisting of *smc* and *eql*.

Ideally, one should not bother about such details. One would like to indicate the connecting lines, say by clicking at both extremes, and proceed with the derivation. For this purpose, the system should be able compute a connecting term from the available information: *what is connected to what.*

We shall now indicate how this can be done.

The structure of the first object is a tree like

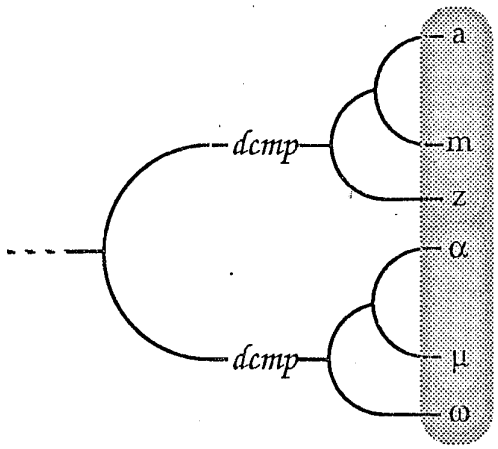

Figure 7: Strucure of the term to be rearranged.

Thus, it is easy to write expressions for each one of its leaves:

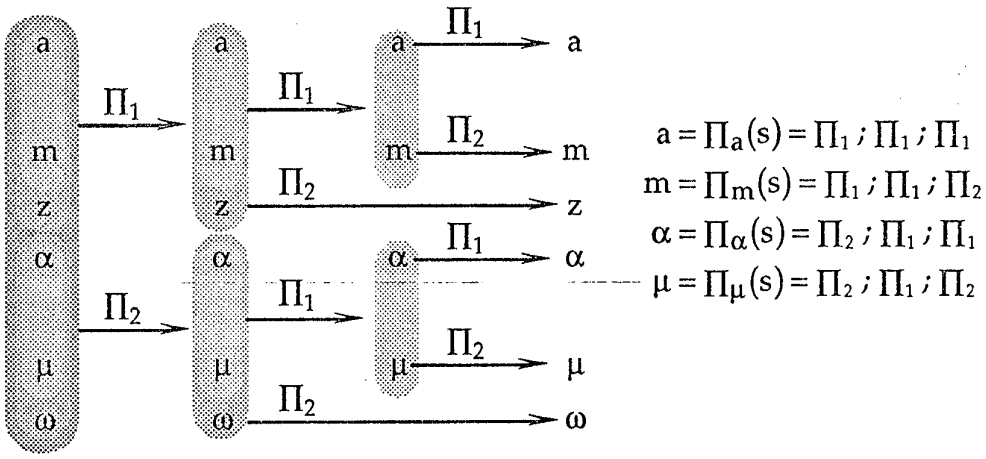

Figure 8: Extracting the components of a structured object.

The structure of the second object looks like

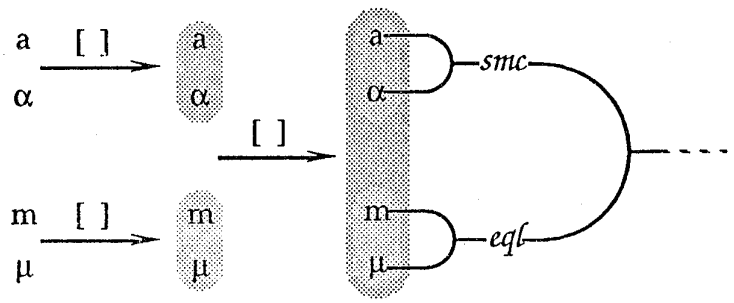

Figure 9: Strucure of the rearranged term.

Now, we can write it in terms of the first object, by putting together the components a, m, α, and μ into the desired template, as follows

$$t = \left[\left[\Pi_a(s), \Pi_\alpha(s)\right], \left[\Pi_m(s), \Pi_\mu(s)\right]\right]$$

It is not difficult to see how one can generalize the above construction. Indeed, given any composite object s, with a tree-like structure, one can write a term $\Pi_{x_k}(s)$, consisting of a composition of projections, to extract its kth leaf. On the other hand, given any composite object t, with a tree-like structure, where the desried content of each leaf is indicated, one can write a term $K(t)$ that constructs the result of inserting this information into its desired place.

One way to formalize the above construction by employing variables to refer to the leaves of such trees. First, recall that our universe $\mathcal{U}$ is structured by the pair-constructing operation $*$ with $u * v = [u, v]$. Thus, the structure of each object can be represented by a term constructed form (distinct) variables and the operation symbol $*$, i. e., an element of the corresponding free term algebra. For instance, the structure of our above objects would be as follows: s represented by $\left(\left(\left(x_1 * x_2\right) * x_3\right) * \left(\left(x_4 * x_5\right) * x_6\right)\right)$ and t by $\left(\left(x_1 * x_2\right) * \left(x_3 * x_4\right)\right)$.

Now, given such a term s, with m distinct variables without repeated occurrences, and an index k with k between 1 and m, we can write a term $\Pi_{x_k}(s)$, consisting of a composition of projections, that extracts of the kth leaf of any object with this structure. More precisely, the denotation of $\Pi_{x_k}(s)$ is the relation

$$\left\{\left\langle s \left\|u_1,\ldots,u_k,\ldots,u_m\right\|, u_k\right\rangle : u_1,\ldots,u_k,\ldots,u_m \in \mathcal{U}\right\}$$

Here, $s \left\|u_1,\ldots,u_k,\ldots,u_m\right\|$ is the value of the term s when its variables $x_1,\ldots,x_k,\ldots x_m$ are assigned the values $u_1,\ldots,u_k,\ldots u_m$, respectively.

