Using Relational Methods in Computer Science

ALI JAOUA

PETER KEMPF

GUNTHER SCHMIDT (EDS.)

Bericht Nr. 1998-03
Juli 1998

Universität der Bundeswehr München

Fakultät für

INFORMATIK

Werner-Heisenberg-Weg 39 • D-85577 Neubiberg

Ali Jaoua, Peter Kempf and Gunther Schmidt (eds.)

Using Relational Methods in Computer Science

July 1998

Institut für Softwaretechnologie
Universität der Bundeswehr München
D-85577 Neubiberg

Preface

The motivation of the RelMiCS events since 1994 is to bring together researchers who use the calculus of relations as a conceptual or methodological tool in some aspect of Computer Science.

The first RelMiCS meeting took place in January 1994, at the *Internationales Forschungs- und Begegnungszentrum für Informatik* in Schloß Dagstuhl in the Saarland, Germany, organised by Chris Brink (University of Cape Town) and Gunther Schmidt (Universität der Bundeswehr München). RelMiCS 2 was held by Armando Haeberer (Pontifícia Universidade Católica do Rio de Janeiro) in Paraty, South of Rio de Janeiro, in August 1995. Continuing the established one and a half year rhythm, RelMiCS 3 took place in Hammamet, Tunisia, in January 1997, organised by Ali Jaoua (Universitè de Tunis - II). A fourth event in September 1998 in Stefan Banach Center in Warsaw, Poland, is organized by Ewa Orlowska.

The present report is one of the results of the Hammamet meeting, besides a forthcoming special issue of the journal Information Sciences. Papers have been carefully reviewed. A selection has been made after which papers underwent the corrections suggested.

We are very grateful to all the participants of the Hammamet *RelMiCS 3* meeting on Relational Methods in Computer Science, to the referees, and in particular to the authors of this volume.

Ali Jaoua
Tunis

Peter Kempf, Gunther Schmidt
Munich

Contents

Some Notes on Logic Programming with a Relational Machine 1
Jim Lipton, Emily Chapman

Relational Programming in Libra 35
Barry Dwyer

Modelling Message Buffers with Binary Decision Diagrams 59
Holger Schlingloff

Visiting Some Relatives of Peirce's 71
Michael Böttner

Some Notes on Logic Programming with a Relational Machine

(Extended Abstract)

JAMES LIPTON EMILY CHAPMAN
Dept. of Mathematics, Wesleyan University

Abstract

We study the use of relation calculi for compilation and execution of Horn Clause programs with an extended notion of input and output. We consider various other extensions to the Prolog core.

1 Introduction

Logic programming is programming with predicates in a certain fragment of logic. More broadly speaking, it is programming with executable specifications: code that has independent mathematical meaning consistent with its input-output behavior. Specifications are often formalized as relations. In this paper we explore how Logic Programming itself may be profitably understood, extended and compiled in terms of an underlying equational relational calculus, in which relation variables play a fundamental role, similar in some regards to the role of function variables in the lambda-calculus. The variables in the resulting terms correspond to predicate (second order) variables in the original program, whereas all first order variables are eliminated. Subsequent abstraction provides closed form solutions to all the program predicates.

In this study we use the relation algebra formalism as an executable algebra of logic programs, somewhat in the Backus tradition[Backus]. Our approach here, as in [BroLip, Colp] is to translate logic programs into combinatory relation expressions, which are then executed via rewriting and output-formatting algorithms. It will be useful to illustrate our intentions here with an example.

Consider the following Horn Clause program defining the transitive closure of a graph

```
conn(X,X).                          edge(a,b).
conn(X,Y):-edge(X,Z),conn(Z,Y).     edge(b,c).
                                    edge(a,l).
                                    edge(l,c).
```

and the queries

```
| ?- conn(a,c).
| ?- conn(X,c).
```

This rather carefully chosen example can be easily reformulated in first-order-variable-free relational terms as follows. We introduce the binary relation variables *conn* and *edge* and translate the program into a pair of relation equations. Composition of relations is denoted by "; ".

$$\begin{aligned} edge &= \{(a,b),(b,c),(a,l),(l,c)\} \\ conn &= id \cup (edge; conn) \end{aligned}$$

where id denotes the identity relation. The queries are then represented by the relation expressions:

$$\{(a,c)\} \cap conn \text{ and } (\mathbf{1};\{(c,c)\}) \cap conn$$

where $\mathbf{1}$ is the universal relation $\mathcal{H} \times \mathcal{H}$ and $\mathcal{H}$ is the set of closed terms in the program signature (hence $(\mathbf{1};\{(c,c)\})$ represents the set of all pairs whose second component is c).

Some questions that arise naturally at this point are: can such a translation be defined for an arbitrary Horn Clause program, or even for extensions of conventional programs, what kind of relation calculus is suitable for it, and can the resulting relational expressions be easily executed, controlled and optimised?

The aim of these notes is to sketch out in some detail a relation calculus, a translation, a semantics and a rewriting system that give evidence for an affirmative answer to these questions. The relational abstract machine sketched here is described in detail in [Chap, Ruhlen][1].

A translation that will work for all Horn Clause programs (with equality) is necessarily a bit more involved than the one just shown. The one presented here draws on foundational ideas of Tarski, Givant, Freyd, Maddux, and Broome that have appeared at different times in the relational and allegory-theoretic literature although it is different from all of them in that is it aimed at producing *executable* intermediate relational code. Our work thus builds computation into the relational formalism, and shows how the relation calculus offers a new vehicle for proof search and automated deduction as well.

In the earlier paper [BroLip] cited, the authors showed how the existence of a logically correct executable translation of Horn clause programs follows from results of Tarski and others, and explored other "pure" relational programming formalisms. Here that work is considerably refined, and a simple rewriting system explicitly written out and shown correct.

Combinatory approaches to logic programming have appeared elsewhere in the literature: Bellia and Occhiuto develop an algebra of programs that captures unification, rewriting and narrowing in [BelOcc]. Our contribution is to define a more expressive algebra of logic programs, which admits first-order queries, extensions to equational logic, negation and higher order logic, as well as to show that such an algebra can be found within the relation calculus, a well-understood mathematical formalism which easily incorporates other programming paradigms.

A great deal of work has been done in specification refinement using relational specifications based on Hoare's work [Algebra, Naumann, Backh], as well as on relational approaches to hardware design [BroHut, JonShe, BroJon]. In light of this work, the relational translation described here may provide a new formal link between logic programming, hardware specification, and program synthesis.

The work presented here also owes a substantial debt to the logic program transformation ideas of Clark [Lloyd] and Warren [WAM].

2 A Relation Calculus for Horn Clause Programming

In the following sections we formalize a relational theory and a class of relational structures suitable as a target compilation language for logic programs. The relational theory is close

[1]The machine has been implemented in SWI-Prolog. Check `http://www.cs.wesleyan.edu/~lipton` for details.

in spirit to the positive fragment of the untyped μ-relation calculus MU introduced by deRoever (see e.g. [deRoe, BakRoe, FriasMad]) with additional equations that capture unification requirements over the Herbrand Universe.

2.1 A Language and Meta-language for Horn-clause Programming

We begin by fixing a first-order signature for a Horn Clause program

$$\Sigma = \mathcal{C}_\Sigma \cup \mathcal{F}_\Sigma,$$

where $\mathcal{C}_\Sigma$ is the set of constant symbols of Σ and $\mathcal{F}_\Sigma$ the set of function symbols. We let $\alpha(f)$ denote the arity of the function symbol f. Let $\mathcal{T}_\Sigma$ be the set of closed terms over Σ. We fix a countable set of variables $\mathcal{X}$(*not* to be confused with relation variables introduced in a later section) and let $\mathcal{T}_\Sigma(\mathcal{X})$ be the set of open terms over $\mathcal{X}$.

Pairing

We will need to formalize an object-level pairing function, in order to account for the coding of vectors of terms in our relational language, that is to say, explicit vectors of terms manipulated by the compiler. This is done by adding a second, top-level tier to our concept of term and signature.

Many efficiencies in the translation of n-ary predicates to binary relations are achieved by working with relations that stand for pairs of vectors of length *at least* n, rather than those of length exactly n, since this permits dynamic expansion of arities on demand. By this device we are able to compile programs independently of queries, which may introduce an unpredictable number of new variables. This dictates certain choices of notation and conventions, such as associating pairs to the right (like simple type expressions).

We define the set of [Σ-closed] *extended* terms over Σ by letting Σ^+ be the signature with constants given by all [closed] terms over Σ together with the binary function symbol

$$[\ ,\ ].$$

A Σ-open or Σ-closed extended term is a closed term over this meta-signature, built up from Σ-terms that are open or closed. In other words, we abuse language, and let closed or open refer to the bottom-tier terms that occur in our pairs. We will never need top-tier variables. Spelling it out via an inductive definition:

Definition 2.1 *Let Σ be a signature and $\mathcal{T}_\Sigma$ the set of terms over Σ. Then the set $\mathcal{T}_\Sigma^+$ of extended terms is defined as follows:*

1. *If u is a [closed] term over Σ it is a [Σ-closed] extended term.*
2. *If u_1, u_2 are [Σ-closed] extended terms then so is $[u_1, u_2]$.*

Henceforth open or closed extended term will mean Σ-open or Σ-closed.

Formal Vectors and Concatenation

The presence of a formal pairing operator $[\,,]$ in $\mathcal{T}^+_\Sigma$ allows us to define an internal product of terms within the collection of extended terms. This pairing operator associates to the right, so that we will write, for example

$$[t1, t2, t3, t4] \text{ for } [t_1, [t_2, [t_3, t_4]]]$$

which we can think of as a *formal vector* of terms. If u is the extended term $[u_1, u_2, \ldots, u_n]$, and x is an extended term, we write ux to denote the extended term

$$[u_1, u_2, \ldots, u_n, x].$$

Thus every extended term u can be assigned a *length* n and components $u_1, \ldots, u_n$ as follows:

Definition 2.2 *An extended term t has* **length** *1 if it is a Σ-term, and* **length** $n+1$ *if it is of the form ux, where x is not extended, and u is $[u_1, u_2, \ldots, u_n]$. We write $\| u \|$ for the length of u and call $[u_1, u_2, \ldots, u_n]$ the formal vector representation of u and u_i the ith component of u.*

Note that, in general, if u is an extended term of the form $[u_1, u_2, \ldots, u_n]$, it is a formal vector of length *at least* n (since u_n may be an extended term). If x is a Σ-term and u, v and extended terms $\| x \| = 1$ and $\| uv \| = \| u \| + \| v \|$.

We now associate to each signature Σ a relation calculus $\mathsf{Rel}\Sigma$ into which we will carry out a translation of both language and metalanguage. Among the atomic relation expressions in the calculus will be the n relation symbols f_i^n $(1 \leq i \leq n)$ for each function symbol of arity n in Σ, formalizing the relation of projection of each term of the form $f(u_1, \ldots, u_n)$ onto its i-th subterm, and the relation symbol (a, a) for each constant a of Σ.

The relation expression (a, a) is a formal counterpart to the singleton relation $\{(a, a)\}$. The *hd* and *tl* relation expressions are formal projections corresponding to the pairing operator in the meta-signature Σ^+.

2.2 Relational Syntax

Definition 2.3 *Let R_{var} be a countable set of relation variables. The relation calculus $\mathsf{Rel}\Sigma$ has the following syntax. The atomic relation expressions are defined by*

$$\langle R_{atom} \rangle \;\; ::= \;\; \langle R_{var} \rangle \mid id \mid di \mid \mathbf{1} \mid \mathbf{0} \mid hd \mid tl \mid \langle R_\Sigma \rangle$$

where R_Σ is the set of relation constants consisting of all expressions $\{f_i^n : f \in \mathcal{F}_\Sigma, \alpha(f) = n, 1 \leq i \leq n\}$ and $\{(a, a) : a \in \mathcal{C}_\Sigma\}$. Compound expressions are given by the following BNF grammar:

$$R \;\; ::= \;\; \langle R_{atom} \rangle \mid R^o \mid R \cup R \mid R \cap R \mid RR \mid \mathbf{fp}\langle R_{var} \rangle.(R)$$

Juxtaposition RR means composition, which we will sometimes write $R;R$ using the infix operator ";" to enhance legibility. Powers $R^{(n)}$ denote iterations of composition. We denote by $\Re_\Sigma$ the open terms (relation expressions) over the variables R_{var}. When the underlying signature is clear from context we will write $\Re$ for this set of terms. We will say a term is recursion-free *if it contains no occurrences of* **fp**.

Certain compound relation expressions will play an especially important role in the calculus, so we introduce them here.

Definition 2.4 *Define the countable sequence $\{P_i : 1 \leq i\}$ of projection relations as follows:*

$$P_1 = hd \qquad P_2 = tl; hd \qquad \cdots \qquad P_n = (tl)^{(n-1)}; hd \qquad \cdots \tag{1}$$

In the standard semantics to be discussed below, P_i is a relation between a formal vector with at least $i+1$ components and its i-th component. That is to say, P_i's set-theoretic interpretation consists of pairs of formal vectors (u, u_i).

Definition 2.5 *The relation of partial identity up to the first n components on vectors of length at least $n+1$ is defined as follows:*

$$I_n := \bigcap_{1 \leq i \leq n} P_i (P_i)^o \tag{2}$$

We also define the relation of true identity on such vectors by

$$id_n := \left(\bigcap_{1 \leq i \leq n} P_i (P_i)^o \right) \cap tl^n (tl^n)^o. \tag{3}$$

id_n is a subrelation of the identity relation id as a consequence of the equational theory we will define below and, of course, also in the standard semantics.

Definition 2.6 *We also define the relation of* identity on f-terms *for f a function symbol of arity n in the signature Σ by*

$$id_f := \bigcap_{1 \leq i \leq n} f_i^n (f_i^n)^o$$

In the standard interpretation, id_f will denote the set of identical pairs of terms beginning with the function symbol f.

Finally, for each pair of ground terms t_1, t_2 over Σ, define the formal relation expression (t_1, t_2), (whose semantics will be the singleton $\{(t_1, t_2)\}$), as follows:

$$(a, a) \quad := \quad \textit{already defined} \tag{4}$$

$$(f(u_1, \ldots, u_n), f(u_1, \ldots, u_n)) \quad := \quad \bigcap_{1 \leq i \leq n} f_i^n (u_i, u_i)(f_i^n)^o \tag{5}$$

$$(t_1, t_2) := (t_1, t_1)\mathbf{1}(t_2, t_2) \tag{6}$$

It is often convenient to work in a *derived* relational structure where the formal relations (t_1, t_2) are new primitives, satisfying the preceding equations. We will not make use of this variant here.

The reader should also note that unlike id_n, I_n is not a subrelation of the identity relation. In the standard set-theoretic interpretation to be defined below, it will denote all pairs of vectors (ux, uy), with u of length n.

2.3 The Equational Theory $\mathsf{Rel}\Sigma$

We will be using an equational theory which captures the general properties of relations we need as well as the behavior of the encoding of the structure of a Σ-term algebra. We therefore break it down into two components, one we will call DRA, the theory of distributive relation algebras, and the other $R\Sigma$. The latter theory is quite domain specific. It is the part we would expect to change to capture constraint logic programming over other domains, although this matter will not be taken up here.

Note that the containment $A \subseteq B$ abbreviates the equation $A \cap B = B$.

DRA

$$R \cap R = R \qquad R \cap S = S \cap R \qquad R \cap (S \cap T) = (R \cap S) \cap T$$
$$R\,id = R \qquad R\mathbf{0} = \mathbf{0} \qquad \mathbf{0} \subseteq R \subseteq \mathbf{1}$$
$$R \cup R = R \qquad R \cup S = S \cup R \qquad R \cup (S \cup T) = (R \cup S) \cup T$$
$$R \cup (S \cap R) = R = (R \cup S) \cap R$$
$$R(S \cup T) = RS \cup RT \qquad (S \cup T)R = SR \cup TR$$
$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$
$$(R \cup S)^o = R^o \cup S^o \qquad (R \cap S)^o = S^o \cap R^o$$
$$R^{oo} = R \qquad (RS)^o = S^o R^o$$
$$R(S \cap T) \subseteq RS \cap RT \qquad RS \cap T \subseteq (R \cap TS^o)S$$
$$id \cup di = \mathbf{1} \qquad id \cap di = \mathbf{0}$$

RΣ

$$\mathbf{1}(a,a)\mathbf{1} = \mathbf{1} \qquad (a,a)R(a,a) = (a,a) \cap R \qquad (a,a) \subseteq id$$
$$hd(hd)^o \cap tl(tl)^o \subseteq id \qquad (hd)^o hd = (tl)^o tl = id \qquad (hd)^o tl = \mathbf{1}$$
$$id_f \equiv \bigcap_{1 \le i \le n} f_i^n (f_i^n)^o \subseteq id \qquad (f_j^n)^o f_i^n = \mathbf{1} \quad (i \neq j)$$
$$(f_i^n)^o f_i^n = id \qquad (f_i^n)^o g_j^m = \mathbf{0}$$
$$(f_1)_{i_1}^{n_1} (f_2)_{i_2}^{n_2} \cdots (f_k)_{i_k}^{n_k} \ \cap \ id = \mathbf{0}$$
$$hd^o f_i^n = 0 = tl^o f_i^n \qquad f_i^n hd = 0 = f_i^n tl \qquad hd \cap id = 0 = tl \cap id$$
$$id = \bigcup\{(a,a) : a \in \mathcal{C}_\Sigma\} \quad \cup \quad \bigcup\{id_f : f \in \mathcal{F}_\Sigma\}$$

We will also make occasional use of a fixpoint-operator, which will be used in relational structures satisfying axiom **fp**

$$\mathbf{fp}x.\mathcal{E}(x) = \mathcal{E}(\mathbf{fp}x.\mathcal{E}(x)). \tag{7}$$

The so-called *modular law* included in **DRA**

$$RS \cap T \subseteq (R \cap TS^o)S$$

plays an important role in the work discussed here. In the presence of the other **DRA** axioms it is equivalent to its left-modular formulation

$$T \cap RS \subseteq R(R^o T \cap S)$$

and the equational formulation

$$T \cap RS = R(R^o T \cap S) \cap T.$$

The reader should consult [BroLip] for an example of how the modular law is used to improve termination behavior of certain relational rewrite systems, in particular in the computation of transitive closure.

We will have need for the following simplification of the modular law in special contexts.

Lemma 2.7 *In the equational theory DRA, from $SS^o \subset id$ we can infer $A \cap SR = S(S^oA \cap R)$. ¿From $S^oS \subset id$ we can infer $A \cap RS = (AS^o \cap R)S$.*

Proof: By the modular law we have, in the first case, $A \cap SR = S(S^oA \cap R) \cap A$. But $S(S^oA \cap R) \subseteq SS^oA \cap SR \subseteq idA \cap SR = A \cap SR$. Thus $S(S^oA \cap R) \cap A$ reduces to $S(S^oA \cap R)$. The argument for the second claim is symmetric. ▮

In the standard interpretation defined below, the axioms

$$f_i^n hd = 0 = f_i^n tl$$

rule out the occurrence of extended terms as arguments to function symbols from Σ.

Some of the axioms of RΣ are a relational translation of the first-order theory CET: Clark's equality theory [Lloyd]. The last axiom is a relational counterpart to the so-called *domain closure* axiom DC_Σ, satisfied in the Herbrand Universe over a finite signature, stating that every individual is a constant or a term beginning with one of the function symbols in the signature. The axiom scheme

$$(f_1)_{i_1}^{n_1}(f_2)_{i_2}^{n_2}\cdots(f_k)_{i_k}^{n_k} \;\cap\; id = \mathbf{0}$$

for $f_1, \ldots, f_k$ function symbols of Σ of arities $n_1, \ldots, n_k$ respectively, enforces OC, the *occurs-check* axiom scheme $\neg(x = t[x])$ for every term t not identical to x in which x explicitly occurs.

In [Maher], building on earlier results of Mal'cev, Maher shows that CET + OC + DC_Σ is complete, decidable, and admits a partial elimination of quantifiers. This can be exploited to give a compact representation of the set-theoretic content of relations over this equational theory. See [BroLip] for details.

A number of useful identities follow from the equations above, and the definitions (6), (4) and (5).

$$\mathbf{1}^o = \mathbf{1} \qquad \mathbf{0}^o = \mathbf{0} \qquad \mathbf{11} = \mathbf{1} \qquad id^o = id \qquad idR = R$$
$$A \cap B = B \Rightarrow A \cup B = A \qquad A \subseteq B \Rightarrow A^o \subseteq B^o$$
$$f_i^n \cap g_j^m = \mathbf{0} \qquad f_i^n \subseteq di$$
$$(u, v) \cap R = (u, u)R(v, v) \qquad A \cap RS \subseteq R(R^oA \cap R)$$
$$A \cap RS = R(R^oA \cap S) \cap A = (AS^o \cap R)S \cap A$$
$$id_n \subseteq id$$

as well as the facts that $\cap$ and $\cup$ are the lattice-theoretic infimum and supremum with respect to containment.

2.4 Semantics

2.4.1 Two-tiered Structures

Our semantics must interpret formal vectors of terms. It is convenient to do this via a two-tiered notion of model, the lower tier of which is a conventional Σ-algebra, and the upper tier of which interprets the product structure.

Definition 2.8 *A* Σ^+-**algebra** $\mathcal{U}$ *is a pair of structures* $(\mathcal{U}_0, \mathcal{U}_1)$ *with the carrier of the first contained in the second* ($U_0 \subseteq U_1$)*, such that* $\mathcal{U}_0$ *is a* Σ*-algebra and* $\mathcal{U}_1$ *is a set equipped with a binary injective function* $\langle _, _ \rangle_{\mathcal{U}} : U_1 \times U_1 \to U_1$*. We will refer to the set* U_1 *as the carrier of the* Σ^+*-algebra* $\mathcal{U}$*, and denote it by* U*, and the function* $\langle _, _ \rangle_{\mathcal{U}}$ *as the pairing function supplied by* $\mathcal{U}$*.*

A Σ^+-**morphism** $\psi : (\mathcal{U}_0, \mathcal{U}_1) \longrightarrow (\mathcal{V}_0, \mathcal{V}_1)$ *is a function from* U_1 *to* V_1 *whose restriction to* U_0 *is a* Σ*-algebra homomorphism from* $\mathcal{U}_0$ *to* $\mathcal{V}_0$ *and which satisfies* $\psi(\langle x, y \rangle_{\mathcal{U}}) = \langle \psi x, \psi y \rangle_{\mathcal{V}}$*.*

The reader can easily check that $(\mathcal{T}_\Sigma, \mathcal{T}_\Sigma^+)$ is an initial Σ^+-algebra, with pairing function $\langle s, t \rangle = [s, t]$.

2.4.2 Relational Structures

We will use the expression *Rel*($\mathcal{U}$) to denote the full relation algebra $\mathcal{P}(U \times U)$.

We will call a structure $\mathfrak{A} = \langle A, \mathbf{0}^{\mathfrak{A}}, \mathbf{1}^{\mathfrak{A}}, id^{\mathfrak{A}}, di^{\mathfrak{A}}, \cap^{\mathfrak{A}}, \cup^{\mathfrak{A}}, (\ )^{o,\mathfrak{A}}, \circ^{\mathfrak{A}} \rangle$ interpreting the relational syntax of the the theory DRA a *distributive relation algebra* or DRA if it satisfies the axioms DRA. It is called a set-theoretic or *standard* DRA if, in addition, A is a subset of the set of binary relations *Rel*(U) $= \mathcal{P}(U \times U)$ on some domain U, with $\mathbf{0}^{\mathfrak{A}}, \mathbf{1}^{\mathfrak{A}}$ the empty and maximal set, and with union, intersection, converse and composition standardly interpreted.

A structure $\mathfrak{A} = \langle A, \mathbf{0}^{\mathfrak{A}}, \mathbf{1}^{\mathfrak{A}}, id^{\mathfrak{A}}, di^{\mathfrak{A}}, (a,a)^{\mathfrak{A}} hd^{\mathfrak{A}}, tl^{\mathfrak{A}}, (f_i^n)^{\mathfrak{A}}, \cap^{\mathfrak{A}}, \cup^{\mathfrak{A}}, (\ )^{o,\mathfrak{A}}, \circ^{\mathfrak{A}} \rangle_{f \in \mathcal{F}_\Sigma; a \in \mathcal{C}_\Sigma}$ is

called a RelΣ-algebra if it interprets relational syntax in the obvious way, and satisfies the equational theory RelΣ. It is set-theoretic or *standard* if in addition A is a subset of *Rel*($\mathcal{U}$) for some Σ^+-algebra $\mathcal{U}$, and in addition to the conditions satisfied by standard DRA's, it interprets $R\Sigma$'s relational constant symbols (a, a) and f_i^n in a standard way:

$$(a,a)^{\mathfrak{A}} = \{(a^{\mathcal{U}}, a^{\mathcal{U}})\}$$
$$(f_i^n)^{\mathfrak{A}} = \{(\boldsymbol{x}, \boldsymbol{y}) : (\exists v_1 \cdots v_{n-1})(\boldsymbol{x} = f^{\mathcal{U}}(v_1, \ldots, v_{i-1}, \boldsymbol{y}, v_{i+1}, \ldots, v_{n-1}))\}.$$

Definition 2.9 *Given a* Σ^+*-algebra* $\mathcal{U}$*, a* $\mathcal{U}$-**interpretation** *is a mapping*

$$[\![\ ]\!]_{\mathcal{U}} : \Re \to Rel(\mathcal{U})$$

satisfying

$$\begin{array}{lcl} [\![(a,a)]\!]_{\mathcal{U}} & = & \{(a^{\mathcal{U}}, a^{\mathcal{U}})\} \\ [\![id]\!]_{\mathcal{U}} & = & \{(u,u) : u \in U\} \\ [\![hd]\!]_{\mathcal{U}} & = & \{(\langle t_1, t_2\rangle, t_1) : t_1, t_2 \in U\} \\ [\![\mathbf{1}]\!]_{\mathcal{U}} & = & U \times U \\ [\![\mathbf{0}]\!]_{\mathcal{U}} & = & \emptyset \end{array} \qquad \begin{array}{lcl} [\![tl]\!]_{\mathcal{U}} & = & \{(\langle t_1, t_2\rangle, t_1) : t_1, t_2 \in U\} \\ [\![di]\!]_{\mathcal{U}} & = & \{(u,v) : u \neq v \in U\} \\ [\![R \cup S]\!]_{\mathcal{U}} & = & [\![R]\!]_{\mathcal{U}} \cup [\![S]\!]_{\mathcal{U}} \\ [\![R \cap S]\!]_{\mathcal{U}} & = & [\![R]\!]_{\mathcal{U}} \cap [\![S]\!]_{\mathcal{U}} \end{array}$$
$$[\![RS]\!]_{\mathcal{U}} = \{(x,y) : \exists v((x,v) \in [\![R]\!]_{\mathcal{U}} \wedge (v,y) \in [\![S]\!]_{\mathcal{U}})\}$$
$$[\![f_i^n]\!]_{\mathcal{U}} = \{(\boldsymbol{x}, \boldsymbol{y}) : (\exists v_1 \cdots v_{n-1})(\boldsymbol{x} = f^{\mathcal{U}}(v_1, \ldots, v_{i-1}, \boldsymbol{y}, v_{i+1}, \ldots, v_{n-1}))\}$$
$$[\![\mathbf{fp}x.\mathcal{E}(x)]\!]_{\mathcal{U}} = \bigcup_{n \geq 0} E^{(n)}$$

where $E^{(0)} = \emptyset$ *and* $E^{(n+1)} = [\![\mathcal{E}(x)]\!]_{\mathcal{U}}[x \leftarrow E^{(n)}]$*.*

The notation $[\![_]\!]_{\mathcal{U}}[x \leftarrow R]$ means the function that returns the same value as $[\![_]\!]_{\mathcal{U}}$ on all inputs save x, for which the output is R.

It is easy to show that any *environment* (function η from relation variables to sets in *Rel*($\mathcal{U}$)) extends uniquely to a $\mathcal{U}$-**interpretation**. We also note that any interpretation

is sound for the axioms DRA. We are interested in interpretations which also satisfy the axioms RΣ. We call these $\mathsf{Rel}\Sigma$-*interpretations.*

In addition to the semantics just given, natural categorical models exist: tabular distributive allegories [FreySce] provide a semantics for a considerably more general notion of logic program, over a finite product category [FFL, NFDP, PowKin, CorMont]. We will not have need of this generality to describe relational complilation of conventional Horn Clause programs, although, in the presence of types and other programming features, there are interesting applications of such structures to logic programming, abstract interpretation and compilation.

Lemma 2.10 *If Σ is a signature with at least one function symbol and one constant symbol, and if U is the free Σ^+-algebra (consisting of all closed extended terms over Σ), then Rel($\mathcal{U}$) is a set-theoretic $\mathsf{Rel}\Sigma$-algebra in which axiom* **fp** *holds. Thus, the theory* $\mathsf{Rel}\Sigma$ + **fp** *is sound in any $\mathcal{U}$-interpretation.*

The proof is straightforward: every full power set is a complete (Boolean) relation algebra. The axioms RΣ, as mentioned above, translate into the first-order statement of CET +OC+ DC_Σ, which holds in the free extended term algebra $\mathcal{T}_\Sigma^+$.

Definition 2.11 *Let $\mathcal{I}_\mathcal{U}$ be the set of interpretations into Rel($\mathcal{U}$).*

We denote by $\sqsubseteq, \sqcap, \sqcup$ the pointwise order and operations on $\mathcal{I}_\mathcal{U}$ induced by the corresponding set-theoretic operations on Rel($\mathcal{U}$). $[\![\,]\!]_\mathcal{U}^\perp$ and $[\![\,]\!]_\mathcal{U}^\top$ denote the maximal and minimal interpretations obtained by setting all relational variables equal to the interpretations of **0** *and* **1** *respectively.*

We now state some useful properties of interpretations, which follow from well known facts about lattices.

Theorem 2.12 *The structure $\langle \mathcal{I}_\mathcal{U}, \sqsubseteq, \sqcap, \sqcup, [\![\,]\!]_\mathcal{U}^\perp, [\![\,]\!]_\mathcal{U}^\top \rangle$ is a complete lattice, with supremum and infimum defined pointwise. Furthermore, each $[\![\,]\!]_\mathcal{U}$ is completely determined by its values on relation variables. If $[\![\,]\!]_\mathcal{U}^1$ and $[\![\,]\!]_\mathcal{U}^2$ are two interpretations with $[\![X]\!]_\mathcal{U}^1 \subseteq [\![X]\!]_\mathcal{U}^2$ for each relation variable X then $[\![\,]\!]_\mathcal{U}^1 \sqsubseteq [\![\,]\!]_\mathcal{U}^2$.*

Theorem 2.13 *Let $X_1, \ldots, X_n$ be relation variables, and let $\mathcal{F}$ be a corresponding set*

$$\{F_i(X_1, \ldots, X_n) : 1 \leq i \leq n\}$$

of relation expressions with at most the X_i free. Define $\Phi_\mathcal{F} : \mathcal{I}_\mathcal{U} \to \mathcal{I}_\mathcal{U}$ by

$$\Phi_\mathcal{F}([\![\,]\!]_\mathcal{U})(X_i) = [\![F_i(X_1, \ldots, X_n)]\!]_\mathcal{U}.$$

Then

1. *$\Phi_\mathcal{F}$ is continuous (hence monotone).*

2. *$\Phi_\mathcal{F}$ has a least fixed point, $[\![\,]\!]_\mathcal{U}^*$, which is the least interpretation in $\mathcal{I}_\mathcal{U}$ satisfying the equations*
$$X_i = F_i(X_1, \ldots, X_n).$$
$[\![\,]\!]_\mathcal{U}^*$ is equal to the supremum of a countable chain $\bigsqcup [\![\,]\!]_\mathcal{U}^n$ of interpretations, where $[\![\,]\!]_\mathcal{U}^o = [\![\,]\!]_\mathcal{U}^\perp$ and $[\![\,]\!]_\mathcal{U}^{n+1} = \Phi_\mathcal{F}([\![\,]\!]_\mathcal{U}^n)$.

Proof: Straightforward. Proofs of most of these facts occur in one form or another in the literature. The last assertion is the well-known Tarski-Knaster theorem. ▮

For the rest of this paper we fix a finite signature Σ (assumed to have at least one function symbol and one constant), and take the universe $\mathcal{U}$ above to be the set of extended ground terms $\mathcal{T}_\Sigma^+$, with pairing function given by

$$\langle\ \rangle(u, v) = [u, v].$$

When $\mathcal{U}$ is so chosen we call the resulting semantics a *standard interpretation* of $\Re$ into $Rel(T_\Sigma^+)$, and denote it by the unadorned bracket $[\![\ ]\!]$. Since interpretations into the same range can only differ on relation variables and open relation expressions, we sometimes refer to $[\![R]\!]$, for closed R, as "the" standard interpretation of R.

We close the section by recalling a few elementary definitions from the theory of relations.

Definition 2.14 *A binary relation R is* **reflexive** *if $id \subseteq R$,* **coreflexive** *if $R \subseteq id$,* **functional** *if $R^o R$ is coreflexive, and* **injective** *if RR^o is coreflexive.*

When stating that a certain relation expression is, e.g. injective or coreflexive, we mean that it is so when interpreted in the standard semantics. When we mean that it can be proved so in one of the equational theories defined in this paper, we will so indicate.

Coreflexive relations play a critical role here, since logic programs are translated into such relations, (although some components of logic programs are built up from more general relations). Coreflexive relations are often a good way to code *data* in the calculus of binary relations, that is to say, by embedding *sets* S into the calculus of relations as $S^c = \{(x, x) : x \in S\}$. Diverse approaches to the formalization of data types in the relation calculus has been studied extensively by the Eindhoven group [Backh] and by Bird and de Moor [Algebra].

Three particularly important coreflexive relations associated with any binary relation R are its formal diagonal, domain and range.

$$\begin{aligned}
\Delta(R) &= R \cap id \\
Dom(R) &= R\mathbf{1} \cap id = RR^o \cap id \\
Ran(R) &= R^o\mathbf{1} \cap id = R^o R \cap id
\end{aligned}$$

3 Logic without Variables

One of the main results in [TarGiv] due to Tarski, Maddux and Givant (the so-called equipollence theorem), is that every first-order sentence in a theory φ over a theory with a pairing operator has a semantically equivalent equational counterpart $X_\varphi = 1$ in the theory **QRA** of relation algebras with quasi-projections. Tarski and Givant also prove a stronger proof-theoretic version of this result, and exhibit a bijective recursive transformation of sentences φ to their associated relation expressions X_φ and of first-order proofs of the former to equational derivations of the latter. We will not make direct use of the results or proofs here, but the work undertaken in this paper was inspired by it. The main contribution of the paper is to apply this transformation to compilation

and evaluation of logic programming by extending it to map proof search in a fragment of first-order logic into rewriting in the appropriate relational theory.

We now sketch a simple proof of a semantic form of the equipollence theorem, for the special case of algebras of relations definable on a term algebra by first-order formulas over a given signature. Let Σ be a finite language. The atomic statements of $\mathcal{T}_\Sigma$ are of the form

$$t_1 = t_2$$

which can be rewritten (after introducing a new variable x) as a conjunction of two *basic* equations $x = t_1$ $x = t_2$. If we continue introducing variables, we can write this as a conjunction of *elementary* or *flat* equations of the form $x = a$ or $x = f(y_1, \ldots, y_n)$ where a is a constant in Σ and the x, y_i are variables. We assume all atomic formulas are of this form.

Now we define a translation $(\ )^r$ from formulas θ in the language of $\mathcal{T}_\Sigma$ to relation expressions as follows. Recall that $\Re_\Sigma$always contains the reserved projection operations *hd* and *tl*[2] Let n be a natural number greater than the largest number of variables occurring in any sentence to be considered below.

Now define, for $(0 \leq i \leq n)$

$$S_1 = P_1 \qquad S_2 = P_2 \qquad \ldots \qquad S_{n-1} = P_{n-1} \qquad S_n = tl^n.$$

where the P_i are given in (1) Observe that, in the standard interpretation for $1 \leq i \leq n$, we have $(u, v) \in [\![S_k]\!]$ iff v is the kth component of u. Now define

$$Q_i^n = \bigcap_{j \neq i \leq n} S_j (S_j)^o \qquad id_n = \bigcap_{j \leq n} S_j (S_j)^o$$

Observe that $(u, v) \in [\![id_n]\!]$ means u and v are vectors of length n and have the same components $(u)_i$ for $(1 \leq i \leq n)$, and that $u [\![Q_i]\!] v$ means all but the i-th component of u and v agree. Let $x_1, \cdots, x_s$ be all the variables, free or bound, that may occur in θ. Recall all atomic formulas θ may be taken *elementary*: either $x_i = a, x_i = x_j$ or $x_{i_0} = f(x_{i_1}, \ldots, x_{i_n})$.

$$\begin{array}{lcllcl}
(x_i = a)^r & = & S_i(a,a)S_i^o \cap id_n & (\varphi \wedge \theta)^r & = & (\varphi)^r \cap (\theta)^r \cap id_n \\
(x_i = x_j)^r & = & S_i S_j^o \cap id_n & (\neg\varphi)^r & = & id_n - (\varphi)^r \\
(x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r & = & \bigcap_j S_{i_0}; f_j^n; S_{i_j}^o \cap id_n & (\exists x_i \varphi)^r & = & Q_i (\varphi)^r Q_i \cap id_n
\end{array}$$

Then we have the following result, where $\mathcal{H}_\Sigma$ denotes the free Σ-algebra, with carrier the set of closed Σ-terms.

Theorem 3.1 (Freyd, Maddux, Tarski) *Let θ be a sentence over the language of $\mathcal{T}_\Sigma$, and let $x_1, \cdots, x_n$ contain all the variables, free or bound, that may occur in θ. Then $\mathcal{H}_\Sigma \models \theta \iff [\![(\theta)^r]\!] = [\![id_n]\!]$. For open formulas φ with free variables among $x_1, \ldots, x_n$*

$$\{(a_1, \ldots, a_n) : ([a_1, \ldots, a_n], [a_1, \ldots, a_n]) \in [\![(\varphi)^r]\!]\} = \{(a_1, \ldots, a_n) : \mathcal{H}_\Sigma \models \varphi[a_1/x_1, \ldots, a_n/x_s]\}$$

The proof is a straightforward induction on the structure of formulas[3].

Proof:

[2]called quasiprojections by Tarski and, in an essentially equivalent form, tabulations of the maximal relation on the product by Freyd[FreySce].

[3]The authors are indebted to Peter Freyd [Freyd92] for (a variant of) this formulation and proof of the variable elimination result.

$\theta \equiv (x_i = a)$:
Let $\vec{u}$ be an n-tuple of terms in $\mathcal{H}_\Sigma$, and suppose

$$(\vec{u},\vec{u}) \in [\![(x_i = a)^r]\!] \;=\; [\![S_i(a,a)S_i^o \cap id_n]\!] \;.$$

This implies that $u_i = a$, hence $\mathcal{H}_\Sigma \models (x_i = a)[u_1/x_1,\ldots,u_n/x_n]$. Conversely $u_i = a$ forces $(\vec{u},\vec{u}) \in [\![(x_i = a)^r]\!]$.

$\theta \equiv (x_i = x_j)$:
Suppose $(\vec{u},\vec{u}) \in [\![(x_i = x_j)^r]\!] = [\![S_iS_j^o \cap id_n]\!]$. Then $u_i = u_j$ and $\mathcal{H}_\Sigma \models (x_i = x_j)[\vec{u}/\vec{x}]$. The converse is also immediate.

$\theta \equiv x_{i_0} = f(x_{i_1},\ldots,x_{i_n})$:
Suppose $(\vec{u},\vec{u}) \in [\![(f(x_{i_1},\ldots,x_{i_n}))^r]\!]$, that is to say, in

$$\bigcap_j [\![S_{i_0}; f_j^n; S_{i_j}^o]\!] \cap [\![id_n]\!].$$

Then $u_{i_0} = f(u_{i_1},\ldots,u_{i_n})$ and $\mathcal{H}_\Sigma \models \theta[\vec{u}/\vec{x}]$. The converse is immediate.

$\theta \equiv \varphi \wedge \theta$:
Suppose $(\vec{u},\vec{u}) \in [\![(\varphi \wedge \theta)^r]\!]$. Then $(\vec{u},\vec{u}) \in [\![(\varphi)^r]\!]$ and $(\vec{u},\vec{u}) \in [\![(\theta)^r]\!]$. By the induction hypothesis, $\mathcal{H}_\Sigma \models \varphi[\vec{u}/\vec{x}]$ and $\mathcal{H}_\Sigma \models \theta[\vec{u}/\vec{x}]$ and hence $\mathcal{H}_\Sigma \models (\varphi \wedge \theta)[\vec{u}/\vec{x}]$

$\theta \equiv \neg\varphi$:
Suppose $(\vec{u},\vec{u}) \in [\![(\neg\varphi)^r]\!]$. Then $(\vec{u},\vec{u}) \in id_n - [\![(\varphi)^r]\!]$ which is equivalent to saying that $(\vec{u},\vec{u}) \in [\![(\varphi)^r]\!]$ is not the case. Using the induction hypothesis, this is equivalent to $\mathcal{H}_\Sigma \not\models \varphi[\vec{u}/\vec{x}]$, and hence to $\mathcal{H}_\Sigma \models \neg\varphi[\vec{u}/\vec{x}]$.

$\theta \equiv \exists x_i\varphi$:
If $(\vec{u},\vec{u}) \in [\![(\exists x_i\varphi)^r]\!]$ we must have $(\vec{u},\vec{u}) \in [\![Q(\varphi)^rQ]\!]$. This means there is an n-tuple of terms $\vec{v}$ with $v_j = u_j$ for every j between 1 and n except i, and $(\vec{v},\vec{v}) \in [\![(\varphi)^r]\!]$. By the induction hypothesis, this means $\mathcal{H}_\Sigma \models \varphi[\vec{v}/\vec{x}]$. But this is equivalent to $\mathcal{H}_\Sigma \models \exists x_i\varphi[\vec{u}/\vec{x}]$. The converse is left to the reader. ▮

We now turn our attention to the translation of Horn Clause programs. We will want to carry this out in a way that not only preserves meaning, as in the preceding translation, but in a way that is faithful to the operational semantics of the program. We will thus make different, more economical choices, and carry out the translation into $\mathsf{Rel}\Sigma$, defined above, a relational calculus without negation, and with extra constants for capturing the structure of terms in a way that such variable dependent phenomena as unification are easily converted to a variable-free combinatory reduction.

It should be noted that the absence of negation in $\mathsf{Rel}\Sigma$is no handicap, vis-a-vis first-order formulas with negation over the Herbrand Universe, because of the well-known results of Mal'cev [Mal'cev, Maher] that any such formula is equivalent to a two-quantifier formula in which negation occurs only immediately preceding equations between terms, which can be modelled in $\mathsf{Rel}\Sigma$ using di for disequality.

4 Embedded Prolog

4.1 Capturing Horn Clause Reasoning over Term Models

We introduce a family of relational terms that will prove useful in translating logic programs over the Herbrand Universe into relational equations. One component of this translation involves a sequence of intermediate transformations of the constituent predicates $p(t_1, \ldots, t_n)$ of the program into relation expressions. We first introduce new variables $\vec{x} = x_1, \ldots, x_n$ and rewrite $p(t_1, \ldots, t_n)$ as

$$p(x_1, \ldots, x_n) \wedge x_1 = t_1 \wedge \ldots \wedge x_n = t_n.$$

Conjunctions will be replaced by commas below.

In the relational translation of programs, conjunctions of equations of the form $x_i = t_i$ are translated to relation expressions $\dot{K}(t_1, \ldots, t_n)$ defined by induction on the structure of the terms t_i, as discussed below. We show that the relevant relational translations preserve unification identities derived from Clark's equality theory. This will require a number of ancillary lemmas and definitions, to which the rest of the section is devoted.

Recall that a relation A is *simple* or *functional* if $A^o A \subset id$ and *injective* if $AA^o \subset id$, i.e. if A^o is functional. Injectivity and functionality are preserved under composition and intersection. Thus, in particular, P_j is functional (and hence $(P_j)^o$ injective) for every j, since it is built up from hd and tl by composition.

Definition 4.1 *Let D be the disjoint union of the set open terms over Σ and the set of sequences $\langle t_1, \cdots, t_n\rangle$ of open terms ($m \geq 1$). Define*

$$K : D \to Rel(\mathcal{U})$$

as follows. Let m, n, r be natural numbers greater than 0, and u be a term, or a sequence of n terms over some set $x_1, \cdots, x_m$ of variables. $K_m(u)$ is defined by induction on the structure of u.

$$\begin{aligned} K(a) &= (a, a)\mathbf{1} \\ K(x_i) &= (P_i)^o \\ K(f(u_1, \ldots, u_r)) &= \bigcap_{i \leq r} f_i^r K(u_i) \\ K(\langle t_1, \cdots, t_n\rangle) &= \bigcap_{i \leq n} P_i K(t_i) \end{aligned}$$

We also define, for u in D

$$\dot{K}(u) = id \cap K(u)\mathbf{1} \qquad \textit{and} \qquad \Delta(K)(u) = id \cap K(u)$$

Terms t and sequences $\langle t\rangle$ of terms of length 1 are treated differently by K in the preceding definition. Note that the sequences $\langle t_1, \ldots, t_n\rangle$ in D are true members of the cartesian product of copies of the set of terms, and not the formal vectors $[t_1, \ldots, t_n]$ in $\mathcal{T}_\Sigma^+$. Observe also that $K(u)$ is always *injective*, and that for any substitution θ (whose range is among the set of terms over the variables $x_1, \cdots, x_m$ in the universe of discourse), since

$$\theta f(u_1, \ldots, u_n) = f(\theta u_1, \ldots, \theta u_n),$$

we have $K(\theta f(u_1, \ldots, u_n)) = \bigcap_i f_i K(\theta u_i)$

In the standard set-theoretic interpretation, the K terms capture the structure of term equations in the following sense.

Lemma 4.2 *Let $\langle t_1, \cdots, t_n\rangle$ be a sequence of terms with free variables among $x_1, \cdots, x_m$. Let u and y be formal vectors of ground terms of lengths n and m respectively, and u', y' arbitrary extended terms. Let $\vec{u}$ be the sequence $\langle u_1, \ldots, u_n\rangle$. Then*

$$(uu', yy') \in [\![K(\langle t_1, \cdots, t_n\rangle)]\!] \iff \mathcal{H}_\Sigma \models \vec{u} = [t_1, \ldots, t_n][y_1/x_1, \cdots, y_m/x_m],$$

where $\mathcal{H}_\Sigma$ is the Herbrand Universe over Σ.

If we think of an m-tuple y of terms as denoting a substitution Θ_y for the variables $x_1, \cdots, x_m$, then $[\![K(\langle t_1, \cdots, t_n\rangle)]\!]$ is the set of instance-substitution pairs

$$\{(\Theta_y[t_1, \ldots, t_n, _], \Theta_{y_}) : y \in \mathcal{T}_\Sigma^m\}$$

where the underscores denote arbitrary additional components.

Proof:[of lemma 4.2] We first show by induction on the structure of terms t with free variables among $x_1, \cdots, x_m$ that for a ground term u and and an m-tuple of ground terms $y = \langle y_1, \cdots, y_m\rangle$ we have

$$(u, y) \in [\![K(t)]\!] \quad \text{if and only if} \quad \mathcal{H}_\Sigma \models \vec{u} = t[y_1/x_1, \cdots, y_m/x_m].$$

Suppose t is the constant a. Then $(u, y) \in [\![K(a)]\!] = [\![(a, a)\mathbf{1}]\!] = \{(a, w) : w \in \mathcal{T}_\Sigma^+\}$ if and only if u is a. But $a = a[y_1/x_1, \cdots, y_m/x_m]$.

If t is the variable x_i then $(u, y) \in [\![K(x_i)]\!]$ if and only if $(y, u) \in [\![P_i]\!]$ which means u is y_i. But this is equivalent to $u = x_i[y_1/x_1, \cdots, y_m/x_m]$.

If t is a term of the form $f(v_1, \ldots, v_n)$ then $(u, y) \in [\![K(t)]\!]$ if and only if

$$(u, y) \in [\![\bigcap_{i \le n} f_i^n K(v_i)]\!] \ .$$

But this means that for each i between 1 and n $(u, y) \in [\![f_i^n K(v_i)]\!]$, or, equivalently, for some ground terms $w_1, \ldots, w_{n-1}$ and z_i

$$u = f(w_1, \ldots, w_{i-1}, z_i, w_{i+1}, \ldots, w_{n-1})$$

in the Herbrand universe, and $(z_i, y) \in [\![K(v_i)]\!]$.

By the induction hypothesis z_i is $v_i[y_1/x_1, \cdots, y_m/x_m]$. Since this is true for every i we obtain

$$u = f(v_1, \ldots, v_n)[y_1/x_1, \cdots, y_m/x_m]$$

which is what we wanted to prove.

If $t = \langle t_1, \cdots, t_n\rangle$ then $(uu', yy') \in [\![K(t)]\!]$ if and only if $(uu', yy') \in [\![\bigcap_{i \le n} P_i K(t_i)]\!]$. Thus, for each i between 1 and n $(uu', yy') \in [\![P_i K(t_i)]\!]$, hence for each i $(u_i, y) \in K(t_i)$. But then, by the preceding case u_i is $t_i[y_1/x_1, \cdots, y_m/x_m]$, so u is $t[y_1/x_1, \cdots, y_m/x_m]$. ∎

Corollary 4.3 *If $m \ge 0$, u is a formal vector of length m of ground terms in $\mathcal{T}_\Sigma$, $\vec{u}$ the corresponding sequence $\langle u_1, \ldots, u_m\rangle$, x is any extended term, and $\langle t_1, \cdots, t_m\rangle$ is an m-tuple of open terms (for the same m) then*

$$(ux, ux) \in [\![id \cap K(\langle t_1, \cdots, t_m\rangle)\mathbf{1}]\!] \iff (\exists\theta)\mathcal{H}_\Sigma \models \vec{u} = \langle t_1, \cdots, t_m\rangle\theta$$

where θ is understood to range over substitutions for the variables free in the t_1.

We say two sequences of terms $\langle t_1, \ldots, t_m \rangle$ and $\langle s_1, \ldots, s_m \rangle$ are jointly unifiable by a single substitution θ if for every i, $t_i\theta = s_i\theta$.

Lemma 4.4 *Let $\langle t_1, \ldots, t_n \rangle$ and $\langle s_1, \ldots, s_m \rangle$ be sequences of open terms, with $n \geq m$. Then, if $\langle t_1, \ldots, t_m \rangle$ and $\langle s_1, \ldots, s_m \rangle$ are jointly unifiable, and θ is a most general unifier,*

$$[\![K(\langle t_1, \ldots, t_n \rangle) \cap K(\langle s_1, \ldots, s_m \rangle)]\!]_{\mathcal{U}} = [\![K(\langle t_1\theta, \ldots, t_n\theta \rangle)]\!]_{\mathcal{U}}.$$

If they are not unifiable, the interpretation of the intersection is the empty set.

In addition, if the two sequences are standardized apart

$$[\![\dot{K}(\langle t_1, \ldots, t_n \rangle) \cap \dot{K}(\langle s_1, \ldots, s_m \rangle)]\!]_{\mathcal{U}} = [\![\dot{K}(\langle t_1\theta, \ldots, t_n\theta \rangle)]\!]_{\mathcal{U}} \tag{8}$$

if they are unifiable. The intersection is empty otherwise.

The proof is an immediate consequence of lemma (4.2) and its corollary.

Proof: By lemma (4.2), if u and y are formal vectors of ground terms of lengths n and m respectively, $\vec{u} = \langle u_1, \ldots, u_n \rangle$ and u', y' are arbitrary extended terms, then

$$(uu', yy') \in [\![K(\langle t_1, \ldots, t_n \rangle) \cap K(\langle s_1, \ldots, s_m \rangle)]\!]_{\mathcal{U}}$$
$$\Longleftrightarrow$$
$$\mathcal{H}_\Sigma \models \vec{u} = \langle t_1, \ldots, t_n \rangle [y_1/x_1, \cdots, y_m/x_m] \wedge \vec{u} = \langle s_1, \ldots, s_n \rangle [y_1/x_1, \cdots, y_m/x_m].$$

Letting θ be the substitution represented by $[y_1/x_1, \cdots, y_m/x_m]$, this implies

$$\langle t_1, \ldots, t_n \rangle \theta = \langle s_1, \ldots, s_n \rangle \theta$$

in the Herbrand universe. Thus $[\![K(\langle t_1, \ldots, t_n \rangle) \cap K(\langle s_1, \ldots, s_m \rangle)]\!]_{\mathcal{U}}$ is precisely the set of instances of $[t_1, \ldots, t_n]\psi$ where ψ is any mgu of $\langle t_1, \ldots, t_n \rangle$ and $\langle s_1, \ldots, s_m \rangle$ whose range is in the set of terms over the variables $x_1, \cdots, x_m$. ▮

This lemma underscores a fundamental feature of variable-elimination in logic programming: *unification is reduced to intersection of combinators*. But one issue remains to be resolved before these results can be fully exploited. The use of $\dot{K}$ expressions in logic program reduction would appear to require standardizing their arguments apart, limiting their usefulness. Fortunately the term sequences that actually arise in our compilation are more robust, as we show in the next, somewhat technical lemma which has surprising consequences for the translation.