We can thus put s into a standard form $\left(x_1 * \ldots * \left(x_k * \ldots * \left(\ldots * x_m\right)\ldots\right)\ldots\right)$, by defining the relation $\Pi(s) = \left(\Pi_{x_1}(s) \nabla \ldots \nabla \left(\Pi_{x_k}(s) \nabla \ldots \nabla \left(\ldots \nabla \Pi_{x_m}(s)\right)\ldots\right)\ldots\right)$, whose denotation is

$$\left\{\left\langle s \left\|u_1,\ldots,u_k,\ldots u_m\right\|, \left(u_1 * \ldots * \left(u_k * \ldots * \left(\ldots * u_m\right)\ldots\right)\ldots\right)\right\rangle : u_1,\ldots,u_k,\ldots u_m \in \mathcal{U}\right\}$$

On the other hand, consider a term $t\left(x_{j_1},\ldots,x_{j_n}\right)$, whose variables $x_1,\ldots,x_n$ have been replaced by $x_{j_1},\ldots,x_{j_n}$, respectively, in order to indicate "what is connected to what". We can now write a term $K(t)$, which puts the values into their places within s. More precisely, the denotation of $K(t)$ is the relation

$$\left\{\left\langle \left(u_1 * \ldots * \left(u_k * \ldots * \left(\ldots * u_m\right)\ldots\right)\ldots\right), t\left\|u_{j_1},\ldots,u_{j_n}\right\|\right\rangle : u_1,\ldots,u_k,\ldots u_m \in \mathcal{U}\right\}$$

Finally, the desired connecting term can be wrtten as $P(s) = \Pi(s)\ ;\ K(t)$. This term indeed constructs the term t from the term s, in the sense that its denotation is

$$\left\{\left\langle s\left\|u_1,\ldots,u_k,\ldots u_m\right\|, t\left\|u_{j_1},\ldots,u_{j_n}\right\|\right\rangle : u_1,\ldots,u_k,\ldots u_m \in \mathcal{U}\right\}$$

Therefore, such a connecting term can always be automatically constructed. Clearly, more efficient algorithms for this task can easily be devised for implementation purposes.

9. CONCLUSIONS

We have reported on an on-going research effort in using an extended version of Tarski's calculus of binary relations for formal program construction. This paper shows that this calculus is adequate for this task.

The expressive power of our extended calculus renders it appropriate for expressing, and reasoning about, programs, as well as strategies and design decisions. Furthermore, the fact that it is based on the single unifying concept of input-output relations over structured universes makes it a truly coherent tool for covering the entire derivation spectrum, from specifications, where expressiveness and ease of expression is important, to programs, where efficient execution is the objective.

This paper has illustrated three main points: the distinction among specification and programming languages and derivation formalisms; that formal specifcation and program construction can, and probably should, be distributed and intertwined along the process; and finally how derivation insights and rationales can be captured within our formalism with the goal of providing machine support for clerical symbolic manipulations.

We have illustrasted that filters (relativized identities) are very useful for deriving programs, as well as for representing sets as binary relations. In 5.3 we have derived some results concernig the specification of sets. In section 7 we have taken one step further and internalized relations by coding input-output pairs. In addition, we have constructed filters and tests over relations and showed how to derive the latter from the former, which appears to be an important rationale. Also, this internalization paves the way for internalizing higher-order concepts.

As regards the automation of clerical symbolic manipulations, we have dealt with two special cases: the "formal noise" of connecting terms and the propagation of restrictions. The latter is a generalization of the usual concept of invariants, an interesting special case being that of types. This has connections, desrving further study, with the view of types as retracts [Sco76].

We are using the results of this paper as a basis for an implementation of our calculus on *Mathematica* ™, with the aim of building an experimental environment for formal program construction.

REFERENCES

[Bac78] Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21, 613-641 (1978)

[Bur80] Burris, D., Sankappanavar, G., *A Course in Universal Algebra*, Springer-Verlag, New York, 1980.

[Chi51] Chin, L. H.,Tarski, A., "Distributive and Modular Laws in the Arithmetic of Relation Algebras"; *Univ. of California Publications in Mathematics, New Series* **1.9** (1951), pp. 341-384.

[Grä71] Grätzer, G. D., *Lattice Theory*, W. H. Freeman, San Francisco, 1971.

[Gut78] Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica 10, 27-52 (1978)

[Hae91] Haeberer, A. M., Veloso P. A. S., Partial Relations for Program Derivation: Adequacy, Inevitability and Expressiveness. In Möller, B., (ed) *Constructing Programs From Specifications*. North Holland, 1991.

[Hoa86] Hoare, C. A. R., Jifeng He, "The Weakest Prespecification", *Fundamenta Informaticae* **9** (1986), pp. 51-84, 217-252.

[Jón52] Jónsson, B.,Tarski, A., "Boolean Algebras with Operators", *Amer. J. of Math.* **74** (1952), pp. 127-162.

[McK40] McKinsey, J. C. C., "Postulates for the Calculus of Binary Relations", *J. of Symbolic Logic* **5.3** (1940), pp. 85-97.

[Sco76] Scott, D., "Data Types as Lattices", *SIAM J. of Computing* **5** (1976), pp. 522-587.

[Tar41] Tarski, A., "On the Calculus of Relations", *J. of Symbolic Logic* **6.3** (1941), pp. 73-89.

[Vel91] Veloso, P. A. S., Haeberer, A. M., "A Finitary Relational Algebra for Classical First-Order Logic", *Bulletin of the Section on Logic of the Polish Academy of Sciences*, **20.2** (1991), pp. 52-62.