Lemma 4.5 (Diagonal Lemma) *Suppose $\langle t_1, \ldots, t_n \rangle$ is a sequence of terms in $\mathcal{T}_\Sigma$ all of whose free variables are among $x_1 \ldots, x_n$ (for the same n). Call a component t_i of such a sequence* **nontrivial** *if $t_i \neq x_i$. Further suppose that for each j between 1 and n if x_j occurs freely in a nontrivial term t_i then $i \neq j$ and t_j is x_j.*

Then for any formal vectors of ground terms u, d in $\mathcal{T}_\Sigma$ of length n and any formal vectors of ground $\mathcal{T}_\Sigma$-terms y, z, if $(uy, dz) \in K(\langle t_1, \ldots, t_n \rangle)$ then $(uy, uy) \in K(\langle t_1, \ldots, t_n \rangle)$ as well. Thus

$$\dot{K}(\langle t_1, \ldots, t_n \rangle) = id \cap K(\langle t_1, \ldots, t_n \rangle). \tag{9}$$

Thus, under these hypotheses, $\dot{K}$ expressions behave like K-terms and (8) holds without the restriction that term arguments be standardized apart. It turns out that the hypotheses of this lemma are always met by the term sequences that will be constructed in the

translation of Prolog programs described below. The reader should note, however, that identity (9) does not hold in general.

Proof: Suppose $(uy, dz) \in K(\langle t_1, \dots, t_n \rangle) = \bigcap_i P_i K(t_i)$. Then, by lemma (4.2), for each i

$$u_i = t_i[d_1/x_1, \dots, d_n/x_n].$$

Suppose $t_{i_1}, \dots, t_{i_n}$ are all the nontrivial terms among the t_i, and that $x_{j_1}, \dots, x_{j_s}$ are variables occurring in these terms. Then, by hypothesis, for all k between 1 and s, $t_{j_k} = x_{j_k}$. Thus

$$\begin{aligned} K(\langle t_1, \dots, t_n \rangle) &= \bigcap_k P_{j_k} K(x_{j_k}) \cap \bigcap_{i \notin \{j_1, \dots, j_s\}} P_i K(t_i) \\ &= \bigcap_k P_{j_k} P^o_{j_k} \cap \bigcap_{i \notin \{j_1, \dots, j_s\}} P_i K(t_i). \end{aligned}$$

Thus for each k between 1 and s, $u_{j_k} = d_{j_k}$. Since the x_{j_k} are only variables free in nontrivial t_i, for each such t_i, $u_i = t_i[d_1/x_1, \dots, d_n/x_n] = t_i[u_1/x_1, \dots, u_n/x_n]$. For every other t_i, $t_i = x_i$, so we immediately have $u_i = t_i[u_1/x_1, \dots, u_n/x_n]$. Therefore

$$\langle u_1, \dots, u_n \rangle = \langle t_1, \dots, t_n \rangle [u_1/x_1, \dots, u_n/x_n],$$

from which $(uy, uy) \in K(\langle t_1, \dots, t_n \rangle)$ by lemma (4.2). But then every uy in the domain of $K(\langle t_1, \dots, t_n \rangle)$ is in the domain of $K(\langle t_1, \dots, t_n \rangle) \cap id$, whence

$$\dot{K}(\langle t_1, \dots, t_n \rangle) = id \cap K(\langle t_1, \dots, t_n \rangle).$$

▮

Definition 4.6 *A sequence $\langle t_1, \dots, t_n \rangle$ of terms in $\mathcal{T}^+_\Sigma$ is called* **clean** *if it satisfies the hypothesis of the preceding lemma.*

Observe that $\langle t_1, \dots, t_n \rangle$ is clean if and only if the substitution $\varphi_{\vec{t}} = \{t_i/x_i : t_i \text{ non-trivial}\}$ induced by the equations $\{x_i = t_i : 1 \leq i \leq n\}$ is idempotent.

Lemma 4.7 *Suppose $\langle t_1, \dots, t_n \rangle$ is clean, and $(uy, uy) \in K(\langle t_1, \dots, t_n \rangle)$. Then the substitution $\theta_u = \{u_1/x_1, \dots, u_n/x_n\}$ satisfies the equations*

$$x_i = t_i \qquad (1 \leq i \leq n)$$

in the sense that $x_i \theta_u = t_i \theta_u$ for $(1 \leq i \leq n)$.

The proof is just a restatement of the fact, shown in the proof of lemma (4.5) that, for each i,

$$u_i = t_i[u_1/x_1, \dots, u_n/x_n].$$

4.1.1 Switching Relations

Definition 4.8 *Fix the natural number $n \geq 1$ and let $\vec{\imath} = \langle i_1, \dots, i_n \rangle$ be a sequence of distinct members of the set $\{1, \dots, n\}$ natural numbers. Then define the relations $W(\vec{\imath})$ as follows:*

$$W(i_1, \dots, i_n) = \bigcap_{j=1}^{n} P_{i_j} (P_j)^o.$$

$W(i_1, \dots, i_n)$ is called a **switching** *relation. We also use the notation $W(\sigma)$ where σ is the permutation of the first n natural numbers into $\vec{\imath}$.*

We now state three useful properties of switching relations, whose simple proofs are left to the reader.

Lemma 4.9 *$W(i_1,\ldots,i_n)$ is functional and injective, provably in the equational theory* $\mathsf{Rel}\Sigma$

Lemma 4.10 *Let $\vec{x}$, and $\vec{y}$ be formal vectors of terms of length at least $n+1$ over the Herbrand Universe. Then*

$$(\vec{x},\vec{y}) \in [\![W(i_1,\ldots,i_n)]\!] \iff \mathcal{H}_\Sigma \models \bigwedge_{j=1}^{n} x_{i_j} = y_j.$$

Furthermore, if $t_1,\ldots,t_n$ is a tuple of n terms, σ a permutation on $\{1,\ldots,n\}$, and $W = W(\sigma)$

$$[\![W^o \dot{K}(\langle t_1,\ldots,t_n\rangle)\, W]\!] = [\![\dot{K}(\langle t_{\sigma^{-1}(1)},\ldots,t_{\sigma^{-1}(n)}\rangle)]\!].$$

Lemma 4.11 *If σ is a permutation of the first n nonzero natural numbers*

$$W(\sigma^{-1}) = W(\sigma)^o.$$

provably in the theory $\mathsf{Rel}\Sigma$.

Another useful property of switching relations follows immediately from 2.7.

Lemma 4.12 *One can prove in the equational theory DRA that for any relation expressions R, Q and switching relations W*

$$R \cap WQW^o = W(W^oRW \cap Q)W^o$$

5 Relational Translation of Programs: A Sketch

An atomic formula is said to be *linear* or *pure* if its arguments constitute a linear sequence of variables, that is to say, one in which all variables are distinct.

The first step in the Clark completion of a prolog program is the linearization of head predicates. A clause

$$p(t_1,\ldots,t_r) : -Tl$$

is replaced by

$$p(x_1,\ldots,x_r): -x_1 = t_1,\ldots,x_n = t_n, Tl.$$

where the x_i are fresh. The variables free in the tail but *not* in the head are then existentially quantified:

$$p(x_1,\ldots,x_r) : -\exists\vec{y}[x_1 = t_1,\ldots,x_n = t_n, Tl].$$

Now all clauses with head predicate letter p how have precisely the same head, and are thus equivalent to a single clause with this head, but with tail replaced by a disjunction of the newly formed tails:

$$p(x_1,\ldots,x_r) : -\bigvee B_i$$

where each B_i is of the form

$$\exists\vec{y_i}[x_1 = t_{i1},\ldots,x_n = t_{in}, Tl_i] \tag{10}$$

We are now going to describe a series of additional transformations that have the effect of putting the logic program into a form that maintains its set of ground atomic consequences over the Herbrand Universe, while bringing it closer to the eventual shape of the desired relational equation. The first transformation involves the linearization of the predicates in the tails B_i, with the subsequent generation of equations, some of which then get modified. This is then followed by the introduction of selection operators. We describe it in two steps: linearization and selection.

1. **Linearization:** First we linearize the tails. Proceeding left-to-right, every atomic formula $q(s_1 \ldots, s_m)$ in Tl_i (the non-equational part of B_i) is rewritten as

$$z_1 = s_1, \ldots, z_m = s_m, q(z_1, \ldots, z_m),$$

where

Case 1: Either s_i is a variable that has not occurred earlier in the same atomic predicate, and z_i is identical to s_i (so the equation reads $s_i = s_i$), or

Case 2: z_i is a fresh variable, which is then existentially quantified on the outside. That is to say the existential quantifier $\exists \vec{y_i}$ becomes $\exists \vec{y_i} \vec{z}$, where $\vec{z}$ is the sequence of all distinct fresh variables in the sequence $\vec{z_1} \cdots \vec{z_\ell}$ of variables now occurring in the ℓ predicates $q_1(\vec{z_1}), \ldots, q_\ell(\vec{z_\ell})$ in the non-equational part of the tail.

Starting with the first index greater than r, the arity of the head predicate, which has already required the introduction of variables $x_1, \ldots, x_r$ that may now occur freely in the tail because Case 1, above, obtained, all new distinct existentially quantified variables are renamed in increasing order from left to right, so that *every variable occuring in the entire tail B_i* except for the original bound variables $\vec{y_i}$ is part of a master list $x_1, \ldots x_k$ where k is bounded by the sum of the arities of all the predicate occurrences in the head or tail of the clause.

Finally, we move all generated equations, now of the form $x_i = t_i$ (where the t_i are terms originally ocurring in the clause predicates) to the beginning of the quantifier-free part of the tail. Thus the tail is now of the form

$$\exists \vec{y_i} \exists x_{r+1} \cdots x_w [x_1 = t_1, \ldots, x_w = t_w, q_1(x_{r+1}, \ldots, x_{r+\alpha(1)}), \ldots, q_\ell(x_a, \ldots, x_{a+\alpha(\ell)})],$$

where r is the arity $\alpha(p)$ of the head predicate, $\alpha(i)$ the arity of the i-th tail predicate, and a is r + the sum of the arities of the preceding $\ell - 1$ tail predicates.

Now we rename the remaining existentially bound variables from $\vec{y_i}$ originally occurring in the tail, so they are added to the (end of the) master list of variables x_j.

Finally, for any variable x_j not occurring on the left hand side of one of the equations $x_i = t_i$ the equation $x_j = x_j$ is added to the equational part in its proper place. The equational part is now written in sequence form as $\vec{x} = \vec{t}$. It is a sequence of length bounded by n, the *weight* of the corresponding clause in the original prolog program, which is the sum of the number of variables k occurring in the tail and not in the head, and the arities of the predicate occurrences either in the head or tail of the original clause,

$$wt(C) = k + n + \Sigma_{i=1}^{\ell} r_i.$$

It may be less than the weight because of the possible occurrence of Case 1 above which ensured reuse of variables that would not obstruct linearity (distinctness of variables) of each predicate in the tail. For reference, each clause has the following appearance

$$p(x_1, \ldots, x_n) :- \bigvee B'_i \tag{11}$$

where

$$B'_i = \exists x_{r+1} \cdots x_n[\vec{x} = \vec{t}, q_{i1}(\vec{x_1}), \ldots, q_{i\ell_i}(\vec{x_\ell})] \tag{12}$$

and where each $\vec{x_j}$ is the j-th block of *distinct* variables of length the arity of q_{ij}, starting with the sum of the arities of the preceding predicate occurrences, including that of the head. Letting $\vec{x_0}$ be the sequence of variables now occurring in the head, the entire master sequence of variables $\vec{x} = \vec{x_0}\vec{x_1}, \ldots, \vec{x_\ell}$ may contain repetitions because of Case 1 above, but, we repeat, each block is linear.

2. **Selection:** Each sequence $\vec{x_i}$ of variables occurring in the i-th block in the tail can now be viewed as the result of applying a selection operator ρ_i to the master sequence $\vec{x}$:

$$\rho_i \vec{x} = \vec{x_i}.$$

This yields **normalized** disjuncts of the form

$$D_i = \exists x_{n+1} \cdots x_w[\vec{x} = \vec{t}, q_{i1}(\rho_1 \vec{x}), \ldots, q_{i\ell_i}(\rho_\ell \vec{x})]. \tag{13}$$

We now replace all implications in clauses by bi-implications, as in the Clark completion. We will call the resulting program $\mathcal{P}_n$, consisting of the clauses:

$$p_i(x_1, \ldots, x_n) :- \bigvee D_{ij} \tag{14}$$

the **completed** or **normal form** of $\mathcal{P}$.

Since we will not need this result in our adequacy theorem we do not take the trouble to establish that the preceding program transformations maintain ground atomic consequences in any term model. It is, however, straightforward to prove, using arguments similar to the proof of the fact that the Clark completion of a program has the same atomic (in fact positive) consequences as the program (see e.g. [Shep88, Lloyd]), and repeated applications of the logical equivalence, for fresh $y_1, \ldots, y_n$

$$\exists \vec{x}[A(t_1, \ldots, t_n)] \iff \exists \vec{x} y_1, \ldots, y_n[y_1 = t_1 \wedge \ldots \wedge y_n = t_n \wedge A(y_1, \ldots, y_n)]$$

In the case of Prolog programs with positive and negative equations, treated briefly in a later section, the completed (or *normalized* program) plays a central role, and should be seen as the proper logical formulation of the program captured by our relational translation.

We note following for future reference.

Lemma 5.1 *The sequences $\vec{t}$ that occur in the equational part of each disjunct in the normal form of a program are clean in the sense of definition (4.6).*

This is clear from the way fresh variables are introduced in the normalization process.

Our resulting set of equivalences are now almost ready to be cast in first-order-variable-free relational form. We need to define one more operation on sequences of variables (or on their indices): the permutation σ_i of the indices of $\vec{x}$ *induced* by the selection operator ρ_i.

Definition 5.2 *Let ρ_i be the i-th selection operator defined above. and w' the length of the i-th disjunct's master list of variables. Then the associated permutation of indices*

$$\sigma_i : \{1, 2, \ldots, w'\} \to \{1, 2, \ldots, w'\}$$

is given by letting $x_{\sigma_i(1)}, \ldots, x_{\sigma_i(w')}$ be the sequence $\vec{x_i}$ followed by all remaining variables in $\vec{x}$ in order. σ_i simply shifts the i-th block of variables to the front of the sequence.

5.1 The Relational Step

We now transform the normalized program defined above into a finite set of relation equations. For each predicate letter $q \in \Pi$ we introduce a relation *variable* $\overline{q}$. Now consider the normalized clause defining predicate p, of arity m. Each disjunct Θ in its tail of the form (13) is translated into the relation expression $(\Theta)^r$

$$id \cap I_m[\dot{K}(\vec{t}) \cap W(\sigma_1)\overline{p_1}W(\sigma_1)^o \cap \cdots \cap W(\sigma_\ell)\overline{p_\ell}W(\sigma_\ell)^o]I_m. \quad (15)$$

We then translate the whole clause (14) by

$$\overline{p} = (\Theta_1)^r \cup \cdots \cup (\Theta_m)^r.$$

As an immediate consequence of the preceding lemma and lemma (4.5), we have

Lemma 5.3 *If $\dot{K}(\vec{t})$ occurs in a disjunct of a normalized Prolog program, then*

$$\dot{K}(\vec{t}) = K(\vec{t}) \cap id.$$

The result of applying this procedure to all predicate definitions in the canonical completion of a Horn Clause program $\mathcal{P}$ is a set $\mathsf{E}_{\mathcal{P}}$ of n equations $e_1, \ldots, e_n$ in the n predicate letters of the program (now viewed as relation variables). The typical equation e_j being of the form

$$\overline{p}_j = \bigcup_i \Theta_i^j$$

as described above. The Θ_i^j contain the symbols $\overline{p_k}$ as relation variables corresponding to the original predicate letters in the program.

We can write this dependency of relation variables as

$$\overline{p}_i = F_j(\overline{p}_1, \ldots, \overline{p}_n). \quad (16)$$

We call this equational system $\mathsf{E}_{\mathcal{P}}$, the equational translation of $\mathcal{P}$.

"Gaussian Elimination"

A final and obvious step is to bind all the variables with the **fp**-operator. We assume that one predicate letter, say p_1 is the one to be queried. Dropping the overbars on relation variables, we rewrite the last equation of system (16) above as

$$p_n = \mathbf{fp} z_n . F_n(p_1, \ldots, p_{n-1}, z_n)$$

and substitute into equation $n - 1$, obtaining

$$p_{n-1} = F_{n-1}(p_1, \ldots, p_{n-1}, \mathbf{fp} z_n . F_n(p_1, \ldots, p_{n-1}, z_n))$$

Now we bind the (possibly multiple) occurrences of p_{n-1} and so on, eventually obtaining a single $\Re_\Sigma$-expression

$$p_1 = \mathbf{fp} z_1.F_1(z_1,\ldots) \tag{17}$$

where the right hand side is a closed (no-free-relation-variable) term. We will call the right-hand expression R_{p_1} and say that R_{p_1} *arises from translation of the predicate* p_1 *defined by the program* $\mathcal{P}$.

Definition 5.4 *A positive relation expression* $R \in \Re_\Sigma$ *(or the defining equation* $p = R$ *for some identifier* p*) is said to be a (relational)* **logic program** *if* R *is of the form* R_q*, i.e. it arises from the translation of a predicate* q *defined by a logic program* $\mathcal{P}$*.*

5.1.1 An Example

We illustrate the steps outlined above with an example. We start with the well-known program $\mathcal{P}_{add}$ defining addition of formal numerals over the Herbrand Universe for $\{o, s\}$:

```
add(o,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Just for reference, the Clark completion (minus the addition of Clark's equality theory) is:

$$\begin{aligned} add(x_1,x_2,x_3) \quad \longleftrightarrow \quad & \exists x(x_1 = o, x_2 = x, x_3 = x) \ \vee \\ & \exists x,y,z(x_1 = s(x), x_2 = y, x_3 = s(z), add(x,y,z)). \end{aligned}$$

After renaming the bound variables x, z as x_4, x_5, (y is already named by x_2) the normal form of $\mathcal{P}$is:

$$\begin{array}{c} add(x_1,x_2,x_3) \quad \longleftrightarrow \\ (\langle x_1,x_2,x_3\rangle = \langle o,x_2,x_2\rangle) \\ \vee \\ \exists x_4,x_5(\langle x_1,x_2,x_3,x_4,x_5\rangle = \langle s(x_4),x_2,s(x_5),x_4,x_5\rangle \\ \wedge\ add(x_4,x_2,x_5)). \end{array}$$

Letting ρ be the selector function $\langle 1,2,3,4,5\rangle \mapsto \langle 4,5,2\rangle$, and σ the associated permutation $\langle 1,2,3,4,5\rangle \mapsto \langle 4,5,2,1,3\rangle$, and writing $\vec{x}$ for $\langle x_1,x_2,x_3,x_4,x_5\rangle$, this can be rewritten:

$$\begin{array}{c} add(x_1,x_2,x_3) \quad \longleftrightarrow \\ \langle x_1,x_2,x_3\rangle = \langle o,x_2,x_2\rangle \\ \vee \\ \exists x_4,x_5(\vec{x} = \langle s(x_4),x_2,s(x_5),x_4,x_5\rangle \ \wedge\ add(\rho\vec{x})). \end{array}$$

The relational translation is:

$$\overline{add} \quad = \quad \dot{K}(o,x_2,x_2) \ \cup I_3[\dot{K}(s(x_4),x_2,s(x_5),x_4,x_5) \ \cap \ W(\sigma)\,\overline{add}\,W(\sigma)^o]I_3.$$

or

$$\overline{add} \quad = \quad \mathbf{fp}.\mathbf{Z}\,\dot{K}(o,x_2,x_2) \ \cup I_3[\dot{K}(s(x_4),x_2,s(x_5),x_4,x_5) \ \cap \ W(\sigma)\,\mathbf{Z}\,W(\sigma)^o]I_3.$$

6 Adequacy of the Translation

We now show the translation preserves the intended meaning of the program. To this end we need to define canonical interpretations of both the original program and of the induced equational system (16), or its associated closed form (17).

Let $\mathcal{P}$ be a Prolog program, $\Pi = \{p_1, \ldots, p_n\}$ the predicate symbols occurring in it, Σ its signature, and $T_{\mathcal{P}}$ the Kowalski-VanEmden continuous operator it induces on the power-set of the Herbrand base (see e.g. [Lloyd]). Let $T_{\mathcal{P}}\uparrow_0 = \emptyset$, $T_{\mathcal{P}}\uparrow_{(n+1)} = T_{\mathcal{P}}(T_{\mathcal{P}}\uparrow_n)$ and $T\uparrow_\omega$ the least fixed point $\bigcup T_{\mathcal{P}}\uparrow_n$ of $T_{\mathcal{P}}$, that is to say, the least Herbrand model of $\mathcal{P}$.

Define, for each $p_i \in \Pi$ and n in ω

$$\{\!\![p_i]\!\!\}^n = \{(u, u) : p_i(u) \in T_p\uparrow_n\}$$

and

$$\{\!\![p_i]\!\!\}^* = \{(u, u) : p_i(u) \in T_p\uparrow_\omega\}$$

Let $\overline{p_i}$ be relation variables associated with the predicate symbols p_i and let

$$\overline{p}_i = F_i(\overline{p}_1, \ldots, \overline{p}_n) \qquad (1 \le i \le n) \tag{18}$$

be the equational translation $\mathsf{E}_{\mathcal{P}}$ of $\mathcal{P}$. Let $\mathcal{F}$ be the set of open relational terms $F_j(\overline{p}_1, \ldots, \overline{p}_n)$, and $R_{p_i} = \mathbf{fp} z_i.F_i(\ldots)$ the associated closed form solutions of this equational translation.

Let $\Phi_{\mathcal{F}}$ be the induced operator on relational interpretations into the standard model $\mathcal{P}(\mathcal{T}_\Sigma^+ \times \mathcal{T}_\Sigma^+)$, as in the statement of theorem (2.13). By the same theorem, $\Phi_{\mathcal{F}}$ has a least fixed point $[\![\,]\!]^*$ which is the supremum of a chain of interpretations $[\![\,]\!]^0 = [\![\,]\!]^\perp, \ldots, [\![\,]\!]^n, \ldots$. We will drop subscripts in the discussion below, when speaking of a typical predicate symbol $p_i \in \Pi$ to simplify notation.

Lemma 6.1 *For each relation variable* $\overline{p}$ *and any interpretation* $[\![\,]\!]$

$$[\![\overline{p}]\!]^* = [\![R_p]\!]$$

Proof: Straightforward, using the semantics of the **fp** construct and the Tarski-Knaster theorem. ▮

If s is the arity of the program predicate letter p, define the s-th *restriction* $[\![\overline{p}]\!]|_s$ of $[\![\overline{p}]\!]$ by

$$[\![\overline{p}]\!]|_s = \{([u_1, \ldots, u_s], [u_1, \ldots, u_s]) : \exists x \in \mathcal{T}_\Sigma^+ \, ([u_1, \ldots, u_s, x], [u_1, \ldots, u_s, x]) \in [\![\overline{p}]\!].$$

Then we have the following Soundness and Completeness theorem for our translation:

Theorem 6.2 (Adequacy) *For each* $p \in \Pi$

$$\{\!\![\overline{p}]\!\!\}^* = [\![p]\!]^*|_s$$

Proof: We establish the two claims below by induction on n, from which the theorem follows immediately.

1. For each natural number n $[\![\overline{p}]\!]^n|_s \subseteq \{\!\![p]\!\!\}^*$.

2. For each natural number n $\{\!\![p]\!\!\}^n \subseteq [\![\overline{p}]\!]^*|_s$.

Claim (1): Since $[\![\overline{p}]\!]^0|_s$ is empty, the base case is immediate. Suppose the claim holds for n and $(u,u) \in [\![\overline{p}]\!]^n|_s$. The clause defining p in the normal form $\mathcal{P}_n$ is of the form

$$p(x_1,\ldots,x_n) :- \bigvee D_i \tag{19}$$

each D_i is of the form

$$D_i = \exists x_{n+1}\cdots x_w[\vec{x} = \vec{t}, q_{i1}(\rho_1\vec{x}),\ldots,q_{i\ell_i}(\rho_\ell\vec{x})]. \tag{20}$$

where the sequence $\vec{t}$, say of length m, is the concatenation $\vec{t_0}\cdots\vec{t_\ell}$ of arguments occurring in the clause

$$p(\vec{t_0}) : -q_{i1}(\vec{t_1}),\ldots,q_{i\ell_i}(\vec{t_\ell}). \tag{21}$$

of the original program $\mathcal{P}$.

The corresponding equation in $\mathsf{E}_\mathcal{P}$ is

$$\overline{p} = \bigcup \Theta_i$$

where

$$\Theta_i = I_s[\dot{K}(\vec{t}) \cap W(\sigma_1)\overline{q_{i1}}W(\sigma_1)^o \cap \cdots \cap W(\sigma_\ell)\overline{q_{i\ell}}W(\sigma_\ell)^o]I_s.$$

So our supposition $(u,u) \in [\![\overline{p}]\!]^n|_s$ implies that for some i $(1 \le i \le \ell)$ there is an extended term x, $(ux,ux) \in [\![\Theta_i]\!]^n$. Since $[\![I_s]\!]^n$ consists of formal vectors of length at least $s+1$ sharing their first s components, there is an extended term y such that the length of uy is at least one greater than that of $\vec{t}$, $(uy,uy) \in [\![\dot{K}(\vec{t})]\!]^n$ and for each j between 1 and ℓ, $(uy,uy) \in [\![W(\sigma_j)\overline{q_j}W(\sigma_j)^o]\!]^n$. But then, by corollary (4.3), for some ground substitution θ, $\langle (uy)_1,\ldots,(uy)_m\rangle = \vec{t}\theta$ (where m is the length of $\vec{t}$), from which we have the following consequences:

- $u = \vec{t_o}\theta$, in other words $p(u)$ is a θ-instance of the head $p(\vec{t_0})$ of the original program clause (21).
- $(\vec{t}\theta z, \vec{t}\theta z) \in [\![W(\sigma_j)\overline{q_j}W(\sigma_j)^o]\!]^n$ for each j between 1 and ℓ and some extended term z, hence (by lemma 4.10)
- $(\vec{t_j}\theta z', \vec{t_j}\theta z') \in [\![\overline{q_j}]\!]^n$, whence $(\vec{t_j}\theta, \vec{t_j}\theta) \in [\![\overline{q_j}]\!]^n|_{r_j}$ where r_j is the arity of q_j.

By the induction hypothesis $(\vec{t_j}\theta, \vec{t_j}\theta) \in \{\!\![q_j]\!\!\}^*$, whence (by definition of $\{\!\![\,]\!\!\}^*$), $q_j(\vec{t_j}\theta) \in T\uparrow_\omega$. But then every θ-instance of the tail of clause (21) holds in the least Herbrand model of $\mathcal{P}$, hence so does $p(\vec{t_0}\theta)$. But recall that $u = \vec{t_0}\theta$ so this means $(u,u) \in \{\!\![p]\!\!\}^*$, which is what we wanted to prove.

The other direction, which requires a symmetric argument, is left to the reader. ▮

6.1 Extending the Translation to Equations and Disequations

Enriched Terms

We will have need of a metalanguage of *enriched terms* for discussing certain operations on terms (such as unification and anti-unification). To this end, we introduce operators

$$\sqcap^2, \quad \neg^1$$

of the arities shown, and define an enriched term to be an expression of the form

$$t_1 \sqcap \ldots \sqcap t_n \sqcap \neg s_1 \sqcap \neg s_2 \sqcap \cdots \sqcap \neg s_m \tag{22}$$

where the t_i and s_i are terms (members of the term model of Σ), and where $m \geq 0$ and $n \geq 0$. Simply put: enriched terms are either un-enriched terms or they are built by application of the unary operator $\neg$ to un-enriched terms, and by iterated applications of the binary symbol $\sqcap$ to enriched terms. The binary operator associates to the right so that the typical enriched term may be written in the form (22).

6.1.1 Horn Clauses with Equations

We now consider programs comprised of finite sets of clauses of the form

$$p(t_1, \ldots, t_r) : -\mathbf{e} \,\square\, Tl$$

where $\mathbf{e}$, the *equality component*, is a finite set of *equality formulas*

$$u_1 \sim v_1, \ldots, u_m \sim v_m$$

with u,v terms, and $\sim$ one of the symbols $\{=, \neq\}$, and Tl is a pure Horn clause tail.

We now summarize the normalization procedure for such programs.

Head linearization: We first introduce fresh variables in the head, *à la* Clark, adding new equations to the equality component of the tail, and existentially quantifying all variables "free" in the tail, that is to say, occurring in the tail and not in the head.

$$p(x_1, \ldots, x_r) : -\exists \vec{y}[x_1 = t_1, \ldots, x_r = t_r, \mathbf{e} \,\square\, Tl].$$

The equality component is now the expanded sequence $x_1 = t_1, \ldots, x_r = t_r, \mathbf{e}$.

First processing of equality component: . Modify every equality formula $u \sim v$ in the equality component as follows:

1. If $u \sim v$ is $u = v$ where both u and v are variables, replace v with u everywhere in the tail, and remove v from the sequence of existentially quantified variables.
2. If $u \sim v$ and u is a variable, and the above conditions are not met, do not modify the equation.
3. Otherwise, replace $u \sim v$ with $z = u \wedge z \sim v$, where z is fresh and is added to the sequence of existentially quantified variables.

Linearization of the tail Now we proceed exactly as with pure prolog programs, replacing sequences of terms t_i occurring in atomic predicates in the tail with sequences of variables x_i (which may or may not be fresh, depending on whether Case 1 or Case 2 obtains in the linearization procedure described in section 5), and adding new equations $x_i = t_i$ to the equality component of the clause.

Renaming: finally all variables occurring in the tail are renamed in the same way as in the pure Prolog case, so as to form part of a master list $\vec{x} = x_1, \dots, x_n$. Each clause now looks like this:

$$p(x_1, \dots, x_r) : -\exists x_{r+1} \cdots x_n[x_{i_1} \sim t_{i_1}, \dots, x_{i_k} \sim t_{i_k} \,\square\, Tl'],$$

where the new tail Tl' is the linearized version of the old, and where the equality formulae $x_{i_j} = t_{i_j}$ include processed equality formulae from the original clause as well as equations added during the linearization steps. Unlike the pure Horn clause case, there may be repeated occurrences of variables on the left-hand side of the equality formulas. As before, we force all variables in the clause to occur on the left hand side of equality formulas, via the possible addition of identities $x_i = x_i$ if necessary.

Enrichment: We now replace all equality formulas of the form $x_{i_j} \neq t_{i_j}$ and rewrite then as $x_{i_j} = \neg t_{i_j}$. Finally we gather all equations (now possibly involving enriched terms) which *share a common variable* x on the left hand side

$$x = t_{j_1}, \dots, x = t_{j_s}$$

and write then as the single equation

$$x = t_{j_1} \sqcap \cdots \sqcap t_{j_s}$$

where some of the t's may be of the form $\neg u$, u a term.

At the expense of adding enriched terms, we have now brought the clause into a form where each variable occurring anywhere in the clause occurs exactly once on the left hand side of an equality formula, and where no occurrences of disequality remain.

$$p(x_1, \dots, x_r) : -\exists x_{r+1} \cdots x_n[x_1 = s_1, \dots, x_n = s_n \,\square\, Tl'].$$

Relational step: The clause is now translated to a relation expression in a manner very similar to the pure case

$$\overline{p} = I_s[\Delta(K)(s_1, \dots, s_n) \cap W(\sigma_1)\overline{q_1}W(\sigma_1)^o \cap \cdots \cap W(\sigma_\ell)\overline{q_\ell}W(\sigma_\ell)^o]I_s$$

where $\Delta(K)(s_1, \dots, s_n)$ is the *diagonal* $K(s_1, \dots, s_n) \cap id$, but where we now need to expand the definition of K to include sequences of enriched terms. This is discussed next.

Definition 6.3 *K maps enriched terms, and sequences of such to relation expressions as follows:*

$$\begin{array}{rcl} K(a) & = & (a,a)\mathbf{1} \\ K(x_i) & = & (P_i)^o \\ K(f(u_1, \dots, u_n)) & = & \bigcap_{i \le n} f_i^n K(u_i) \\ K(\langle t_1, \cdots, t_n \rangle) & = & \bigcap_{i \le n} P_i K(t_i) \\ K(\neg t) & = & di K(t) \\ K(t_1 \sqcap t_2) & = & K(t_1) \cap K(t_2) \end{array}$$

We now need to extend lemma 4.7 in the appropriate way. It should be noted that in the enriched case, the analogue of diagonal lemma (4.5) is of no use. We can no longer assume that sequences $\langle t_1, \ldots, t_n\rangle$ of enriched terms appearing in expressions of the form $\Delta(K)(\langle t_1, \ldots, t_n\rangle)$ are *clean* in the sense of that lemma. Thus the diagonal $\Delta(K)(\langle t_1, \ldots, t_n\rangle)$ may be a proper subset of $\dot{K}\langle t_1, \ldots, t_n\rangle)$, which is why we are forced to use the diagonal in the translation. With this choice, we obtain the correct translation of equational component of a logic program clause in the following sense.

Lemma 6.4 *Let $x_{i_1} \sim t_{i_1}, \ldots, x_{i_k} \sim t_{i_k}$ be a sequence of equality formulas, where the t_{i_j} are terms, and $\langle s_0, \ldots, s_n\rangle$ the sequence of enriched terms produced by the enrichement process described above, with $x_1, \ldots, x_n$ the list of all free variables occurring in the t_{i_j} (and hence the s_i). Then (where θ_u is the substitution $\{u_1/x_1, \ldots, u_n/x_n\}$):*

$$\begin{aligned}[\![\Delta(K)(\langle s_0, \ldots, s_n\rangle)]\!] \quad = \quad & \{(v, v) : v = uz \text{ where} \\ & u, z \in \mathcal{T}_\Sigma^+, u = [u_1, \ldots, u_n] \text{ and} \\ & x_{i_1}\theta_u \sim t_{i_1}\theta_u, \ldots, x_{i_k}\theta_u \sim t_{i_k},\end{aligned}$$

In other words, the diagonal of $K(\langle s_0, \ldots, s_n\rangle)$ consists of precisely those ground substitutions that satisfy the original equations and disequations.

With this lemma, it is easy to extend the adequacy theorem (6.2) to the equational Horn clause case. For details the reader is referred to [Chap].

7 Evaluation

The equational theory $\mathsf{E}_\mathcal{P}$ induced by a program $\mathcal{P}$ now opens up a new way of computing queries to logic programs of a quite general nature, via directed relational rewriting. In this section we sketch a rewrite system, together with a deterministic rewriting strategy that simulates SLD resolution of conventional prolog queries . It is our first concern to show that the conventional operational interpretation of the original program can be recovered relationally, before addressing alternative evaluations.

Suppose $\mathcal{P}$ is a program with predicate symbols $\Pi = \{p_1, \ldots, p_n\}$ and G is a query, say of the form

$$q_1(\vec{t_1}), \ldots, q_m(\vec{t_m}) \tag{23}$$

where the q_i are members of Π of arity $\alpha(i)$ and the $\vec{t_i}$ are tuples of terms of appropriate arities. Let the corresponding relation variables in the induced equational theory $\mathsf{E}_\mathcal{P}$ be $\overline{p_i}$.

Definition 7.1 *The translation A_G of a query G of the form (23) is the relation expression:*

$$\dot{K}(\vec{t_1} \ldots \vec{t_m}) \cap \overline{q_1}' \cap \ldots \cap \overline{q_n}'.$$

where $\overline{q_i}'$ is $W(\sigma_i)\overline{q_i}W(\sigma_i)^o$, σ_i being the permutation on $\{1, \ldots, m\}$ such that

$$\langle t_{\sigma_i(1)}, \ldots, t_{\sigma_i(n)}\rangle = \vec{t_i}\vec{t_1} \cdots \vec{t_{i-1}}\vec{t_{i+1}} \cdots \vec{t_m}$$

in other words, the permutation asociated with the i-th selector ρ_i, as in definition (5.2).

In the canonical semantics, the meaning of the translated query $[\![A_G]\!]^*$ is easily seen to coincide with the set of ground terms satisfying G in the least Herbrand model of $\mathcal{P}$.

Evaluation of the query G is simulated by rewriting the relation expression A_G according to certain strategy outlined below.

We have a choice to make in the way we describe evaluation. We may explicitly model recursion at the object level by using the fix-point operator in our syntax, replacing all relation variables by their closed form solutions, or we may handle it metalogically by dynamically replacing relation variables by their definitions when they are evaluated in rewriting. We adopt the latter approach here.

Termination occurs when the resultant term has no free variables (or, when closed forms are used, when the term is *recursion-free*: without occurrences of **fp**). We now describe the rewriting system and the evaluation strategy used, and sketch a proof of its soundness and completeness. It should be remarked that certain reductions in the system below are best called meta-reductions, since they involve rules that carry out in one step some rather intricate rewriting of expressions. For example, we carry out the rewriting

$$\dot{K}(t_1,\dots,t_n) \cap \dot{K}(s_1,\dots,s_m) \stackrel{P}{\longmapsto} \dot{K}(\theta t_1,\dots,\theta t_n) \tag{24}$$

$$\tag{25}$$

where

- the tuples $\vec{t} = (t_1,\dots,t_n)$ and $\vec{s} = (s_1,\dots,s_m)$ are standardized apart
- $m \leq n$ and
- θ is an mgu[4] of $(t_1,\dots,t_m)$ and $(s_1,\dots,s_m)$.

and

$$\dot{K}(t_1,\dots,t_n) \cap \dot{K}(s_1,\dots,s_m) \stackrel{P}{\longmapsto} \mathbf{0} \tag{26}$$

if no unifier of $(t_1,\dots,t_m)$ and $(s_1,\dots,s_m)$ exists.

These rules are sound with respect to $\mathsf{Rel}\Sigma$-interpretations (lemma 4.4).

Another meta-reduction is

$$W(\sigma)^o\dot{K}(t_1,\dots,t_n)W(\sigma) \stackrel{P}{\longmapsto} \dot{K}(t_{\sigma(1)}\dots,t_{\sigma(n)}) \tag{27}$$

$$I_k\dot{K}(t_1,\dots,t_n)I_k \stackrel{P}{\longmapsto} \dot{K}(t_1,\dots,t_k) \qquad \text{if } k \leq n. \tag{28}$$

which is sound with respect to $\mathsf{Rel}\Sigma$-interpretations by lemmas 4.9, 4.10, 4.11 and 4.12.

Each meta-reduction *abbreviates a sequence of term rewritings*. That is to say, is equationally derivable in the theory $\mathsf{Rel}\Sigma$. The chief aim of the translation presented in this paper is to exploit the benefits of variable free rewriting in compilation, so it is essential that, in practice, these reductions be carried out within the relational syntax. The presentation of this rewriting system and a proof of its correctness is straightforward, but lengthy. It will not be discussed further in this sketch.

We also use some bona-fide rewriting rules, which are just directed equations from the theory of Distributive Relation Algebras, or **fp**-algebras, hence obviously sound for $\mathsf{Rel}\Sigma$- and **fp**-interpretations.

[4]whose range is among the set of terms over the master list of variables $x_1,\cdots,x_m$

Table 1: Nondeterministic Logic Programming Reductions

$(A \cup B) \cap C$	$\overset{P}{\longmapsto}$	$(A \cap C) \cup (A \cap C)$
$A \cap (B \cap C)$	$\overset{P}{\longmapsto}$	$(A \cap B) \cap C$
$A \cap (Q \cup R)$	$\overset{P}{\longmapsto}$	$(A \cap Q) \cup (A \cap R)$
$A \cap \mathbf{fp}x.\mathcal{E}(x)$	$\overset{P}{\longmapsto}$	$A \cap \mathcal{E}(\mathbf{fp}x.\mathcal{E}(x))$
$\overline{p}_i$	$\overset{P}{\longmapsto}$	$F_i(\overline{p}_1, \ldots, \overline{p}_n) \qquad (1 \leq i \leq n)$

Table 2: Meta-reductions

$A \cap I_m Q I_m$	$\overset{P}{\longmapsto}$	$I_m[I_m A I_m \cap Q] I_m \cap A$
$A \cap W(\sigma)\, Q W(\sigma)^o$	$\overset{P}{\longmapsto}$	$W(\sigma)\, [W(\sigma)^o\, A W(\sigma)\, \cap Q] W(\sigma)^o$
$W(\sigma)^o \dot{K}(t_1, \ldots, t_n) W(\sigma)$	$\overset{P}{\longmapsto}$	$\dot{K}(t_{\sigma(1)} \ldots, t_{\sigma(n)})$
$\dot{K}(u) \cap \dot{K}(v)$	$\overset{P}{\longmapsto}$	$\dot{K}(\theta v) \qquad (\theta = \mathsf{mgu}(u, v), \parallel u \parallel \leq \parallel v \parallel)$
$\dot{K}(v) \cap \dot{K}(u)$	$\overset{P}{\longmapsto}$	$\dot{K}(\theta v) \qquad (\theta = \mathsf{mgu}(u, v), \parallel u \parallel \leq \parallel v \parallel)$
$\dot{K}(v) \cap \dot{K}(u)$	$\overset{P}{\longmapsto}$	$\mathbf{0} \qquad (u, v \text{ not unifiable})$
$I_s \dot{K}(t_1, \ldots, t_n) I_s$	$\overset{P}{\longmapsto}$	$\dot{K}(t_1, \ldots, t_s) \qquad (s \leq n)$

7.1 The Evaluation Strategy

The typical query

$$\dot{K}(\vec{t_1} \ldots \vec{t_m}) \cap \overline{q_1}' \cap \ldots \cap \overline{q_m}'.$$

is evaluated as follows:

1. Associate to the left (second reduction):

$$(\dot{K}(\vec{t_1} \ldots \vec{t_m}) \cap \overline{q_1}') \cap \ldots \cap \overline{q_m}',$$

   where, since $\overline{q_1}'$ is $W(\sigma_1)\overline{q_1}W(\sigma_1)^o$,

2. apply the modular law in the form of the second meta-reduction:

$$W(\sigma_1)(W(\sigma_1)^o \dot{K}(\vec{t_1} \ldots \vec{t_m}) W(\sigma_1) \cap \overline{q_1}) W(\sigma_1)^o \cap \ldots \cap \overline{q_m}',$$

   apply the third meta-reduction (which permutes the indices of the sequence $\vec{t_1} \ldots \vec{t_m}$)

$$W(\sigma_1)(\dot{K}(\vec{t}_{\sigma_1(1)} \ldots \vec{t}_{\sigma_1(m)}) \cap \overline{q_1}) W(\sigma_1)^o \cap \ldots \cap \overline{q_m}',$$

3. replace $\overline{q_1}$ by its definition $\Theta_1 \cup \overline{\Theta}_2$ where $\overline{\Theta}_2$ is $\bigcup_{2 \leq i \leq r} \Theta_i$ for some r:

$$W(\sigma_1)(\dot{K}(\vec{t}_{\sigma_1(1)} \ldots \vec{t}_{\sigma_1(m)}) \cap [\Theta_1 \cup \overline{\Theta}_2]) W(\sigma_1)^o \cap \overline{q_2}' \ldots \cap \overline{q_m}'.$$

   and

4. Distribute the first $\cap$ accross the union:

$$W(\sigma_1)([(\dot{K}(\vec{t}_{\sigma_1(m)} \ldots \vec{t}_{\sigma_1(m)}) \cap \Theta_1) \cup (\dot{K}(\vec{t}_{\sigma_1(m)} \ldots \vec{t}_{\sigma_1(m)}) \cap \overline{\Theta}_2)]) W(\sigma_1)^o \cap \overline{q_2}' \cap \ldots \cap \overline{q_m}'. \tag{29}$$

   Now we concentrate on the evaluation of $(\dot{K}(\vec{t}_{\sigma_1(m)} \ldots \vec{t}_{\sigma_1(m)}) \cap \Theta_1)$ which is an expression of the form (see (15)):

$$\dot{K}(\vec{t}_{\sigma_1(1)} \ldots \vec{t}_{\sigma_1(m)}) \cap I_s[\dot{K}(\vec{u}) \cap W(\tau_1)\overline{q_{i1}} W(\tau_1)^o \cap \cdots \cap W(\tau_\ell)\overline{q_{i\ell}} W(\tau_\ell)^o] I_s.$$

5. Now we apply the first meta-reduction:

   $$I_s[I_s\dot{K}(\vec{t}_{\sigma_1(1)}\ldots\vec{t}_{\sigma_1(m)})I_s \cap \dot{K}(\vec{u}) \cap W(\tau_1)\overline{q_{i1}}W(\tau_1)^o \cap \cdots \cap W(\tau_\ell)\overline{q_{i\ell}}W(\tau_\ell)^o]I_s \cap \dot{K}(\vec{t}).$$

   Observing that $\vec{t}_{\sigma_1(1)}$ is precisely the truncation of the sequence $\vec{t}_{\sigma_1(1)}\ldots\vec{t}_{\sigma_1(m)}$ to its first s components, and that $\vec{t}_{\sigma_1(1)}$ is precisely $\vec{t_1}$ (since the first selector permutation is in fact the identity) and applying the last meta-reduction:

   $$I_s[\dot{K}(\vec{t_1}) \cap \dot{K}(\vec{u}) \cap W(\tau_1)\overline{q_{i1}}W(\tau_1)^o \cap \cdots \cap W(\tau_\ell)\overline{q_{i\ell}}W(\tau_\ell)^o]I_s \cap \dot{K}(\vec{t}).$$

   Letting $\vec{u'}$ be the result of applying $\mathsf{mgu}(\vec{u},\vec{t_1})$ to $\vec{u}$, we obtain

   $$I_s[(\dot{K}(\vec{u'}) \cap W(\tau_1)\overline{q_{i1}}W(\tau_1)^o) \cap \cdots \cap W(\tau_\ell)\overline{q_{i\ell}}W(\tau_\ell)^o]I_s \cap \dot{K}(\vec{t}). \qquad (30)$$

   This brings us back to the evaluation of a basic form *term-sequence* $\cap$ *W(relation-variable)W*^{o}

   $$\dot{K}(\vec{u''}) \cap W(\tau_1)\overline{q_{i1}}W(\tau_1)^o$$

   so we proceed as before, left-to-right.

   As we are, in essence, reducing the entire SLD-tree, termination requires a finite SLD-tree. To simulate search along *one* branch, we would have to define *termination* to mean the reduction to a relation term *one of whose* unionands is relation-variable-free. This can be enforced with a different reduction strategy that we will not consider here. See [Ruhlen] for details.

7.1.1 Returning Answers

The meta-reduction rules have the effect of reducing every composition and intersection of a K-term $\dot{K}(\vec{t})$ that can occur in evaluation *to another* K-term. Thus a resulting variable-free expression will be a union of such terms. These expressions, which constitute a normal form of the rewriting system, can be returned to the user, or alternatively, they can be *printed* as a normalized constraint-set description of the solution. This notion of printing has been described in [BroLip]. Although this extra strength is not exploited in the results of this paper, the printing algorithm reduces *any* variable-free relation expression using the full relation-algebra connectives (that is to say, including negation) to a two-quantifier formula in which negation occurs only immediately before atomic formulas (equations between terms) using the quantifier elimination algorithm of Mal'cev [Mal'cev], also described in [Maher]. It is therefore able to return readable answers to a considerably stronger relational rewriting system in which arbitrary first-order constraints over the Herbrand Universe are used as queries and outputs. The quantifier elimination process is the basis of Constructive Negation [Chan, Stuckey].

7.2 Soundness and Completeness of Evaluation

Soundness of evaluation is an immediate consequence of the fact that every rewrite rule $R \overset{P}{\longmapsto} R'$ is equationally sound: in any interpretation into $\mathcal{P}(\mathcal{T}_\Sigma^+ \times \mathcal{T}_\Sigma^+)$

$$[\![R]\!] = [\![R']\!]$$

Suppose we are given a program $\mathcal{P}$ and a query G as in (23), with a finite SLD tree with leftmost selection rule. Since our reduction system is equationally sound, and the

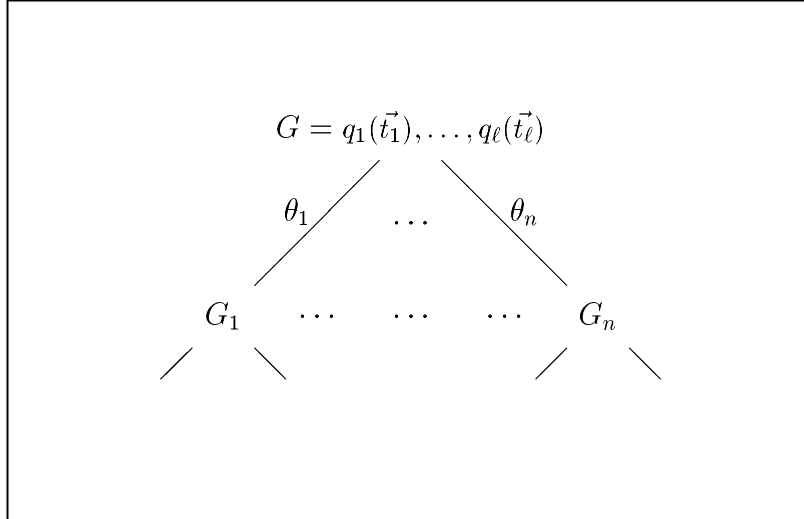
adequacy theorem shows that a pair of ground terms (u, u) is in $[\![A_G]\!]^*|_s$ precisely when the components of u make the query true in the least Herbrand Model of $\mathcal{P}$, the only way completeness can fail to hold is if evaluation of A_G fails to terminate. If it does terminate, i.e. A_G reduces to a closed term A, then we have the following:

1. $[\![A_G]\!]^* = [\![A]\!]^*$.
2. For s the number of terms in query G, $[\![A_G]\!]^*|_s = \{(u, u) : u$ ground and $\mathsf{M}_\mathcal{P} \models G(u)\}$ where $\mathsf{M}_\mathcal{P}$ is the least Herbrand model of $\mathcal{P}$.

Thus, in order to establish completeness it suffices to prove the following theorem.

Theorem 7.2 *Suppose $G = q_1(\vec{t_1}), \ldots, q_m(\vec{t_m})$ is a query for a logic program $\mathcal{P}$, and has a finite SLD tree with the leftmost selection rule. Let A_G be the relational query associated with G. Then there is a closed term A such that $A_G \overset{P}{\longmapsto} A$.*

Proof: By induction on the depth of the given SLD tree. Suppose G has the following finite SLD tree:



where each G_i is of the form

$$B_i\theta_i, q_2(\vec{t_2}\theta_i), \ldots, q_m(\vec{t_m}\theta_i).$$

and where B_i is the tail of the i-th clause in the program $\mathcal{P}$ whose head predicate is p_1:

$$p_1(\vec{u_{i0}}) : -q_{i1}(\vec{u_{i1}}), \ldots, q_{i\ell}(\vec{u_{i\ell}}).$$

Then G_1 is precisely the goal:

$$(q_{11}(\vec{u_{11}}), \ldots, q_{1\ell}(\vec{u_{1\ell}}))\theta_1, q_2(\vec{t_2}\theta_1), \ldots, q_\ell(\vec{t_\ell}\theta_1),$$

where θ_i is a most general unifier of $\vec{u_{10}}$ and $\vec{t_1}$.

The SLD tree below this goal is finite (and of smaller depth than the original goal), hence so is the SLD tree (with left-most selection rule) below the first subgoal $G_{11} = q_{11}(\vec{u_{11}}\theta_1)$.

Let $A_{G_{11}}$ be the relational translation of this goal, namely

$$\dot{K}(\vec{u_{11}}\theta_1) \cap q_{11}$$

We may apply the induction hypothesis to conclude that our evaluation strategy will reduce $A_{G_{11}}$ to a relation-variable-free A_{11}, which, in fact, must be the K-term corresponding to the computed answer substitution for this goal.

Recall (from 29) that evaluation of the original query A_G yields, in several steps, an expression of the form

$$W(\sigma_1)([(\dot{K}(\vec{t}_{\sigma_1(m)} \dots \vec{t}_{\sigma_1(m)}) \cap \Theta_1) \cup (\dot{K}(\vec{t}_{\sigma_1(m)} \dots \vec{t}_{\sigma_1(m)}) \cap \overline{\Theta}_2)])W(\sigma_1)^o \cap \overline{q_2}' \cap \dots \cap \overline{q_m}'.$$

the leftmost reducible subexpression of which reduces to (30)

$$I_s[(\dot{K}(\vec{u_{11}}\theta_1) \cap W(\tau_1)\overline{q_{11}}W(\tau_1)^o) \cap W(\tau_2)\overline{q_{12}}W(\tau_2)^o \cap \dots \cap W(\tau_\ell)\overline{q_{1\ell}}W(\tau_\ell)^o]I_s \cap \dot{K}(\vec{t}).$$

Since the first selector $W(\tau_1)$ is the identity, the leftmost reducible subterm $\dot{K}(\vec{u_{11}}\theta_1) \cap W(\tau_1)\overline{q_{11}}W(\tau_1)^o$ is precisely $A_{G_{11}}$ which reduces to A_{11}. We are left with (using primes to hide the switching relations W):

$$W(\sigma_1)(I_s[A_{11} \cap \overline{q_{12}}' \cap \dots \cap \overline{q_{1\ell}}']I_s \cap \dot{K}(\vec{t}) \cup (\dot{K}(\vec{t}_{\sigma_1(m)} \dots \vec{t}_{\sigma_1(m)}) \cap \overline{\Theta}_2))W(\sigma_1)^o \\ \cap \overline{q_2}' \cap \dots \cap \overline{q_m}'.$$

The stack of goals (relation-variables) present is bounded by the size of the original SLD tree, and for each one we argue in the same way, eventually producing a union of relation-variable free goals

$$A = A'_1 \cup \dots \cup A'_\nu$$

where ν is the number of success nodes on the fringe of the original SLD tree. ▮

8 Related and Future Work, and Conclusions

There have been other efforts in the literature to remove variables from Prolog, and to make evaluation and compilation more algebraic. Bellia and Occhiuto develop an algebra of programs that captures unification, rewriting and narrowing in [BelOcc]. Our work rests heavily on the fact that such an algebra can be found within the relation calculus, whose semantics is well-understood, and which admits natural extensions to higher-order and linear contexts, as well as a rich representation theory (see e.g. the treatment of Allegories in [FreySce]). The compilation process described here can be viewed as an application of the canonical inclusion of a regular category into its associated category of relations. The greater generality of the categorical framework opens the way to applying this technique to extensions of the logic and to constraints, as well as to a denotational treatment based on relations, which we are currently exploring.

Categorical treatments of logic programming provide alternative ways of algebraicizing the subject, which are, in a sense, variable free (see e.g. [AspMart, Diac, PowKin, FFL, NFDP, Pym]). Corradini and Montanari [CorMont] have given a categorical analysis of logic program execution in terms of transition systems. None of these approaches have as yet been applied to compilation or the definition of an abstract machine for logic programs, although this might be an interesting alternative to the work in this paper.

At this point there are a number of questions raised by this approach which we hope to address. Can a significant portion of our relational machine be captured with a Church-Rosser, strongly normalizing set of rewrite rules? Comon and Jouannaud and Kirchner's work [Comon, CoHaJo] on rewriting systems for unification and disunification suggests that this is quite feasible, as does the work of Bellia and Occhiuto, *op. cit.*

We would also like to exploit the rich semantics of relational formalisms to obtain new notions of observables, and abstract interpretation, as well as to extend the relational compilation to higher-order logic programming.

To some extent this work was a foray into the terrain of relation-based computing as a separate discipline, with logic programming as an extended case-study . Some instances of "pure" relational programming languages, of limited expressive power, were studied in [BroLip]. A useful rewriting system for the full relation calculus seemed a lot to ask for in the absence of some computational paradigm and we thought it would help, at the start, to anchor such a system in logic programming-inspired reductions.

Work by the RUBY group at Oxford on hardware [BroHut, JonShe], and on program synthesis via relations by Bird and de Moor[Algebra], Maddux[Maddux, RelSem], Naumann [Naumann] and Backhouse [Backh], to name a few of the many researchers in this field, suggests that the relational paradigm can provide significant computational insights at almost every level of the field. Exploration –via relations– of possible connections between program synthesis and a more general notion logic of programming seems particularly tempting.

References

[AspMart] A. Asperti and S. Martini. Projections instead of variables, a category theoretic interpretation of logic programs. In *Proceedings of the 6th International Conference on Logic Programming*, pages 337–352. MIT Press, 1989.

[WAM] Hassan Ait-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Series in Logic Programming. 1991.

[Backh] Backhouse, R, et. al., "A Relational Theory of Data Types," in *Workshop on Constructive Algorithmics: The Role of Relations in Program Development,* Utrecht University, 1990.

[Backus] Backus, J., "Can Programming be liberated from the von Neumann style?" in *ACM Turing Award Lectures*, ACM Press (Addison-Wesley), New York, 1987.

[BakRoe] de Bakker, J. and de Roever, "A Calculus for Recursive Program Schemes" , ed. Nivat, A., in *Automata, Languages and Programming*, 1973.

[BelOcc] Bellia, M., and Occhiuto, M. E., "C-expressions, a variable-free calculus for equational logic programming", in *Theoretical Computer Science* 107, Elsevier, 1993.

[Naumann] David A. Naumann "A recursion theorem for predicate transformers on inductive data types", *Information Processing Letters*, volume 50, 6, pp. 329–336, 1994.

[BirdMoor1] R S Bird, Oege de Moor and P Hogendijk. Generic programming with types and relations. *Journal of Functional Programming*, 6(1), 1996.

[Algebra] Bird, R. S. and De Moor, O. *Algebra of Programming.* Prentice Hall, 1996

[BroHut] Brown, C. and Hutton, G., "categories, Allegories and Circuit Design", in *Proceedings of the 9th Symposium on Logic in Computer Science*, IEEE, 1994.

[Broome86] Broome, P. *Transformations of Parallel Programs with Higher-order Relational Operators*, Ph. D. dissertation, Univ. of Delaware, 1986.

[Broome91] Broome, P. "Applications of Algebraic Logic to Recursive Query Optimization," *8th Army Conference on Applied Mathematics and Computing,* 1991.

[BrLi92] Broome, P., and Lipton, J., "Constructive Relational Programming", *Transactions of the 9th Army Conference on Applied Mathematics and Computing,* ARO-Report 92-1, 1992.

[BroJon] Brown, Carolyn and Jones, G., "Hardware Component Allegories", in Proc. LICS 1994, IEEE.

[Chap] *Compilation of Logic Programs to a Relational Machine*, Honors Thesis, Wesleyan University, 1997.

[Chan] Chan, D. *Constructive Negation Based On the Completed Database,* Logic Programming: Proceedings of the Fifth International Conference and Symposium Eds. R. A. Kowalski and K. A. Bowen, MIT Press, Cambridge, MA, pp 111–125, 1988.

[Clark] Clark, K., "Negation as Failure", in *Logic and Data Bases*, Gallaire and Minker eds., Plenum Press, New York, 1978.

[CorMont] Corradini, A. and Montanari, U. An algebraic semantics for structured transition systems and its application to logic programs. In *Theoretical Computer Science* 103. Elsevier, 1993.

[Comon] Comon, H. "Disunification, A Survey", in *Logic and Automation*, Lassez and Plotkin, eds., MIT, 1992.

[CoHaJo] Comon, H., Haberstrau, M., Jouannaud, J-P, "Decidable problems in shallow equational theories", in *Proc. 7th. Symp. of LICS*, IEEE Computer Society Press, 1992.

[clp94] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20, 1994.

[BroLip] Broome, P and Lipton, J., "Combinatory Logic Programming: Computing in Relation Calculi" in *Proceedings of the 1994 International Symposium on Logic Programming*, M. Bruynooghe, ed., M.I.T. Press, pp. 269-285, 1994.

[Colp] Lipton, J., "A Relational Foundation for Logic Programming", to appear.

[Diac] R. Diaconescu. *Category Semantics for Equational and Constraint Logic Programming.* PhD thesis, Oxford University, 1994.

[FriasMad] "Completeness of a Relational Calculus for Program Schemes", proceedings of LICS 98, IEEE, 1998.

[FFL] Finkelstein, S., Freyd, P. and Lipton, J., "Logic Programming in Tau-Categories", in *Computer Science Logic '94*, LNCS 933, Springer, pp. 249-263, 1995.

[Freyd92] e-mail manuscript, Categories, Relations and Computation bulletin board, Feb. 15, 1992.

[FreySce] Freyd,P. and Scedrov, A., *Categories,Allegories,* North-Holland, 1990.

[NFDP] Finkelstein, S., Freyd, P. and Lipton, J., "A New Framework for Declarative Programming", to appear in *Theoretical Computer Science*, Elsevier.

[LakRed] T. K. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Proceedings of the 1991* IEEE Symposium on Logic Programming, pages 202–220, 1991.

[JafLas] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of Symposium on Principles of Programming Languages.* ACM, 1987.

[JonShe] Jones, G. and Sheeran, M., "Circuit Design in RUBY". In *Formal Methods in VLSI Design*, ed. J. Staunstrup, North-Holland, 1990.

[Lloyd] Lloyd, J. W., *Foundations of Logic Programming*, second ed., Springer-Verlag, 1987.

[Maddux] Maddux, R., "Introductory course on relation algebras, finite-dimensional cylindric algebras, and their interconnections", in *Algebraic Logic* (Proc. Conf. Budapest 1988) ed. by H. Andreka, J. D. Monk, and I. Nemeti, Colloq. Math. Soc. J. Bolyai 54 North-Holland Amsterdam, 1991, 361–392.

[RelSem] Maddux R., Relation-Algebraic Semantics, in *Theoretical Computer Science*, Volume 160, June 1996.

[Maher] Maher, M., "A Complete Axiomatization of the Theories of Finite, Rational and Infinite Trees" *Proceedings of the third IEEE Symposium on Logic and Computer Science,* 1988.

[MdM96] R. McPhee and O. de Moor, "Compositional Logic Programming", in *Proceedings of the JICSLP'96 post-conference workshop: Multi-paradigm logic programming*, M. Chakravarty and Y. Guo and T. Ida, eds., Technische Universität Berlin, also PRG Report 96-28, Oxford, 1996.

[Mal'cev] Mal'cev, A.I.,"On The Elementary Theories of Locally Free Universal Algebras", *Soviet Math. Doklady*, 1961, pp.768-771.

[Miller] Miller, M., Nadathur, G., Pfenning, F. and Scedrov, A., "Uniform Proofs as a Foundation for Logic Programming." *Annals of Pure and Applied Logic,* 1990.

[Moor92] de Moor, Oege, *Categories, Relations and Dynamic Programming*, Dissertation, and Technical Monograph PRG-98, Oxford Computing Lab, Oxford, 1992.

[MycOk] A. Mycroft and R. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

[Peirce] Pierce, C.S., *The Logic of Relatives*, Studies in Logic. Johns Hopkins University, Baltimore, 1882, 235-277.

[PowKin] Power, J. and Kinoshita, Y., A new foundation for logic programming. In *Extensions of Logic Programming '96.* Springer Verlag, 1996.

[Pym] Pym, D., Functorial Kripke models of the $\lambda\pi$-calculus. Lecture at Newton Institute Semantics Programme, Workshop on Category Theory and Logic Programming, Cambridge, September 1995.

[deRoe] de Roever, W.P., "Recursion and Parameter Mechanisms: an Axiomatic Approach", Report IW 20/74, Matematisch Centrum, Amsterdam, 1974.

[Ruhlen] *Development of a Rewrite System for Relational Algebra Equations Modeling Horn Clause Resolution* Master's Thesis, Wesleyan University, 1997.

[Schmidt] Schmidt, G. and Ströhlein, T., *Relations and Graphs*, EATCS Monographs, Springer, 1993.

[Shep88] Sheperdson, J.C., Negation in Logic Programming, in *Deductive Databases and Logic Programming*, Jack Minker, Rd., Morgan Kaufman, Los Altos, CA, 1988.

[Stuckey] Stuckey, P. "Constructive Negation for Constraint Logic Programming" in Proceedings of the 6th IEEE Symposium on Logic and Computer Science, New York, pp.328-339, 1991.

[TarGiv] Tarski, A. and Givant, S., *A formalization of set theory without variables,* Colloquium publications, V. 41, American Mathematical Society, Providence, RI, 1987.

Relational Programming in Libra

Barry Dwyer
University of Adelaide, South Australia, 5005.

Abstract

Libra is a general-purpose programming language based on the algebra of binary relations. It attempts to unify functional and logic programming, retaining the advantages of both, and avoiding some of the problems. It has all the features needed of a programming language, and a straightforward semantic interpretation. Since program specifications are easily expressed as relations, it offers a simple path from a specification to a program and from the program to its proof of correctness. The algebra of binary relations has several operators whose effects are like those of familiar procedural language constructs, for example, relational composition is analogous to sequential execution. The Libra language is illustrated by its application to a simple programming exercise. Some conclusions are drawn.

1 Introduction

This paper is a condensed version of a technical report [3] describing the Libra programming language. An implementation of Libra is available from

`ftp://ftp.cs.adelaide.edu.au/pub/dwyer/libra`

or via the author's web page at

`http://www.cs.adelaide.edu.au/~dwyer`

1.1 What is a Binary Relation?

In mathematics, the expression $X\rho Y$ is true if X and Y satisfy the relation 'ρ'. For example $X < Y$ is true if X and Y satisfy the relation '$<$'. There are three ways the '$<$' relation can be considered:

1. As the (infinite) set of all (X, Y) pairs for which $X < Y$.
2. A predicate that can be applied to (X, Y) pairs.
3. As a 'relator' (in the sense of [10, 11]) that, given X, will yield all Y values greater than X.

In Libra, a relation to describe changes in temperature could be defined as follows:

```
transition->{'Cold','Warm';'Warm','Hot';'Hot','Warm';'Warm','Cold'}.
```

The braces enclose a set of 4 elements, each of which is an ordered pair of terms. The relation is given the name 'transition' — or rather the name 'transition' maps onto the relation. The two means used in this example: set and pair formation, are Libra's only means of structuring data. Members of sets are unordered, but pairs are ordered.

There are three ways this relation could be used:

1. It could generate the four pairs of values.

2. It could be used to test whether a pair of values satisfy the relation.

3. Given the first term of a pair or pairs, it could give the corresponding second terms as a result. For example, given `'Warm'`, it could yield both `'Cold'` and `'Hot'`.

The first two ways of looking at relations are shared by all sets. The third view puts the relation in a more active role, and is described as 'applying' the relation to an argument to generate one or more values as a result. This is analogous to the view we take of applying functions in a functional programming language. Libra distinguishes these three roles syntactically.

To generate the members of 'transition', we write:

```
? @ transition.
('Cold','Warm')
('Warm','Hot')
('Warm','Cold')
('Hot','Warm')
```

The '`@`' operator generates each element of a set, in arbitrary order. It may be read as 'all' or 'for all'. In the existing Libra system, successive elements are generated by back-tracking, but in principle they could be generated in parallel. We may imagine that the computation splits into several threads: four in this case. The threads are independent, and cannot communicate with one another.

To test set membership, we may write:

```
? ('Warm','Hot') ? transition.
'True'
```

The '`?`' operator may be read as 'is a member of' and corresponds to the mathematical symbol '$\in$'.

To apply a relation to an argument, we write:

```
? 'Warm' ! transition.
'Hot'
'Cold'
```

There is an important distinction to make here. Applying a relation to an argument generates several result values, two in this case. It does not yield the set, `{'Hot'; 'Cold'}`. Applying relations that generate many values is computation intensive rather than storage intensive, but there are ways for a programmer to trade time for space if desired.

Libra does not support a fourth mode of using relations: given the second term of a pair, to determine its corresponding first terms. Accordingly, the first term of a pair is called its argument, and the second term is called its value. It is possible to go from argument to value, but not in general from value to argument — any more than existing programming languages allow inputs to be derived from outputs. This is stressed by the alternative notation for pairs: using '`->`':

```
transition -> {'Cold'->'Warm'; 'Warm'->'Hot';
               'Hot'->'Warm'; 'Warm'->'Cold'}.
```

from which it will be seen that the name 'transition' and the relation it defines are also an ordered pair.

1.2 Why Binary Relations?

Why should a programming language be based on the algebra of binary relations? One view is that it is an attempt to combine the advantages of both functional programming and logic programming.

An advantage of pure functional programming is 'referential transparency': the idea that a program is an algebraic expression, which when simplified, yields the value of the program, i.e., its output. An aspect of this idea is that functions can be given names, and that any occurrence of the name can be replaced by the corresponding function.

A difficulty that besets functional programming languages is that a function always has exactly one value. This makes it hard to deal with exceptions: dividing a number by zero yields no result, yet finding the square root of a number yields two results. This difficulty is avoided in a logic language such as Prolog, which can produce no result by 'failing', or produce two results by back-tracking.

On the other hand, Prolog has its own difficulties. Although a subset of Prolog can be understood in terms of propositional calculus, most Prolog programs need to use extra-logical predicates, which can only be understood with reference to a specific model of program execution. A language based on the algebra of relations can combine the best of both approaches, while avoiding some of their problems.

A different view is that relations are to general-purpose programming languages what matrices are to scientific languages. They are large scale data structures, which can be manipulated as wholes. In the scientific field, matrices have led to the evolution of specialised parallel computer architectures such as vector processors, but there is no similar well-established concept that has formed the basis of parallel architectures in more diverse fields — unless one counts the n-ary relations used in databases. Perhaps binary relations will provide such a concept. (The reader may recall that the original Connection Machine [4] was based on similar ideas, but it failed to exploit the full power of relational algebra.)

A third view is that relational algebra contains a rich supply of operators useful to programmers. For example, a common programming exercise is, given a list of words, to display the positions of each word. In the algebra of binary relations, if W is the list of words, the result is essentially given by the expression W^{-1}.

A fourth view is that relations include functions as special cases. They are like multi-valued functions, and can do anything that functions can. They are also known to be a universal model for data representation, being the building blocks of relational databases.

In Libra, there is no distinction between a relation as data or a relation as an operator, except how it is used. The syntax of Libra is such that, the limitations of the ASCII character set aside, any legal program is a valid expression in the algebra of binary relations that specifies the program's behaviour. This gives Libra a flavour similar to the specification language 'Z'. Libra is not an attempt to imitate 'Z', but it makes it easy to derive a Libra program from a 'Z' specification.

1.3 Some Previous Work

The Libra language owes its inspiration to Sanderson's Relator Calculus [10, 11]. However, there is a fundamental shift of viewpoint from his work. Sanderson regarded programming constructs such as 'if...else' and 'while...do' as having proved their worth, and based the Relator Calculus on similar constructs. Their disadvantage is that their mathematical properties are messier than the similar concepts of union and closure, except when the

program is functional.

Another source of inspiration is the work of MacLennan [5, 6], who wrote several relational programs using the language RPL. RPL was really a functional language that could operate on relational data structures. Its control structure did not exploit backtracking or parallelism. It also had the blemish that different operators were used to denote the same operation, depending on whether it acted on relations expressed as data (extensional relations) or as program (intensional relations). It was also awkward to mix operations on intensional and extensional structures.

This defect of RPL was addressed by Drusilla [1], which allows the same operator to be used uniformly on intensional or extensional relations, or a mixture of both. Drusilla also distinguishes the three modes of using relations identified earlier: as sets of pairs, as predicates that can be applied to pairs, and as means of generating results from an argument. The Drusilla compiler uses an extension of Milner type checking [7] to deduce the correct ways of combining relations from these properties. The same process also performs conventional type checking, detecting potential programming errors. In this respect Drusilla is superior to Libra, which leaves type checking until execution time. On the other hand, Libra allows operators to be overloaded, so that, for example, the programmer can extend the built-in '+' operator, which originally applies only to integers, to apply to vectors and matrices. Overloading is natural to a relational language, where arguments can map to several values, but it is complex with Milner type checking, because the types of operands and results are deduced from the types of operators, which therefore have to be fixed [9].

The 'Z' specification language [8] has also been an influence on Libra. Libra includes many constructs found in 'Z'. Libra could serve as a specification language. It is possible to turn a Libra specification into an executable program using theorems of discrete mathematics.

The immediate precursors to Libra are a language proposal written by the author [2], and Hydra [9], a partial implementation of it. Like Drusilla, the proposal tried to rectify defects of MacLennan's RPL, but adopted solutions different from Drusilla's in almost every case. This is probably because Drusilla has a functional programming background, whereas Libra is based on logic programming. Both Drusilla and Libra attempt to meld the functional and logic styles, but start from opposite ends of a spectrum. Hydra was influenced by Drusilla, and revealed several problems in the language proposal.

An important difference between Drusilla and Libra is their treatment of iteration. Drusilla simulates iteration by recursion — which is to be expected of a functional language — whereas Libra uses transitive closure. A typical Libra program is free of recursion, making its structure more transparent. Recursion can simulate 'go to' statements. Badly used, it can make a program just as hard to understand as they do. Thought was given to banning recursion in Libra. It remains available — in case of emergency. Another difference is that Libra uses sets and pairs as its basic structuring operations, whereas Drusilla uses lists and tuples. Libra's approach seems more appropriate to a relational language. The elements of sets have no particular order, which means that they can be dealt with in parallel, at least in principle. Operations on different elements can never interact with each other, again simplifying understanding of the program.

Another Libra feature worthy of note is 'reduction', which enables the elements of a set or sequence to be combined under a suitable binary operation. For example, reducing a set of numbers under '+' forms their total. A special case of reduction is to choose an arbitrary member of a set. This is useful in problems that admit of many solutions, but where only one solution is needed.

Libra is intended to assist the development of relational programming by encouraging experiment. To this end, the current interpreter is designed for simplicity and ease of extension and modification.

2 The Libra Language

2.1 Basic Syntax

Libra syntax is based on that of Prolog. This aided its implementation, but has sometimes resulted in compromise. Libra's lowest-level terms are variables, integers, atoms, delimiters and operators. Variables and integers are defined as in Prolog. All variable names begin with a capital letter.

There are two kinds of atoms. Those beginning with capitals, such as `'Warm'`, are called 'literals', and stand for themselves. (They are enclosed in quotes to distinguish them from variable names.) Literals give meaningful names to discrete values. The only pre-defined literals are `'True'` and `'False'`, but the programmer can invent new ones. All other atoms are called 'names'. There are three kinds of names: strings beginning with a small letter and comprising letters, underscores or digits, any string enclosed in quotes that is not a literal, and strings of the symbols `+-*/\^<>=:.?@#$&`. A name maps to one or more expressions. Referential transparency means that wherever a name appears, it can always be replaced by any of the things it stands for. Operators are merely a notational convenience. As in Prolog, any name may be defined as an operator. The expression `1+2` is indistinguishable from `+(1,2)`. Both are interpreted as relational application, i.e., as `(1,2)!(+)`. It may seem strange to treat '+' as a relation rather than a function, but a function is merely a special case of a relation.

2.2 User Commands

The Libra programming environment is a command loop in which the user may choose one of the following actions:

- To create mappings from names to expressions.
- To evaluate an expression. If there are several results, all are displayed.
- To display the definitions associated with a name.
- To drop existing definitions, either wholly or in part.
- To add new operators to the language.
- To load prewritten programs.
- To dump the current set of definitions.
- To obtain on-line help.

A typical program file consists of a series of definitions followed by a query. A definition has the form '*name* `->` *expression.*', and a query has the form '`?` *expression.*'

2.3 Structures

Libra's basic structuring operations are set formation and pair formation.

Sets are written as lists of elements between braces '`{}`', separated by semicolons. `{2;3;1}` and `{1;2;2;3}` both denote the same set. The members of sets may be structured and may even be of different kinds, e.g., `{(1,2);{1,2};1;'True'}`. A shorthand is provided for ranges of integers: `{M..N}` denotes the set containing the integers M to N inclusive.

In contrast to 'extensional' sets whose members are listed explicitly, it is also possible to define an 'intensional' set by a means of a pattern and a predicate. For example:

```
{(X,Y) : X<Y}
```

denotes the set of all (X, Y) pairs such that X is less than Y, i.e., it is the same as the relation '`<`' itself.

There is an important restriction on a set defined in this way. It is possible to test a given element for membership of it, but it is not possible to generate its members. Libra cannot deduce what set is implied by a pattern and a predicate. Such sets are called 'filters' to distinguish them from those whose members are listed explicitly, which are called 'generators'. A generator can be used as a filter, but a filter can't be used as a generator.

Ordered pairs are defined by an argument and a value, e.g.:

```
Arg -> Value
```

or

```
Arg, Value
```

maps the input `Arg` onto the output `Value`. They are formed by the infix operators '`,`' and '`->`', which differ only in that '`->`' binds its operands more loosely than '`,`'. Having a choice of operators saves parentheses.

2.4 Binary Relations

Binary relations are merely sets of pairs, e.g:

```
swap -> {X,Y -> Y,X}.
```

Defines a relation that swaps a pair of values. It is possible to use this relation as a filter to test if one pair is the reverse of another, for example:

```
? ((1,2), (2,1)) ? swap.
'True'
```

(The first (unary) '`?`' may read as 'evaluate', the second (binary) '`?`' tests for set membership, and is equivalent to '$\in$'.)

It is not possible to use `swap` as a generator. However, it allows a third mode of use: as a relator [10, 11] or 'constructor'. The `swap` relation can be applied to the pair `(1,2)` to construct the pair `(2,1)` as follows:

```
? (1,2) ! swap.
(2,1)
```

It is possible to write what appears to be a ternary relation, e.g:

```
ternary -> {X,Y,Z : X<Y<Z}.
```

But this is identical to:

```
ternary -> {X,(Y,Z) : X<Y<Z}.
```

This means that ternary relations, quaternary relations, and so on, do not exist in Libra, although it is usually harmless to pretend that they do.

2.5 The Ranks of Relations

There is a hierarchy of relations in Libra: 'generators', 'constructors', and 'filters'. As we have just seen, a constructor may be used as a constructor or a filter. A generator may used as a generator, a constructor or a filter. A filter may be used only as a filter. The distinction is made as follows. A generator contains no variables, but constructors and filters do. In a constructor, only the first term of a pair may introduce new variable names, which may be reused in the second term. A filter contains a pair whose second term introduces new variables. If a relation contains several pairs, its lowest ranking pair determines the rank of the relation.

There are no parameters associated with the name of a relation; each pair carries its own argument pattern:

```
max -> {X,Y -> X : X>Y; A,B -> B : B>=A}
```

In addition to the means of definition given so far, which are simply those for sets, constructors also allow the second term of a pair to be defined by an expression. For example:

```
succ -> {X -> X+1}.
```

maps `succ` to a relation that adds one to its argument.

2.6 Sequences

Sequences are relations whose arguments occupy the range 1–n, for some $n \geq 0$. Sequences are written as lists enclosed in square brackets; for example `[97,98,99]` denotes the relation `{1,97; 2,98; 3,99}`. 'Strings' are sequences whose terms are characters, i.e., small integers. For convenience, `[97,98,99]` may also be written as `"abc"`.

The empty sequence is denoted by `[]`. It is the same object as the empty set, denoted by `{}`. The expression `[M..N]` denotes the sequence whose first term is the integer M and whose last term is the integer N. So that:

```
[3..5]
```

denotes the relation:

```
{1->3; 2->4; 3->5}
```

Sequences, sets, and pairs may be nested as required, e.g.,

```
{(1,2); {3;4}; [5,(6,7)]}
```

denotes the set containing the pair `(1,2)`, the set `{3;4}`, and the two-term sequence `[5,(6,7)]`.

2.7 Application

Applying a relation to an argument generates outputs:

```
? 3 ! {X -> X-1; X -> X+1}.
2
4
```

If a relation is applied to an argument that fails to unify with any of its argument patterns, it has no value, and is said to be 'inapplicable'. Even if it unifies, a relation may be inapplicable if a predicate following '`:`' yields `'False'`.

There are several other operators closely related to relational application. One is called 'functional application' (`~`), and is written thus:

```
? max~(1,3).
3
```

It differs from relational application only when the relation yields several values; functional application yields only one, arbitrarily. Among other things, it is useful when a problem has several solutions, but any solution will do.

There is an asymmetry between the evaluation of the two operands of the application operators. The argument is always fully evaluated, but the relation is evaluated lazily — only those values matching the argument are evaluated. For example, given the definition:

```
factorial -> {0->1; N->N*factorial(N-1):N>0}.
```

and the query:

```
? factorial(2*3-6).
1
```

The expression `2*3-6` is fully evaluated to yield 0, which then unifies with 0 in the `factorial` relation. The argument also unifies with `N` in the second term, but fails the predicate `N>0`. Thus the expression `N*factorial(N-1)` is not evaluated. This illustrates an important point. A program typically defines a complex relation — often infinite. If the whole program had to be evaluated (i.e., its input-output relation was computed) before it could be applied to an argument, Libra would not be a practical programming language.

2.8 Relational Operators

Since relations are sets, they may be combined using set operators, such as `join`, `meet` or `omit`. In addition, there are several operators that apply only to relations:

The composition $A \circ B$ of relations A and B is denoted by `A o B` (the letter '`o`'). The expression `X!(g o f)` has the same effect as `f(g(X))` or `(X!g)!f`. Relational composition is similar to sequential execution in a procedural language. It is guaranteed that the second relation is applied after the first.

Two operators, `else` and `but`, provide the operations of 'relational extension' (analogous to functional extension) and 'over-ride' ($A \oplus B$). If `A` is applicable to an input argument, `A else B` maps it to the output values of `A`, and `B` is ignored. However, if `A` is inapplicable to the argument, `A else B` maps it to the outputs of `B`.

The expression `A but B` is equivalent to `B else A`. It is often used when it is desired to derive a relation that is like an existing one, except in some particular, e.g: `{X -> X but [2!X, 1!X]}` swaps the first two elements of `X`.

The inverse A^{-1} of A (denoted in Libra by `A^-1`) is such that (X, Y) is a pair in A^{-1} if and only if (Y, X) is a pair in A. Asking for the inverse of a relation is to ask what arguments can produce a given value. The relation: `square -> {X -> X*X}` has as its inverse the relation that finds the positive and negative square roots of the perfect squares. Although the inverse of a generator is a generator, and the inverse of a filter is a filter, since Libra cannot solve equations, the inverse of a constructor is only a filter.

There are four 'restriction' operators. Each modifies a relation by restricting its arguments or values according to membership of a set. The left restriction operator '`<?`' is such that `s<?r` restricts the domain of `r` to arguments in `s`. It comprises those pairs in `r` whose first terms are members of `s`. The left anti-restriction operator '`<\?`' is such that `s<\?r` restricts `r` to arguments *not* in `s`. The right restriction operator '`?>`' is such that `r?>s` restricts the codomain of `r` to values in `s`. The right anti-restriction operator '`\?>`' is such that `r\?>s` restricts `r` to values *not* in `s`.

2.9 Sequence Operators

Sequences are relations, so any set or relational operator can be applied to them. However, they have a few operators of their own.

Sequences may be concatenated, using the operator '`&&`', so that `"ab"&&"cd"` is the string `"abcd"`.

It is easy to find the n-th element of a sequence, e.g., `2!"abcd"` and `"abcd"~2` both yield 98. The expression `[2..3]o"abcd"` simplifies to `"bc"` — illustrating how a substring can be selected from a string. It is also straightforward to convert a single character to and from its ASCII equivalent; `1!"a"` = `97`, and `[97]` = `"a"`.

Given the argument `[5,6,7,8]`, the built-in function `head` returns the term `5`, the function `tail` returns the sequence `[6,7,8]`, the function `last` returns the term `8`, and the function `front` returns the sequence `[5,6,7]`.

The built-in `sort` function takes a set as argument and yields a sequence whose terms are the elements of the set in Prolog's standard order. For example:

```
? {2;3;1}!sort.
[1,2,3]
```

The `sort` function always sorts into ascending order. However, the postfix `<-` operator reverses a sequence, so that:

```
? ({2;3;1}!sort)<-.
[3,2,1]
```

is a combination that sorts a set into descending order.

2.10 Homogeneous Relational Operators

Relations that have the same argument and value types are called homogeneous. They can be applied to their own results, allowing transitive closure — an important concept in Libra, analogous to iteration or search.

The infix '`^+`' operator is used to apply a relation to its own output a fixed number of times; `R^+2` is equivalent to `R o R`, `R^+3` is equivalent to `R o R o R`, and so on (R^2,

R^3, etc. in mathematical notation). `R^+1` (R^1) is simply `R` itself, and `R^+0` (R^0) is the identity function on the domain of `R`.

The postfix '`^+`' operator forms the transitive closure of a homogeneous relation (R^+ in mathematical notation). It is defined by the infinite union:

$R^+ = R^1 \cup R^2 \cup R^3 \ldots$

There is an important restriction on the relations whose closure Libra can compute: they must be acyclic. Libra's implementation of transitive closure is not sophisticated enough to recognise that in exploring longer and longer paths, no new terms are added to the closure.

It is entirely possible to devise an algorithm that finds the transitive closure of a cyclic relation. One way is, when exploring a path, to test whether the next vertex to be added to the path is already on it. Another way is to test whether a newly generated pair is already part of the result. Why aren't these methods built into Libra?

There are three reasons: The first is that transitive closure is often used where cycles obviously cannot occur:

```
? (0,1) ! ({M,N -> N,M+N}^+).
(1,1)
(1,2)
(2,3)
..
```

(the Fibonnaci series). Testing whether a cycle has occurred would be an unnecessary overhead in such a situation.

The second reason is that the transitive closure of a relation can be very large. Complex search problems can be modelled by closures. A typical search has the form:

```
? start ! (improve^+ ?> solution).
```

which finds all solutions by repeatedly using improve to transform the initial state, `start`, until it is in `solution`. Keeping track of the states generated by `improve^+` might use all available storage.

The third reason also relates to search problems: it is usually necessary to record how a solution has been reached. This means storing the sequence of moves chosen. This sequence becomes part of the input and output of `improve`, so it would defeat an automatic cycle detector. On each iteration around a cycle, the sequence of moves grows longer. However, a built-in cycle detector could not distinguish this growing sequence from the rest of the state, and would consider each iteration to generate a new state.

The '`^+`' operator always applies its relation at least once. The similar '`^*`' operator forms the reflexive transitive closure of a relation:

$R^* = R^0 \cup R^1 \cup R^2 \cup R^3 \ldots$

A closure operator often needs both the termination condition and its complement to be written, so a limit (`^^`) operator is provided for greater convenience. Mathematically, the limit $R^\wedge$ of relation R yields the pairs that are in the reflexive transitive closure of R, but to whose second term R is inapplicable. In terms of graphs, the limit operator finds all the paths that cannot be extended further.

The inverse A^{-1} of relation A (denoted by `A^-1`) is such that (X, Y) is a pair in A^{-1} if and only if (Y, X) is a pair in A. The '`^-`' operator can take any non-negative exponent. The expression `A^-N` is directly equivalent to `(A^-1)^+N`; i.e., it yields all paths of length N in the reversed relation.

2.11 Higher-Order Relations

A higher-order relation is one whose values are other relations. The following example defines a class of functions that add a number to their arguments:

```
add -> {X -> {Y -> X+Y}}.
```

When `add` is given the argument ‘1’,

```
? add(1).
{(A,(1 + A)}
```

it yields a function that adds 1 to its argument.

‘Set reduce’ (`>>->`) is an important higher-order operator that applies its second operand, an associative commutative binary operator, to combine the values of its first, e.g.,

```
? @{1;2;3;4}>>->(+).
10
```

Each second operand yields a different relation, e.g.,

```
? @{1;2;3;4}>>->(*).
24
```

Reduction has an important property: it first generates, then gathers together several threads into a single result. Reduction provides the only way threads interact. The symbol for set reduction emphasises this many-to-one aspect.

The sequence reduction operator, `>>=>`, operates on sequences in a similar way. Its second operand need not be commutative. One use of `>>=>` is to flatten a sequence:

```
? ["abc","def","xyz"]>>=>(&&).
"abcdefxyz"
```

Any binary operator can be ‘amplified’ [10] using the ‘zip’ operator, `\\`:

```
? ([1,2,3],[4,5,6])\\(+).
[5,7,9]
```

The effect of this is to add the terms pairwise.

The `\\` operator may be used to define new operators, or to overload existing ones:

```
'+' -> {R1,R2 -> (R1,R2)\\(+)}.
```

defines an operator that is capable of vector addition — among other things. Because of the way `\\` and the built-in `+` operator are defined, there is no ambiguity about which `+` applies in a given case; `\\` can only apply to relations, and the built-in `+` operator only applies to integers.

3 An Example Program

The syntax and semantics of Libra are fully defined elsewhere [3], but to give the flavour of Libra, this section explains the development of a simple program. It illustrates the use of relations both as data and as program objects.

Consider the following planning problem:

> A farmer has with him a sack of corn, a chicken, and a rather vicious dog. He reaches a river, which he must cross in a small boat. The boat has only space enough for the farmer and one item. He must therefore ferry the corn, chicken and dog from the left bank to the right bank of the river one item at a time. The problem is that he cannot leave the dog alone with the chicken, for it will certainly eat it, nor can he trust the chicken alone with the corn. How can he ferry them all across safely?

We may sketch a solution immediately. Starting with an initial state in which everything is on the left bank of the river, the program must choose a sequence of moves that result in everything being moved to the right bank. This suggests the following:

```
solution -> (initial_state!safe_move^+) = final_state.
```

We assume that `safe_move` is a relation that applied to a state to give a new state, such that the new state is 'safe', i.e., nothing gets eaten. It is a relation rather than a function, because several moves are possible from a given state. The transitive closure operator (`^+`) will compose all possible sequences of choices to implement a search.

This scheme has a basic flaw: if the problem has a solution, it will simply yield `'True'`. To be useful, the program should generate plans showing how the problem is solved. A plan could be a sequence of either moves or states — or both. In the solution given here, the plan is a sequence of states, from which it is easy to deduce the moves.

A second problem is that the transitive closure operator lets a program become trapped in a cycle. For example, the farmer could ferry the chicken back and forth across the river for ever. The way to avoid such cycles is to make sure that each new state added to the plan is not already part of it. This is why it is better to record the plan as a sequence of states rather than a sequence of moves. A solution should not pass through the same state twice, but it might need to make the same move several times.

The solution should therefore have the form:

```
solution -> [initial_state]!add_to_plan^+ ?> solved.
```

Initially, the plan consists of a single term: the initial state, which we include in the plan because we want the program to check that it doesn't return to it. Since the test for completion is no longer a simple equality, we use right restriction to make sure the finished plan is in the set `solved`.

We may now elaborate `initial_state`:

```
initial_state -> (everything, {}).
everything -> {'Farmer'; 'Dog'; 'Corn'; 'Chicken'}.
```

A state is a pair whose first member is the set of things on the left bank of the river, and whose second member is the set of things on the right bank. We need to keep to track of the position of the farmer, but it is not necessary to worry about the boat; where the farmer goes, the boat goes too.

The problem is solved by all plans whose last term is the desired final state:

```
solved -> {Plan : last(Plan) = final_state}.
final_state -> ({}, everything).
```

States may be added to the plan by generate and test:

```
add_to_plan -> suggest o verify.
```

where `suggest` generates a possible state, and `verify` ensures that it is not already in the plan.

The argument of `suggest` is the existing plan, and its output should include the new state, but in addition it needs to copy the existing plan — otherwise `verify` could not check the new state or add it to the plan.

```
suggest -> {Plan -> Plan, last(Plan)!cross_river}.
```

This definition extracts the last state from the plan, and uses `cross_river` to generate new states.

The `verify` relation is straightforward, remembering that a sequence of states is a set of (integer, state) pairs. The set of states already visited is the codomain of the plan:

```
verify -> {Plan, State -> Plan && [State] : State \? codom Plan}.
```

This appends the new state to the existing plan. We now must explain why we didn't write, in one step:

```
add_to_plan -> {Plan -> Plan && [last(Plan)!cross_river]
              : last(Plan)!cross_river \? codom Plan}.
```

The expression `last(Plan)!cross_river` generates a new state. We add it to the plan, provided that it is new. Unfortunately, the expression appears twice, and `cross_river` being a relation, there is no guarantee that it will yield the same value in each place. Therefore the state being added to the plan is not necessarily the one that proved to be new. The two references to the variable `State` in the earlier two-step approach ensure that the same state is used in both places.

The `cross_river` relation operates on states rather than on plans. Crossing can occur left-to-right or right-to-left:

```
cross_river -> left_to_right join right_to_left.
```

Ferrying an object from left to right is a two step process consisting of first choosing an object on the left bank, then moving it to the right bank. As with `add_to_plan`, using a connecting variable ensures that the object taken from the left bank is the same object added to the right bank. Whichever object is chosen, the farmer must go with it:

```
ferry_object -> {Left, Right -> Left, Right, @Left}
             o {Left, Right, Choice
                             -> Left omit {Choice} omit {'Farmer'},
                                Right join {Choice} join {'Farmer'}}.
```

Since the farmer is on the left bank, the farmer can be the chosen object. If so, the farmer is removed from the left bank twice, and added to the right bank twice, but since we are dealing with sets, this won't matter. When the farmer is the chosen object, this models his crossing the river alone.

Crossing from left to right occurs only when the farmer is on the left bank, so we check the argument of `ferry_object` to ensure it is in the set `farmer_on_left`:

```
left_to_right -> farmer_on_left <? ferry_object \?> unsafe.
```

We use right anti-restriction to ensure the result is not in `unsafe`, e.g., by leaving the dog alone with the chicken.

The name `farmer_on_left` maps to a set, defined as follows:

```
farmer_on_left -> {Left, Right : 'Farmer' ? Left}.
```

Next, we define the set of unsafe states. Assuming the right bank was in a safe state after the previous move and will certainly be safe once the farmer reaches it, we only need to concern ourselves with the safety of the left bank. There are three unsafe situations, when the dog is left with the chicken, when the chicken is left with the corn, or when all three are left together. This may be expressed as follows:

```
unsafe -> {Left, Right : Left includes {'Chicken'; 'Corn'}
                       v Left includes {'Dog'; 'Chicken'}}.
```

This completes the strategy for moving things from left to right. To program moving things from right to left we have a choice: to write a new relation analogous to the `left_to_right` relation, to write a generalised 'transfer' relation that accepts 'Left' and 'Right' as arguments, or to notionally exchange left and right:

```
swap -> {Left,Right -> Right,Left}.
```

Armed with which, the `right_to_left` relation is simply:

```
right_to_left -> swap o left_to_right o swap.
```

That is the problem solved. Typing:

```
? solution.
```

causes all solutions to be displayed.

4 Type Checking

Because operators may be overloaded, it is sometimes necessary to test the type of the argument of a relation. Types are simply sets, and any set expression can be used as a type. Types may be checked using the set membership operator, e.g., `X ? integers` is true if `X` is an integer.

Some useful sets are built-in:

- integers
- naturals
- positives
- characters
- booleans
- literals
- sets

- relations
- sequences
- strings
- any

A few of these sets are defined as generators, although their main use is as filters. For example, `X ? integers` yields `'True'` if and only if `X` is an integer. On the other hand, the expression `@integers` will (start to) generate all the integers.

'Naturals' is the set of non-negative integers, and 'positives' is the set of positive integers. 'Characters' is the set of integers from 0 to 255. These are all generators, as is 'booleans', which generates the set `{'True'; 'False'}`.

The remaining sets are filters. The set 'literals' includes all atoms beginning with a capital letter — including `'True'` and `'False'`. The set 'sets' contains sets of all kinds, including relations, sequences and strings. 'Relations' contains only those sets that consist of ordered pairs. 'Sequences' are functions whose arguments range from 1 to n. 'Strings' are sequences whose values are characters. 'Any' denotes the universal set; `X ? any` is always true.

Because any valid set expression may be used to define a type, some of the built-in sets are related, for example:

```
sets -> sets_of any.
sequences -> sets_of (positives x any).
```

It is also possible to define new types:

```
integer_pairs -> integers x integers.
```

5 Program Execution

It is important to know how a Libra program is executed, for two reasons: to know what ranks of relation (generator, constructor or filter) allow execution to be possible, and to be able to make some estimate of program complexity.

A Libra program is best thought of as a collection of threads of control. A thread of control usually carries an argument with it. When a relation is applied to the argument, each result generates a new thread of control. In the current interpreter each result thread is explored in turn by back-tracking, but it is better to consider that they are all executed in parallel; there is no way for the program to communicate between the threads, and the order in which they will be executed is unpredictable anyway. A second way in which several threads can arise is by a thread invoking the '`@`' (for all) operator, which generates a thread for each member of a set. Closure operators apply a relation to its own results. If the relation is a function, yielding a single result, the effect is like a loop. But if the relation yields several threads, closure increases their number exponentially.

What mechanisms remove threads? The simplest is when a relation is inapplicable to its argument. A thread reaches the relation, but no result emerges. This is typical of a 'generate and test' strategy, or pruning during a tree search. A similar effect is achieved by the restriction operators, which kill off threads according to whether their arguments are members of a given set or not.

Reduction first multiplies threads and then reduces their number to one. When a thread of control reaches a reduction operator, it generates a thread for each value of its first operand, which can either generate a set, or apply a multi-valued relation to an argument. These threads exist within an envelope that allows them to be collected together again and combined using the second operand of the operator. (The current implementation relies on Prolog's 'findall' predicate.) Thus, one thread emerges to match the one that reached the construct — or, if the collection proves empty, none emerges at all. Functional application (`f~X`) is a special case of reduction, where the combining operation has the form `{(X,Y)->X}`, i.e., one result is chosen arbitrarily. Through this operator, Libra introduces non-determinism.

It is usually important for a program to terminate. It can fail to through uncontrolled recursion or transitive closure. The programmer must ensure that a closure terminates without being trapped in a cycle. There is little the programmer can assume about the order of execution. Libra does not guarantee that alternatives will be chosen in any particular order. For example, the expression:

```
? 0 ! {X->X+1; X->X-1}^* .
```

is capable of generating all the integers — in principle. Depth-first search might generate 0,1,2... or 0,-1,-2,.... A better strategy would be breadth-first search, generating the series 0,1,-1,2,-2,... . However, Libra does not promise to do either of these things, but might alternately add and subtract one, cycling between the values 0 and 1 for ever.

What does Libra promise about closure? Only this: a result cannot occur until after an argument that generates it. In the above example, 3 is either the child of 2 or of 4, so 3 cannot be generated until after 2 or 4 has been generated. (Since 4 cannot be generated until after 3 or 5 has been generated, it is easy to prove that 2 must be generated before 3.) This sequential property derives from the composition operator. In the expression `X!r o s`, first `r` is applied to `X`, then `s` is applied to the results. This is guaranteed in any implementation of Libra.

Another programming consideration is the asymmetry between the treatment of the operands of 'apply' (`!`). The first operand is always evaluated as fully as possible, but the second is always evaluated as little as possible. Normally, the first operand will contain no variables, and is said to be 'grounded'. If it does contain variables, and cannot be reduced to a grounded form, it will passed in 'symbolic' form, i.e., still containing variables, although it may become partially simplified. This allows a constructor or filter to be passed as an argument to a higher order relation. However, if a first operand can be reduced to a grounded form, it always will be. Passing an expression as a first operand to some other relation is one way to force its evaluation.

The programmer must be aware that each time any relation is applied to an argument its results are calculated afresh. If a complex relation is likely to be applied to the same argument several times, it may be worth evaluating the relation first, and storing it as a set of argument-value pairs — provided it is not too large. There are several other ways of forcing evaluation. For example, finding the inverse of a relation currently forces the relation to be evaluated.

It is difficult to summarise the rules governing the ranks of relations needed by different operators in different modes of use. For example, if `A meet B` is used as a generator, `A` must be a generator, but `B` may be a filter. However, if `A join B` is used as a generator, both `A` and `B` must be generators. As a general rule, the first operand will need to have at least the same rank as the mode of use, so that a generator is needed to generate a set, a constructor is needed to construct a result, but only a filter is needed to test membership.

The second operand may need the same or a lower rank, depending on the operator. If a relation has insufficient rank, an error will be detected during program execution. As a rule of thumb, the programmer is advised always to place the higher-ranking operator first, and hope for the best. Currently, the only safe alternative is to consult the source program of the interpreter.

6 Scope Rules

A program is a set of named definitions of relations. It is itself a relation from names to expressions. In the current implementation of Libra, all definitions are global in scope.

Variable names are local to the elements of a relation or set. For example, in the definition:

```
change -> {X -> X+1;X -> X-1}.
```

the first two occurrences of `X` are independent of the last two. They are also independent in the following construct:

```
? 1 ! {X -> X+1} o {X -> X+1}.
3
```

If they were not, the argument `1` would not only bind the first relation to the pair `(1,1+1)`, but the second relation to `(1,1+1)` as well, so the second relation would not be applicable to the intermediate result (i.e., 2).

A relation or set that has another set as an element may cause a potential ambiguity. For example, what should the following definition do?

```
neighbours -> {X -> {X-1; X+1}}.
```

The intention seems clear. We would expect:

```
? neighbours(1).
{0; 2}
```

The rule is adopted that a variable name has scope throughout the set element in which it occurs, including any sets occurring within the element. This means caution is needed in defining higher-order relations. For example, the intention of the following example is to define a function that will evaluate relation `R` over the domain `X`, thus producing a generator:

```
eval -> {X,R -> {@X!{X -> X,X!R}>>->}}.
```

However, in the construction `{X->{X,X!R}}`, `X` is bound to the first argument of `eval`, but the programmer's intention was for `X` to be local. Libra will detect the problem and issue a warning. The problem is easily corrected:

```
eval -> {S,R -> {@S!{X -> X,X!R}>>->}}.
```

7 Discussion

What, if anything, has been learnt from implementing an interpreter for Libra?

7.1 Type-Checking and Inapplicability

The first lesson — which came as a surprise to the author — is that although the relational programs developed were quite short, they were surprisingly tricky to debug. Some of the difficulties arose because the interpreter was under development, some were due to the author's lack of practice in relational programming, and some due to the design of Libra itself. The main problem is the notion of inapplicability. If a relation is presented with an argument to which it does not apply, it simply produces no result. It is fundamental to relational programming that this should occur, but it means that if a programming error leads to the wrong type of argument being passed, then the whole program typically produces no result. It is not easy to deduce where the error lies from an empty output.

There would seem to be several solutions to this problem. They all involve some kind of type-checking. Some checking is already done by Libra's built-in relations. If the wrong kind of argument is passed to one, a warning results. However, once the relation is overloaded by the programmer's own definition, Libra turns off its warning, because it cannot tell which definition is meant to be effective in any given case. Even if the programmer's definition proves inapplicable, Libra can't know whether it is an error.

Although the definition of a relation can verify the type of its argument, this only serves to restrict it further. Arguments that do not pass the type test are inapplicable, so the situation is unchanged. It would be possible for a programmer to define a relation so that if its argument had the wrong type, it wrote an error message. However, this is not a modular solution, because overloading the name to map to a relation accepting a different type of argument (perhaps by a separate package) would require the error condition in the first definition to be modified.

The solution adopted by Drusilla [1] is to infer data types statically from the types of operators, and in turn, to deduce the types of programmer-defined relations. In its basic form, this means that the built-in operators cannot be overloaded. However, it is possible to imagine an extension of this idea to allow for overloading. Indeed, Drusilla does precisely this in treating generators, constructors and filters as different types. It seems to the author that allowing overloading of names is the only logical decision in a language that allows arguments to yield multiple results.

The author's personal view is that static type-checking is unduly restrictive. Although it can detect many programming errors, it is just one of many checks that could be made. Programs can fail because of division by zero or other arithmetic errors, or by entering an infinite loop or infinite recursion. However, nobody suggests that we should insist that all such errors should be detectable statically. Indeed, if we devise a language in which all programs are guaranteed to terminate, we know it is not Turing-complete, and less than universal. I believe the same kind of objection applies, in a subtler way, to languages that are statically type-checkable. What the exact problem is, I don't know, but I believe it is the reason why many artificial intelligence languages avoid static type-checking, and why systems programming languages usually have type loop-holes.

One way to solve these problems would be to make a distinction between 'inapplicability' and 'no result'. If no definition is applicable to an argument, it could be treated as a programming error; or, the definition could remain applicable but deliver a special null result.

7.2 The Domain Problem

This leads naturally to a related problem: the question of what defines the domain of a relation. A relation maps from its domain to its codomain. In mathematics, this presents no particular difficulty because a relation is a static object. In a programming language, there are three possibilities for the domain or codomain: their types, their potential values, or their actual values. The distinctions are that the type of a domain or codomain might be the integers, its potential values might be the prime numbers, and its actual values for some given state of execution might be 2, 3 and 5. The 'Z' specification language [8] distinguishes two of these cases: the values of a domain are chosen from a 'source' type and the values of a codomain from a 'target' type. It might be argued that the set of potential values and the type of an object are the same thing. This is true if the language has a flexible enough way of specifying types. Libra treats types as ordinary sets, so — depending on one's point of view — it either offers a very flexible means of type definition, or none at all. However, in most statically typed languages the means for defining types aren't flexible enough to allow 'prime numbers', say, to be declared as a type.

We have already discussed one situation where these distinctions cause a problem: Libra can't tell whether a relation is meant to apply to a given argument. If Libra could determine that an argument was in the domain of the relation but that the relation did not apply, it could consider this a normal result; but if it could determine that the argument wasn't in the domain of the relation — or in the case of overloading, in the domain of any definition of a name — it could consider it an error.

A second aspect of the domain problem arises in finding the reflexive transitive closure (`^*`) or limit (`^^`) of a relation. Here, it is best to visualise the graph of the relation. The graph contains a number of edges, and the reflexive transitive closure consists of all paths of length zero or more. In addition to the paths found by transitive closure (`^+`), Libra also adds to the closure a loop linking each vertex to itself. The set of vertices is found from the edges of the graph: each edge must leave and enter a vertex of the graph. However, it is possible for a graph to contain isolated vertices. Libra has no way of knowing what these are.

A solution is to force each relation to specify its domain and codomain. Perhaps this might be written as follows:

```
left_to_right ->
        (state x state):farmer_on_left<?ferry_object\?>unsafe.
```

Since Libra sets are dynamic objects, domains and codomains could be used as arbitrary assertions, to help debugging. Naturally, any solution to the domain problem would also solve the applicability problem of Section 7.1.

Another advantage of specifying types is to optimise data representations. For example, if the result of a relation is known to be of type 'string', its representation could be made an efficient one for strings rather than a general-purpose one that treats the result merely as a set of pairs.

7.3 Referential Transparency

In the implementation of Libra, a decision was made to evaluate the arguments of relations — using 'simplify' — before applying them, except for the built-in relations, which try to achieve the same effects more lazily. Originally, this was motivated by an analogy with procedural languages, which typically require their input parameters to be fully evaluated,

but which evaluate conditional constructs lazily. On reflection, it is seen perhaps to be an arbitrary decision. For example:

```
and -> {A,B -> A&B}
```

defines an `and` relation exactly like the built-in `&` relation, except that it always evaluates both operands. It would clearly be an advantage if no operand were ever evaluated until its value was needed. It would seem to follow that arguments should be passed by reference rather than by value, and should not be simplified until necessary.

Such a change would have an important consequence. Consider the definition:

```
double -> {X -> X+X}.
```

which is intended to double the value of its argument. Under the existing implementation of Libra, this operates as follows:

```
? @{1;2}!double.
2
4
```

However, if `simplify` wasn't called until `X` was evaluated, the effect would be to evaluate:

```
? @{1;2}+@{1;2}.
2
3
3
4
```

This is certainly referential transparency, but is it sensible? In Section 3, in connection with the relation `add_to_plan`, we explained exactly why two instances of the same variable should be given the same value. In that example, we wanted the program to make sure that the move it added to its plan was the same move that it had just tested for safety. But the result is that this destroys the notion of referential transparency, at least as it is usually understood.

It is possible to argue — as some functional programming languages do — that variables are an unnecessary concept, and should be eliminated from the language. But to the author, this would be simply sweeping the problem under the carpet. To achieve the effect of `double`, it would be necessary to have some way of duplicating an argument. Suppose there was a built-in relation called `duplicate` as follows:

```
duplicate -> {X -> X,X}.
```

Then we could define `double` as:

```
double -> duplicate o (+).
```

without using variables. But how are we to know that the two values generated by `duplicate` are the same? Surely,

```
? @{1;2}!duplicate.
```

should be equivalent to:

```
? (@{1;2},@{1;2}).
```

which generates:

```
(1,1)
(1,2)
(2,1)
(2,2)
```

In favour of the current interpretation, the language of Conceptual Graphs [12], one of whose aims is to present a variable-free version of the predicate calculus, introduces variables into its linear notation for precisely this purpose: to indicate when two occurrences of an expression refer to the same object. (In its graphical notation, this is done simply by pointing at the same vertex twice.)

Although variable names are potentially an evil no better than the infamous 'goto', the author believes that in the restricted context of a relational definition they are harmless, even beneficial. After all, the alternative is to use built-in relations such as `left` and `right`, effectively defined as:

```
left -> {L,R -> L}.
right -> {L,R -> R}.
```

to dissect structured arguments. Libra variables are merely local names for compositions of such relations.

Choosing when to evaluate expressions can be important for other reasons. In Prolog, the expression `1+2` means exactly that, and is not evaluated to yield `3` unless the programmer chooses. One result is that Prolog can manipulate expressions symbolically. A second is that the programmer has more control over execution. It would be useful to have a similar feature in Libra.

7.4 Input and Output

The current treatment of files within Libra is very unsatisfactory, and should be improved as soon as possible. Part of a remedy would be to treat files as relations. A possible set of file operators was suggested in the language proposal on which Libra is based [2].

One of the properties of files is that they can be modified. Updating a file has a side-effect on how it will be read, which is not expressed by the Libra programming language. The core problem is that Libra programs — like functional programs — are expressions, not procedures taking place in time. It is not easy to see the best way to integrate Libra with the idea of a time-dependent state.

The problems of file input-output are still more manifest when dealing with the user of a Libra program. Dialogues generated by several parallel threads could become confusingly intertwined; question and answer would need to be treated atomically. What happens if several threads decide to ask the user the same question? Should each question be answered separately, or should Libra ask the question once and remember the answer? In a relational language, only the first choice makes sense, since it allows multiple results, although it might sometimes prove inconvenient. For that matter, perhaps even a single question should be allowed to have multiple answers.

7.5 Nested Scopes and Modules

In the language proposal on which Libra was based [2], it was proposed that names might be given limited scope. The reason for this is that in defining a module or library of

relations for general use, it might be useful to define local relations that would not be of general interest. A programmer using the library module should not have to worry about them. There is a danger that the programmer will define a new relation with the same name as one in a module, and on invoking it, find that spurious results are produced.

One possible way of avoiding this problem would be to introduce a `where` operator, so that:

```
{add2 -> add1 o add1} where {add1 -> {X->X+1}}.
```

would have the effect of making `add2` global, but making `add1` private to the declaration. In general, a set of global definitions should be able to access a set of local ones.

Simple as this idea is, it is not currently implemented. For one thing, it is hard to see how to integrate it with the one-definition-at-a-time approach of the current command interpreter — whose serial view of the program text is already inappropriate to a relational language.

Another aspect to consider is the object-oriented approach to programming. In the context of Libra, where objects are unimportant, many of the problems that object-oriented languages solve aren't present. On the other hand, it would be useful to have some concept similar to inheritance. This means that if a general type of object has been defined, and a more specialised type of object has been based on it, the specialised object should be able to inherit the relations from the more general object, or over-ride them with its own. For example, if the `number_of_legs` relation applied to members of the set `mammal` yields '4', we would want to over-ride this for the set `humans` to yield '2'. Libra does not provide any means of over-riding one relational definition by another; if two definitions apply, *both* take effect. Nor does Libra have any way at present of using the knowledge that `humans` are `mammals`.

A partial solution to this problem would be to use `else` rather than `join` to link definitions of relations having the same name. If a name defines several relations, they could be tried in turn. As soon as a definition is found that applies to the argument, no further definitions would be tried. Some means of specifying the search order is needed. Letting the search order be defined by the sequence of the program text seems a poor approach, but Libra does not currently offer any other language feature that could be exploited. A better suggestion, which fits well with the nature of relations, is to allow the program text to contain modules that are relations from names to definitions, and compose them using an arbitrary relational expression.

7.6 Efficiency

Treating data and program text in a uniform way causes a major problem in the implementation of Libra. Consider an operation such as composition. The expression `R o S` should compile to a Prolog rule something like:

```
'R o S'(X,Z) :- R(X,Y), S(Y,Z).
```

if `R` and `S` are static program objects, but to:

```
'o'(R,S,'R o S').
```

if they are data objects (where `o` is a built-in predicate). The present implementation represents everything as data, which deals with static objects very poorly. The converse, of expressing data objects as facts, means that a Prolog implementation would spend much

of its time asserting and retracting facts. A better solution would be to have different approaches for static and dynamic objects.

One such approach is to follow Drusilla, and choose suitable representations during compilation. This preserves a uniform syntactic treatment for static and dynamic objects — although problems arise when they are mixed. Higher-order relations are also a problem because they may need to accept both static and dynamic arguments at different times. Catrall [1] seems to make no mention of how Drusilla's higher-order relations are compiled.

In order to make sensible space-time trade-offs, the programmer needs to distinguish clearly between dynamic (data) and static (program) objects. Perhaps they could be distinguished syntactically, e.g:

```
compose -> {R,S -> {X ->X!(R o S)}}.
```

could define a program object that could be applied to an argument (`X`), leading to the translation above; whereas:

```
compose -> {R,S -> R o S}.
```

might define a data object. However, this approach raises new problems: e.g., finding the composition of a static and dynamic relation.

References

[1] D.M. Cattrall, *The Design and Implementation of a Relational Programming System*, PhD Thesis, Dept. of Computer Science, University of York, 1992.

[2] B. Dwyer, "Programming Using Binary Relations: a proposed programming language", *Technical Report 94-04*, Dept. of Computer Science, University of Adelaide, 1994.

[3] B. Dwyer, "LIBRA: A Lazy Interpreter of Binary Relational Algebra", *Technical Report 95-10*, Dept. of Computer Science, University of Adelaide, 1995. (Also available via `http://www.cs.adelaide.edu.au/~dwyer`.)

[4] W.D. Hillis, *The Connection Machine*, MIT Press, 1985.

[5] B. J. MacLennan, "Relational Programming", *Technical Report NPS52-83-012*, Naval Postgraduate School, Monterey, CA, 1983.

[6] B. J. MacLennan, "Four Relational Programs", *Technical Report NPS52-86-023*, Naval Postgraduate School, Monterey, CA, 1983.

[7] R. Milner, "A theory of type polymorphism in programming", *Journal of Computer and System Sciences*, 17(3) pp348–375, 1978.

[8] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, Prentice-Hall 1991.

[9] J.D. Prosser, "A Programming Language Based on the Algebra of Binary Relations", *Honours Report*, Dept. of Computer Science, University of Adelaide, 1994.

[10] J.G. Sanderson, "A Relational Theory of Computing", *Lecture Notes in Computer Science 82*, Springer-Verlag Berlin 1980.

[11] J.G. Sanderson, "Relator Calculus", *Technical Report 84-02*, Dept. of Computer Science, University of Adelaide, 1984.

[12] W.M. Tepfenhart, J.P. Dick, J.F. Sowa (Eds.), "Conceptual Structures: Current Practices", *Lecture Notes in Computer Science 835*, Springer-Verlag Berlin 1994.

Modelling Message Buffers with Binary Decision Diagrams

Bernd–Holger Schlingloff
Universität Bremen, TZI-BISS

1 Introduction

Binary decision diagrams (BDDs, [Bry92]) have been recognized as an extremely efficient data structure for the representation of transition relations in the verification of finite-state reactive systems. With BDDs, it is possible to represent relations over domains with more than 2^{100} elements ([BCDM91]), provided the represented relation is well-structured. Asynchronous parallel systems such as communication protocols often use implicit or explicit buffering of messages which are sent between the processes. In these notes, we analyze the complexity of various possibilities to model the transition relation of a bounded buffer with BDDs, and discuss alternative approaches to this problem.

2 Binary Decision Diagrams

To make these notes self-contained, we quickly describe the symbolic representation of sets and relations with BDDs. For a detailed survey, the reader is referred to [Bry92]. Consider a sequence of variables $V \triangleq (v_1, ..., v_k)$ over domains $(D_1, ..., D_k)$, where each D_i is finite. An *ordered decision diagram* (ODD) or *deterministic branching program* for V is a tuple $(N, \mathcal{L}, E, n_0)$, where

- N is a finite set of nodes,
- $\mathcal{L} : N \to V \cup \{\top, \bot\}$ is a labelling of nodes,
- $E \subset N \times D \times N$ is a set of edges ($D = \bigcup_i D_i$), and
- n_0 is the initial node.

The following conditions are imposed:

- E is functional on D_i: If $\mathcal{L}(n) = v_i$, then for each $(n, d, n') \in E$ it holds that $d \in D_i$, and for each $d \in D_i$ there is exactly one n_d such that $(n, d, n_d) \in E$, and
- E is acyclic: If $(n, d, n') \in E$ with $\mathcal{L}(n) = v_i$ and $\mathcal{L}(n') = v_j$, then $i < j$.

It is easy to see that this definition is equivalent to the one given, e.g., in [Bry92]. Any ODD accepts (defines) a subset of $(D_1 \times ... \times D_k)$ via the following definition:

$$(N, \mathcal{L}, E, n_0) \models (d_1, ..., d_k) \quad \text{if} \quad (N, \mathcal{L}, E, n_0) \models_1 (d_1, ..., d_k).$$

In this definition, the notion $\models_m$ is declared by:

$(N, \mathcal{L}, E, n) \models_m (d_1, ..., d_k)$ if

- $\mathcal{L}(n) = \top$, or
- $\mathcal{L}(n) = v_i$ and $m < i$ and $(N, \mathcal{L}, E, n) \models_{m+1} (d_1, ..., d_k)$, or
- $\mathcal{L}(n) = v_i$ and $m = i$ and $(n, d_m, n') \in E$ and $(N, \mathcal{L}, E, n') \models_{m+1} (d_1, ..., d_k)$.

In other words, given a specific tuple, it can be determined whether it belongs to the set represented by an ODD by traversing its edges according to the components of the tuple.

When drawing ODDs, we usually omit the node labelled $\bot$ and all edges leading to it. For example, the ODD with two variables v, v' over $D_1 = D_2 = \{a, b, c, d\}$ given in Figure 1 below represents the set of tuples $\{(a, a)$, (a, b), (a, c), (a, d), (b, b), (b, d), (c, c), (c, d), (d, a), $(d, d)\}$. *Binary* decision diagrams (BDDs) are ODDs where all domains are $\{0, 1\}$. Given any ODD, there exists a BDD of the same order of size which represents the same set: Choose any binary encoding of the domains, and replace each m-ary branch by a $\log m$-depth binary decision tree. Thus, in practice only BDDs are used; ODDs can be understood as abbreviations of the respective binary encoded BDDs. For example, choosing the encoding $a \mapsto 00$, $b \mapsto 10$, $c \mapsto 01$, and $d \mapsto 11$, the BDD given in the right half of Figure 1 represents the same set as the respective ODD on its left.

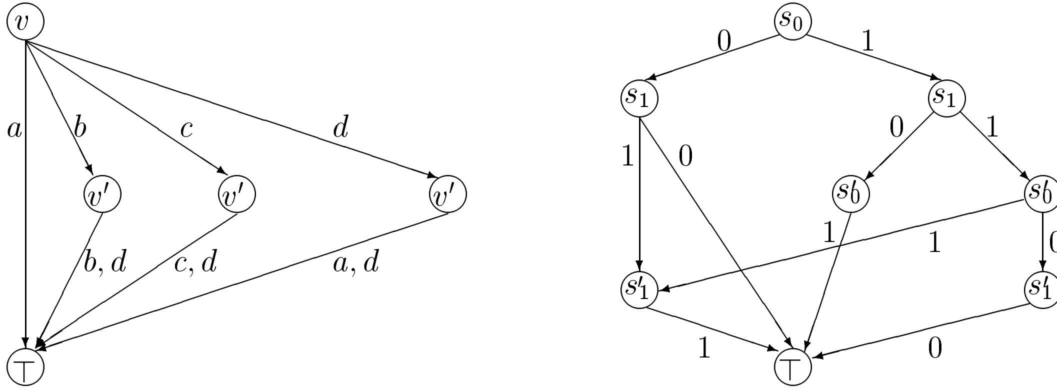


Figure 1: An ordered decision diagram and its binary encoding

The size of an ODD is the number of nodes it consists of. For a given ordering of the domains, and any set of values, there is a unique minimal ODD representing this set of values. The size of this minimal representation is not dependent on the size, but only on the structure of the represented set of values. E.g., the empty set and the set of all tuples both have an ODD representation of size one.

As another example, consider the elementship relation between a set $S = \{a, b, c\}$ and its powerset 2^S. The table and BDD are given in Figure 2. As can be seen, the table has no "regular" structure, thus both table and BDD are of order $S \cdot 2^S$. If we choose a different encoding as shown in Figure 3, the BDD representation exploits the fact that the matrix can be decomposed into isomorphic and constant submatrices.

Given a process P with state space D. Then the *transition relation* of P is a subset of $D \times D$. If P consists of k parallel processes $P_1, ..., P_k$ with state spaces $D_1, ..., D_k$, then the global state space of P is $D_1 \times \cdots \times D_k$. Therefore the transition relation can be described by $2k$ variables $s_1, ..., s_k, s_1', ..., s_k'$, where s_i and s_i' are over domain D_i and describe the current and next state of process P_i. Again, if each D_i has up to m states, the global transition relation has up to m^{2k} elements and can be described by a BDD over

$s_2s_3s_4$	000	001	010	011	100	101	110	111
s_0s_1	$\{\}$	$\{a\}$	$\{b\}$	$\{c\}$	$\{ab\}$	$\{a,c\}$	$\{b,c\}$	$\{a,b,c\}$
$00(a)$		x			x	x		x
$01(b)$			x		x		x	x
$10(c)$				x		x	x	x

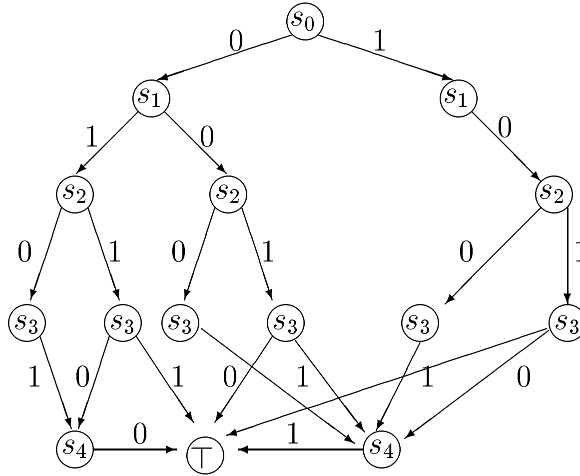


Figure 2: Power set relation and corresponding BDD

$s_2s_3s_4$	000	001	010	011	100	101	110	111
s_0s_1	$\{\}$	$\{a\}$	$\{b\}$	$\{a,b\}$	$\{c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,b,c\}$
$00(a)$		x		x		x		x
$01(b)$			x	x			x	x
$10(c)$					x	x	x	x

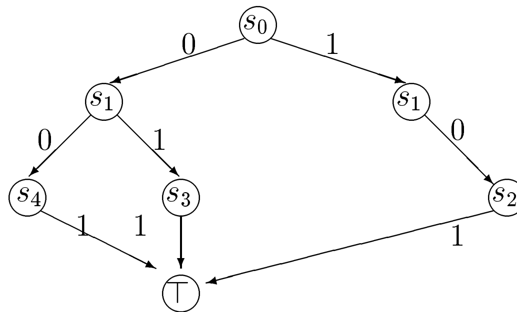
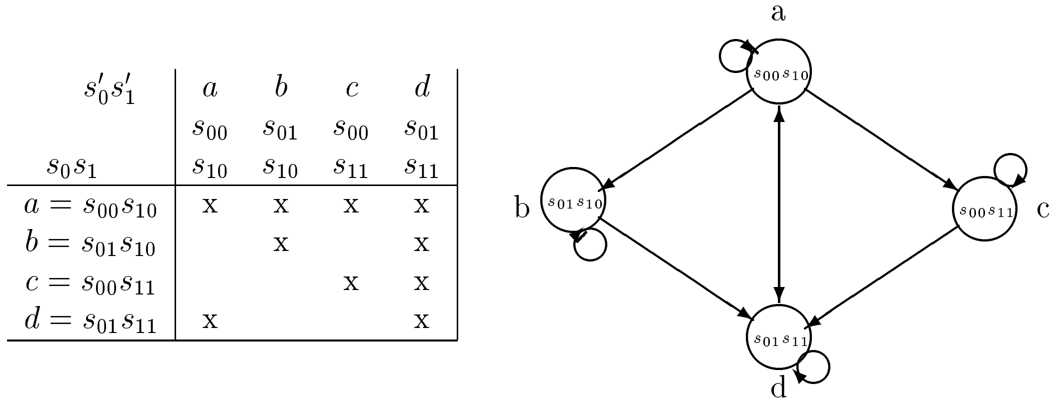


Figure 3: Power set relation and BDD with different encoding



$s_0's_1'$ / s_0s_1	a: $s_{00}s_{10}$	b: $s_{01}s_{10}$	c: $s_{00}s_{11}$	d: $s_{01}s_{11}$
$a = s_{00}s_{10}$	x	x	x	x
$b = s_{01}s_{10}$		x		x
$c = s_{00}s_{11}$			x	x
$d = s_{01}s_{11}$	x			x

Figure 4: Matrix and graph of the encoded relation

$2k \cdot \lceil \log |m| \rceil$ boolean variables. For example, consider the elementary net of Figure 5; it models two processes synchronizing on a common transition. The states of the first process are $D_0 = \{s_{00}, s_{01}\}$, the states of the second are $D_1 = \{s_{10}, s_{11}\}$. Since these domains are binary, we can use boolean variables s_0, s_1, s_0', s_1' to describe the current and next state of the processes. The global states are $a \triangleq (s_{00}, s_{10})$, $b \triangleq (s_{01}, s_{10})$, $c \triangleq (s_{00}, s_{11})$, and $d \triangleq (s_{01}, s_{11})$. In state d, either both processes idle or both processes synchronize and go to state a; in each other state, process P_i can either idle or make a step from s_{i0} to s_{i1}, independently of the other process. The transition relation of this system is the one represented by our example.

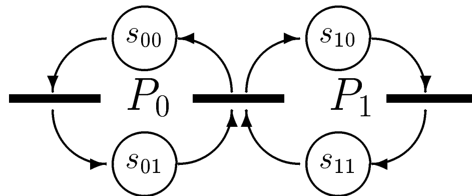


Figure 5: An elementary net model of synchronization

The set of *reachable states* of a system is the image set of the initial state(s) under the reflexive transitive closure of the transition relation. With BDDs, the transitive closure of a relation usually is calculated as the smallest fixed point of the recursive equation $R^* = \mathrm{I} \cup R;R^*$. Relational composition is calculated by the definition $xR;Sy$ iff $\exists z(xRz \wedge zRy)$, and existential quantification over finite (binary) domains is replaced by a disjunction of the possible values of the domain.

Therefore, to calculate the set of reachable states with BDDs it is necessary to represent the complete transition relation. Since BDDs are graphs with a nonlocal connection structure, usually it is not possible to use virtual storage for BDD nodes; present technology limits the number of BDD nodes representing a transition function to approx. 10^6. The size of the BDD representation of the reachable states or reflexive transitive closure of a relation is often totally unrelated to the size of the representation of the relation itself; in our example, the transitive closure is the universal relation, and thus all states are reachable, with a BDD representation of size 1.

However, the size of a BDD crucially depends on the number and ordering of variables. In our example, consider the two processes as producer and consumer of messages which are passed at the synchronization step via handshake. That is, each process has an additional variable, m_0 and m_1, which are both over a domain $\mathcal{M}$ of, e.g., 4 messages $\{nil, x_1, x_2, x_3\}$. Process P_0 produces a message, i.e. sets variable m_0 to an arbitrary non-*nil* value, in the transition from s_{00} to s_{01}. On transition from (s_{01}, s_{11}) to (s_{00}, s_{10}) the value of m_0 is transferred to m_1, and m_0 is reset to *nil*. Process P_1 consumes (resets) variable m_1 in the transition from s_{10} to s_{11}. On idling transitions, the value of the message-variables is stable. The SMV-code (for SMV, see [McM93]) for this system is given in Figure 6, and the resulting BDD for variable ordering $(s_0, s_0', s_1, s_1', m_0, m_0', m_1, m_1')$ is shown in Figure 7.

```
MODULE main
VAR s0 : boolean; s1 : boolean; m0 : {nil,x1,x2,x3}; m1 : {nil,x1,x2,x3};
INIT    (s0 = 0 & s1 = 0)
TRANS   (s0 = 0 & s1 = 1 -> next(s1) = 1)
&       (s0 = 1 & s1 = 0 -> next(s0) = 1)
&       (s0 = 1 & s1 = 1 -> next(s0) = 0 & next(s1) = 0 |
                              next(s0) = 1 & next(s1) = 1)
& (s0 = 0 & next(s0) = 1 -> next(m0) in {x1,x2,x3})         -- produce
& (s0 = 1 & next(s0) = 0 -> next(m0) = nil)                 -- reset
& (s0 = next(s0)         -> next(m0) = m0)                  -- stable
& (s1 = 1 & next(s1) = 0 -> next(m1) = m0)                  -- transfer
& (s1 = 0 & next(s1) = 1 -> next(m1) = nil)                 -- consume
& (s1 = next(s1)         -> next(m1) = m1)                  -- stable
```

Figure 6: SMV-code for message passing between two processes

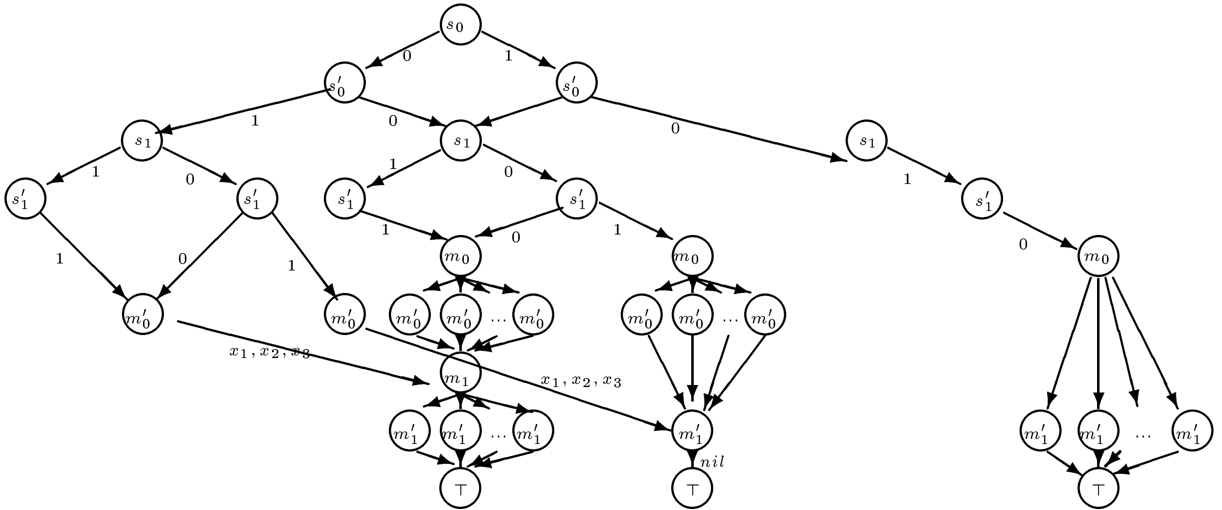


Figure 7: BDD for synchronous message passing

As can be seen, the size of this BDD is linear in the number $m \triangleq |\mathcal{M}|$ of possible messages. In this example, the linear complexity is caused only by "local diamonds", i.e.,

nodes branching into m successor nodes, which again join into one successor. This structure arises by the copying instructions `next(m0)=m0`, `next(m1)=m1` and `next(m1)=m0`. Variables m_0 and m_1 can be seen as consisting of w boolean variables $m_{01}...m_{0w}$ and $m_{11}...m_{1w}$, where $w \triangleq \lceil \log m \rceil$ is the *message width*. If we interleave the order of these variables, i.e., use variable ordering $(m_{01}, m'_{01}, m_{11}, m'_{11}, ..., m_{0w}, m'_{0w}, m_{1w}, m'_{1w})$, local diamonds are represented with complexity linear in w, see Figure 8. Thus, for the ordering $(s_0, s'_0, s_1, s'_1, m_{01}, m'_{01}, m_{11}, m'_{11}, ..., m_{0w}, m'_{0w}, m_{1w}, m'_{1w})$, the BDDs for the above SMV-code are logarithmic in m.

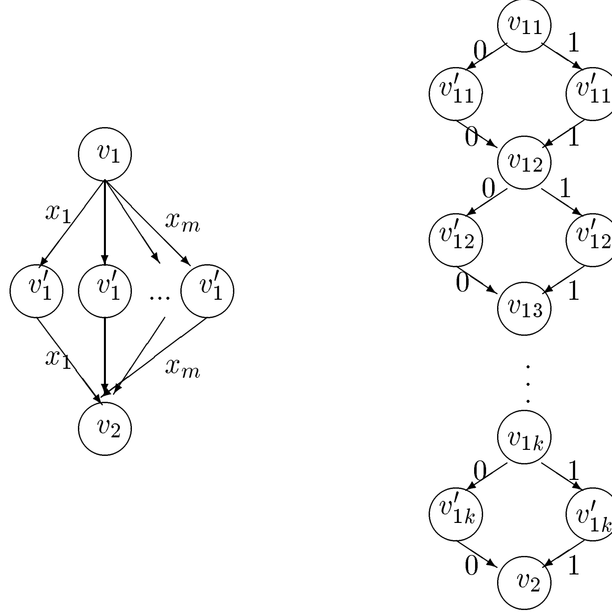


Figure 8: Interleaved encoding of a local diamond

3 Modelling of Message Buffers

Distributed parallel processes often use asynchronous (buffered) communication. Asynchronous message passing can be modelled with global variables by introducing a separate buffer process for each communication line. In many systems, the amount of messages which can be buffered is finite; in such systems buffer overflow often indicates erroneous behaviour of the system. For a fixed message alphabet $\mathcal{M} \triangleq \{nil, x_1, ..., x_{m-1}\}$, the formal specification of a bounded buffer of length n with input and output variables i and o over $\mathcal{M}$ is given in Table 3 on page 64.

In the right half of this table, an empty entry means that the respective variable is set by the environment. An input value of nil in i indicates that there is no message to be sent; in this case the next value of i is determined by the producer. If this process has put a non-nil value $x \in \mathcal{M}$ into i, then this value is appended to the buffer, and i is reset to nil. The last line indicates a condition of buffer overflow: If a message is to be sent with the message buffer already filled, i remains stable. If the output variable o is nil and there is a message to deliver, it is copied into o. The consumer receives a message y from o by resetting o to nil.

The content of the buffer b is given as a sequence $\langle x_1, ..., x_\nu \rangle$ of messages, where $\langle\rangle$ denotes the empty buffer. There are various possibilities to implement sequences of

i	b	o		i'	b'	o'
nil	$\langle\rangle$	nil			$\langle\rangle$	nil
x	$\langle\rangle$	nil		nil	$\langle\rangle$	x
nil	$\langle x_1, ..., x_\nu\rangle$	nil			$\langle x_1, ..., x_{\nu-1}\rangle$	x_ν
x	$\langle x_1, ..., x_\nu\rangle$	nil		nil	$\langle x, x_1, ..., x_{\nu-1}\rangle$	x_ν
nil	$\langle\rangle$	y			$\langle\rangle$	
x	$\langle\rangle$	y		nil	$\langle x\rangle$	
nil	$\langle x_1, ..., x_\nu\rangle$	y			$\langle x_1, ..., x_\nu\rangle$	
x	$\langle x_1, ..., x_\nu\rangle$	y	$(\nu < n)$	nil	$\langle x, x_1, ..., x_\nu\rangle$	
x	$\langle x_1, ..., x_n\rangle$	y		x	$\langle x_1, ..., x_n\rangle$	

Table 3: Specification of the transition relation of a bounded buffer

messages with BDDs. The most obvious choice is to use n variables $b_1, ..., b_n$ over $\mathcal{M}$, such that b_1 contains the front element of the message queue, and incoming messages are appended into the smallest b_ν which is empty (contains *nil* as value). The necessary assignment operation for this modelling is given in Figure 9.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : b[j+1];
  !(i=nil) & !(o=nil) : if !(b[j-1]=nil) & b[j]=nil then i
                        else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j]=nil then nil
                        else if b[j+1]=nil then i
                        else b[j+1] fi fi;
  esac;
```

Figure 9: Bottom-version of buffer slot assignment

In this modelling, we rely on the fact that whenever $b_j = nil$, then for all $k \geq j$, also $b_k = nil$. This assumption only holds for the reachable states of a buffer which is initially empty; there are many transitions from illegal, i.e., nonreachable states to other illegal states in this model. In an explicit representation of the transition relation, one should try to avoid these redundant entrys. With BDDs, however, even though the size of the transition relation is much bigger than the transition relation restricted to the reachable states, its representation is much smaller. Since the value of each buffer slot depends only on its immediate neighbours, in fact the size of the representation is linear in the number of slots.

In the above implementation, the buffer content is shifted upon output. We refer to this modelling as the *bottom* version, because sent messages can be imagined to "sink to the ground". A dual implementation of the buffer shifts down the content one slot whenever an input is performed, and inserts the new element into the topmost slot b_n. Consequently, we call this modelling, where messages "float to the surface", the *top-version* of a bounded buffer. To perform an output in this version, the content of the lowest non-*nil* slot is copied into the output variable o. The respective code segment is given in Figure 10.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : if (b[j-1]=nil) then nil else b[j];
  !(i=nil) & !(o=nil) : if (b[1]=nil) then b[j+1] else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j]=nil then nil else b[j+1] fi;
  esac;
```

Figure 10: Top-version of buffer slot assignment

A third possibility is to use a *circular* implementation of the buffer: On input, the value of the input variable is copied into slot b_i, where $b_i = nil$ and $b_{i-1} \neq nil$; on output, o is set to b_j, where $b_j \neq nil$ and $b_{j-1} = nil$. To be able to distinguish between first and last element of the queue in this version, we have to make sure that there is at least one slot with content *nil*; therefore there has to be one more place than the actual capacity of the buffer. In the assignment clause in Figure 11, subtraction and addition of one is to be understood modulo n.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : if b[j-1]=nil then nil else b[j];
  !(i=nil) & !(o=nil) : if !(b[j-1]=nil) & b[j]=nil & b[j+1]=nil
                        then i else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j-1]=nil then nil
                        else if b[j]=nil then i else b[j] fi;
  esac;
```

Figure 11: Circular version of buffer slot assignment

An alternative to the use of an empty slot would be to introduce queue-pointers for the position of the first and/or last element of the queue; this idea can be applied to all three of the above modellings. However, these alternative versions turn out to be worse than the direct encoding via *nil*-test which is given above. In general, the queue-pointers would be functionally dependent of the content of the buffer; such functional dependencies can blow up the BDD size significantly ([HD93]).

Similarly, we can introduce additional BDD-variables indicating whether the buffer is empty or full; however, these variables tend to increase the size of the representation by a linear factor and usually can be replaced by appropriate boolean macro definitions. On the other hand, such variables can be important if the BDD is represented as a conjunction of partitioned transition relations, see [BCL91].

Finally, it is not always advisable to test whether a slot b_i contains the value *nil* by the test `b[i]=nil`. As we will see in the next section, it can be better to increase the message width w by one, such that the first bit of each message is a kind of checksum, indicating whether this message is *nil* or not.

4 Complexity Considerations

The BDD for the bottom version of a buffer of size n consists of two parts, one for the case that the buffer content remains stable, and one for the case that the buffer content is shifted down by one slot. The first part consists of a sequence of local diamonds for each slot, similar as in the example above. The BDD for the second part is depicted in Figure 12 for the special case $n = 2$ and $\mathcal{M} = \{x_1, x_2, x_3\}$.

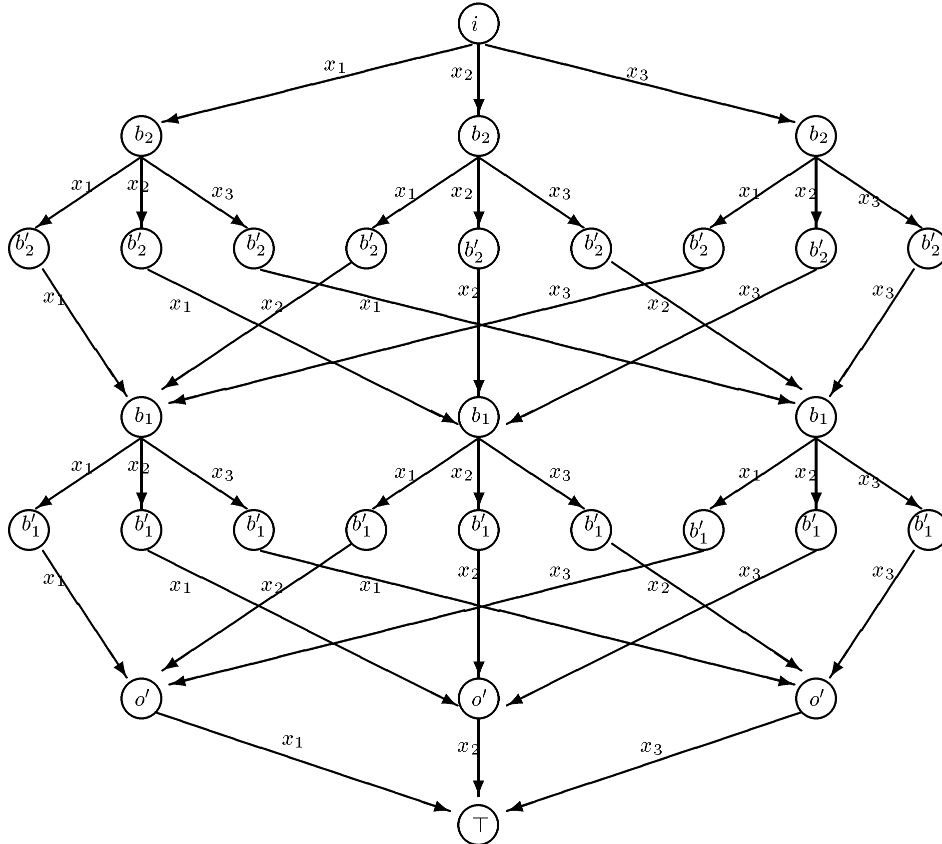


Figure 12: BDD for shifting down the buffer content

As can be seen, for a new buffer slot b_{n+1}, $O(m^2)$ nodes are added to the BDD for a buffer of length n. Therefore the representation is of order $n \cdot m^2$, i.e., linear in the length and exponential in the width of the buffer. Since the transition relation is "almost" a function, a matrix representation would require $O(m^n)$ entrys, whereas a boolean algebra or programming language representation such as the SMV code above, is of order $m + n$ (or even constant, if array subscripts are allowed).

For the top version, the complexity of the representation is comparable to the bottom version. In the circular version, b'_n depends on b_n, i, b_{n-1}, and on b_1. This non-local dependency causes a blowup of factor 2, since the emptyness of b_1 has to be decided while testing b'_n. Moreover, to test whether a buffer is full or not we have to test whether any two adjacent slots are nil. This nonlocal test again blows up the complexity of this modelling.

As was to be expected, the number of reachable states is identical in the bottom and top modellings; of course, this number is exponential in the length of the buffer. For the circular implementation, the number of reachable states is approximately m times as much, since it contains an additional slot.

Table 4 summarizes the size of the BDDs of the transition relation for $m = 4$ (i.e., $w = 2$), and order $i < b_n < ... < b_1 < o$. All results were obtained with the public-domain SMV system; other BDD-based verification tools yield similar results. The buffers were embedded in a simple producer-consumer environment, where the producer and consumer are asynchronous, and the message to be sent or received does not depend upon or influence the state of the sender or receiver, respectively.

length	3	5	7	9	11
bottom	714	1458	2204	2950	3696
top	599	1113	1627	2141	2655
circular	1038	2350	3999	5307	6833
reach	1400	12740	2^{16}	2^{20}	2^{23}

Table 4: BDD size of transition relation and reachable state set

In this example, the size of the representation of the set of reachable states was of the same order of magnitude as the representation of the transition relation. Some considerations about this size are given below.

A critical factor in our approach is the message width w. As indicated in Table 4, e.g. the bottom implementation of a buffer of length 5 and width 2 has size 1458. For $w = 3$, this size is 11774, and for $w = 4$, it is 108357. In [BS90] it is proved that for any finite function, a BDD of polynomial size exists iff the function can be realized by a polynomially bounded depth circuit. For message buffer, certainly the transition function can be realized by such a circuit; thus there exists a BDD which is polynomial both in n and w.

If there is no constraint on the order of variables, then such a BDD can be constructed by interleaving the bits of all slots: Let $i = i_1...i_w$, $b_j = b_{j1}...b_{jw}$, and $o = o_1...o_w$. Then for each $k \leq w$, $(i_k, b_{nk}, ..., b_{1k}, o_k)$ can be regarded as a buffer of width 1. The only "nonlocal test" in this buffer of length 1 is whether some slot b_j is empty: if this is determined by comparison of b_{j1} and ... and b_{jn}, then we still have an exponential growth. If we introduce additional bits $(i_0, b_{n0}, ..., b_{10}, o_0)$ which are 0 iff the corresponding message is *nil*, then each bit-slice is linear in the length of the buffer. For the order $i_0 < b_{n0} < ... < b_{10} < o_0 < i_1 < b_{n1} < ... < b_{11} < o_1 < ... < i_w < b_{nw} < ... < b_{1w} < o_w$, these small BDDs are simply added, and the overall complexity is $O(w \cdot n)$.

Unfortunately, in many practical examples it is not possible to choose such a bitwise interleaved order. Usually, the input and output variables are imported from other processes, and their order cannot be chosen arbitrarily. An argument similar to the one from Section 1 on page 63 shows that for any order, in which i is before all buffer bits, the representation is exponential in w. Therefore, in practical verification, w should be kept as small as possible. There are several ways to do so:

- For every channel, define a separate message alphabet;
- replace a parametrized message $m(t)$ with $t \in \{t_1, ..., t_k\}$ by a list of messages m_{t1}, ..., m_{tk};
- replace a compound message by a sequence of messages, and
- abstract several different messages into one.

When using the latter two methods, one has to be careful to preserve the semantics of the original model ([CGL92]). Using these techniques, we have been able to verify systems with up to 2^7 different messages.

5 Alternative Approaches

In [GL96],[BG96] it is suggested to extend the BDD data structure for the representation of message buffers. The new data structures are called QBDDs and QDDs, respectively. The basic idea is to replace the consecutive testing of buffer variables by a repeated test of one and only one variable. Therefore, the representation of the transition relation is independent of the buffer size. Moreover, even systems of which the maximum amount of buffer space is not statically known can be verified.

However, as we have shown above, the (static) length of a buffer may not be the most important factor in the representation of the transition relation. Moreover, "buffer overflow" errors in the system can only be detected with a bounded buffer. Even worse, in systems on which a full buffer forces delay of the sender, with QBDDs we have to introduce an additional counter variable. For these type of systems, BDDs seem to be more adequate than QBDDs or QDDs.

Being able to represent the transition relation is only a necessary prerequisite for the verification of a system. Equally important is the size of the representation of the *reachable states* $\mathcal{R}$ of the system. Unfortunately, the size of the BDD for $\mathcal{R}$ has no predictable connection to the size of the BDD for the transition relation.

In many systems both the number of reachable states and its representation are linear in the number of iteration steps of the model, iff the system is correct. This is due to the fact that on reachable states, the transition relation is "almost" functional, yielding either a single or a small number of successor states. On the other hand, from an "impossible" state usually many other "impossible" states are reachable. A drastic example is Valmari's elevator for which the reachable state set (in any representation) explodes as the elevator breaks through the ceiling and skyrockets into the air. Thus an exponential increase in (the representation of) $\mathcal{R}$ after some number of steps almost certainly indicates an error.

In [GL96] it is claimed that "there are cases where the QBDD representation is strictly more concise than the BDD representation". Assume our buffer in a context where the producer sends one fixed sequence of messages x_1, x_2, ... x_ν. That is, the reachable buffer content is $\{\langle\rangle, \langle x_1\rangle, \langle x_2 x_1\rangle, ..., \langle x_\nu ... x_2 x_1\rangle, \langle x_2\rangle, ... \langle x_\nu ... x_2\rangle, ..., \langle x_\nu\rangle\}$. With the top- and bottom version of the buffer, the representation of this set is quadratic in ν, whereas with the circular representation and also with QBDDs it is linear in ν.

On the other hand, consider the case that the producer can send an arbitrary sequence of messages. In this case, the top- and bottom-versions are of constant size, whereas the QBDD implementation is linear in the number of sent messages.

In practical examples, such extreme cases are rare. In our experiments, we have found no significant difference in the size of the reachability sets of the various alternatives. The number of parallel processes and their relative order has a much bigger impact on the size of the BDD for $\mathcal{R}$ than the actual implementation of the buffer. Typically we can handle systems of up to 5 processes, each with approx. $2^4 - 2^5$ local states, where each process is equipped with a buffer of $n, w \leq 5$. However, there still is a need for heuristics which use dependencies between the processes to obtain a "good" order for the process state variables.

An important observation is that the content of a message buffer used to coordinate

processes shows regular patterns, which also depend on the state of the processes. E.g., in a certain process state the buffer might always contain only copies of two different messages from $\mathcal{M}$. As another example, some specific message might always be followed by some other specific message in the buffer. Currently we are investigating methods how these regularities can be exploited to further reduce the size of the representation of the reachability set.

References

[BCDM91] J. Burch, E. M. Clarke, D. Dill, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In 5th *IEEE LICS*, June 1991.

[BCL91] J. Burch, E. M. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *Proc. IFIP Conf. on VLSI*, Edinburgh, August 1991.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of* 5th *CAV*, New Brunswick, July 1996.

[Bry92] R. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Comp. Surv.*, Vol 24, No 3:293–318, 1992.

[BS90] R. Boppana and M. Sipser. The complexity of finite functions. In J van Leeuwen, editor, *Handbook of theoretical computer science, Vol. A*, chapter 14, pages 757–805. Elsevier, Amsterdam, 1990.

[CGL92] E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In 19th *ACM POPL*, January 1992.

[GL96] P. Godefroid and D. Long. Symbolic protocol verification with queue BDDs. In *Proceedings of IEEE LICS*, New Brunswick, July 1996.

[HD93] A. Hu and D. Dill. Reducing BDD size by exploiting functional dependencies. In *Proc.* 30th *ACM/IEEE DAC*, 1993.

[McM93] K. McMillan. *Symbolic model checking.* Kluwer, 1993.

Visiting Some Relatives of Peirce's *

Michael Böttner
Max-Planck-Institut für Psycholinguistik, Nijmegen, The Netherlands

Abstract

The notion of a relational grammar is extended to ternary relations and illustrated by a fragment of English. Some of Peirce's terms for ternary relations are shown to be incorrect and corrected.

Binary relations have been studied extensively by Peirce and Schröder in the nineteenth century, and in this century by Tarski and his students. No comparable attention has been paid to ternary relations. This is surprising, for Peirce had already dealt with ternary relations on various occasions. But Schröder strictly confined himself to binary relations, and it is the topic of binary relations that has become the focus of interest for Tarski and his students. For notable exceptions see Carnap (1929), Copi and Harary (1953), and Aubert (1955) and the theory of relational database systems.

Peirce has illustrated his ternary relational terms by natural language examples. For instance, the equation

$$(b^a)^m = b^{(am)}$$

where b = *betray*, a = *enemy*, and m = *man* was explained in English as follows:

> "those individuals each of which stand to every man in the relation of betrayer to every enemy of his are identical with those individuals each of which is a betrayer to every enemy of a man of that man."[1]

This may be hard to swallow and even Peirce himself had some problems here as we shall see. Therefore I think that Peirce's discussion of ternary relations can best be studied in a framework that is as rigorous with respect to syntactic structure as it is to semantic structure. Such a method is provided by relational grammar. Relational grammar was proposed in Suppes (1976).

The purpose of this paper is therefore both to extend the notion of a relational grammar by adding ternary relations and to apply this extension to a better understanding of Peirce's writings about relations, or "relatives" to use his own term. Our focus is therefore more on the establishment of mapping natural language into the language of relation algebra than on the development of the algebra of ternary relations. The paper continues my work on relational grammar and builds especially on previous results on anaphoric pronouns in Böttner (1992, 1994, 1997).

The paper is organized as follows: in section 1, the notion of a relational grammar is introduced. In section 2, this notion is extended in order to account for ternary relations. In section 3, an example of a ternary relational grammar is given. In section 4, our analyses are compared to Peirce's examples. In section 5, our results are discussed and put in perspective.

---

* Previous versions of this paper have been presented at the RelMiCS III workshop in Hammamet (Tunisia) and at the university of Osnabrück. I would like to thank Melissa Bowerman (Nijmegen), Chris Brink (Cape Town), Barry Dwyer (Adelaide), Arnold Günther (Berlin), Peter Jipsen (Cape Town), Siegfried Kanngießer (Osnabrück), Roger Maddux (Nashville, Tennessee), and Bill Purdy (Syracuse, NY) for many useful comments on a preliminary version of this paper.

[1] Peirce (1870: 379).

1 Relational Grammar

A *denoting grammar* is a context-free phrase structure grammar that provides a semantic function for each production rule.[2] A *relational grammar* is a denoting grammar with the restriction that denotations are elements of an extended relation algebra over some set D.[3] An *extended relation algebra* over D is any collection of subsets of and binary relations over D that is closed with respect to *union*, *complement*, *conversion*, *composition* and *Peirce product*.[4] In line with Brink, Kahl and Schmidt (1997) we use the following notation:

- *union*: $A \cup B$
- *complement*: $\overline{A}$
- *intersection*: $A \cap B$
- *Cartesian product*: $A \times B$
- *conversion*: $\breve{R}$
- *composition*: $R; S$
- *Peirce product*: $R : A$

In addition, we shall use the following operations that can be defined in terms of the previous operations:

- *domain*: $domR = \{x \in D | (\exists y)(xRy)\}$
- *(progressive) involution*: $R^A = \overline{(\overline{R} : A)}$
- *range-restriction of R by A*: $R \upharpoonright A = R \cap (D \times A)$

We refer to the identity relation by I. We refer to the maximal subset of D by the constant U and to the maximal binary relation over D by the constant V.

An example of a relational grammar is the following:

PRODUCTION RULE		*SEMANTIC FUNCTION*	
$NP \rightarrow$	$TN + P + EQ + NP$	$[NP] =$	$[TN] : [NP]$
$NP \rightarrow$	$TN + P + UQ + NP$	$[NP] =$	$[TN]^{[NP]}$
$NP \rightarrow$	N	$[NP] =$	$[N]$

The symbols abbreviate the names of conventional grammatical categories: NP = noun phrase, N = common noun, TN = transitive noun, P = preposition, UQ = universal quantifier, EQ = existential quantifier. A lexicon for this grammar would be as follows:

P	*of, to*
EQ	*some, a, an*
N	*flower, lady, horse, ...*
UQ	*each, every*
TN	*owner, enemy, lover, woman, ...*

[2]Suppes (1973).

[3]Suppes (1976).

[4]Suppes (1976).

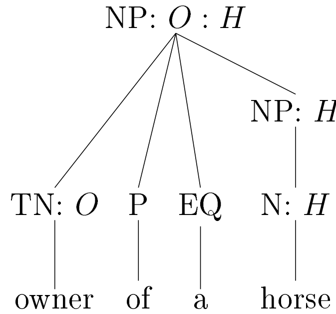


Figure 13.

This grammar derives semantic trees for terms like, e.g., *owner of a horse* or *owner of every horse*. A semantic tree is a derivation tree in the sense of the theory of formal languages where the nodes of the tree, in addition to their category labels, bear denotations as semantic labels. An example of a semantic tree is given in Figure 1.

2 Ternary Relations

Relational grammar is restricted to subsets of some domain D and binary relations over D. This would not be sufficient to provide meanings for sentences like, e.g., *Mary is sitting between John and Bill* or *John gives Mary a book* since *between* denotes a ternary relation and so does *give*. We therefore shall extend our ontology by ternary relations.

One way to introduce ternary relations is to introduce them as Cartesian products of a binary relation over D and a subset of D. This definition, however, has the following drawback. One and the same ternary relation gets two representations that need to be identified by stating separately

$$< a, < b, c >>=<< a, b >, c > .$$

We therefore prefer to start from ordered triples and define a ternary relation as a set of ordered triples.

Relational operations have been defined for binary relations. Adding ternary relations requires a slight redefinition of our relational operations. In the case of union and intersection it is understood that both operands should be of the same type, i.e. either D, D^2 or D^3. In the case of complement of X we understand the complement with respect to either D, D^2, or D^3 depending on the type of X.

We assume two operations R^t and R^c as primitive: R^t switches the last two places of a ternary relation and R^c moves the first place of a ternary relation to the end. The operation that reverses a ternary relation R can be expressed by the composition of a transposition and a cyclic permutation: R^{tc}.

Since binary relations are sets, the operation of a *Peirce product* can be generalized to ternary relations provided that $m < n$. Let R denote an n-ary relation on D and let S denote an m-ary relation. Then the *generalized Peirce product* of R and S is defined

$$R : S := \{< x_1, ..., x_{n-m} > |(\exists x_{n-m+1})...(\exists x_n)(Sx_{n-m+} \wedge Rx_1...x_n)\}. \tag{31}$$

This definition looks rather complicated but in fact captures only three cases: either R is binary and S is unary, or R is ternary and S is unary, or R is ternary and S is binary. If in particular R is a ternary relation over D and $S \subseteq D$, then $R : S$ is a binary relation

over D, and if R is a ternary relation over D and S is a binary relation over D, then $R : S \subseteq D$.

In a similar fashion, the operation of *range-restriction* is generalized

$$R \restriction S := \{< x_1, ..., x_n >\in R | < x_{n-m}, ..., x_n >\in S\} \tag{32}$$

where R is an n-ary relation and S is an m-ary relation. If R is a binary relation and S some subset of the domain the operation coincides with the operation defined in section 1. If R is a ternary relation and S is a subset of the domain $R \restriction S$ denotes a binary relation over D. If R is a ternary relation and S is a binary relation over the domain $R \restriction S$ denotes a subset of D.

Composition is defined as an operation on the set of binary relations. We extend this operation to pairs of a ternary relation R and a binary relation S like this:

$$R \circ_{3,2} S = \{< x, y, z > | (\exists u)(Rxyu \wedge uSz)\}. \tag{33}$$

Therefore

$$(R \circ_{3,2} S)xyz \text{ iff } (\exists u)(Rxyu \wedge uSy\}.$$

Since *dom* can be defined in terms of Peirce product, it shares this ambiguity with it: if R is a binary relation, then $domR$ is a set, and if R is a ternary relation, then $domR$ is a binary relation.

Since involution can be defined in terms of Peirce product and complement, a notion of *generalized involution* can be defined

$$R^S := \overline{(\overline{R} : S)}. \tag{34}$$

Many more operations can of course be defined in the context of ternary relations. But since our main focus is on the interaction of ternary relations with either binary relations or sets, so-called exterior operations will be more important than interior operations involving just the set of ternary relations. We have therefore refrained from defining various types of composition since we have not found them exemplified in any construction of English.

Definition 1 *Let D be some nonempty set. An extended ternary relation algebra of sets over D is any subset of*

$$\mathcal{P}(D) \cup \mathcal{P}(D^2) \cup \mathcal{P}(D^3)$$

that is closed with respect to

i. *union*

ii. *complement*

iii. *transposition*

iv. *cycle*

v. *composition*

vi. *composition of a a ternary relation with a binary relation*

vii. *generalized Peirce product*

viii. *generalized domain-restriction.*

Notice that this list of operations appears to lack conversion. But in fact it occurs twice because both transposition and cycle coincide with conversion in the case of binary relations. Notice that we do not have composition of two ternary relations because this would return a quaternary relation. This does not mean that quaternary relations of this kind do not arise in natural language. Peirce himself has given the example *praiser of – to a maligner of – to –*.[5]

Some simple arithmetical properties of operations of ternary relations are listed below.

Theorem 1 *Let $R \subseteq D^3$.*

i. $R^{tt} = R$.

ii. $R^{ccc} = R$.

iii. If X, Y are either both subsets of D or both binary relations over D, then

$$R : (X \cup Y) = (R : X) \cup (R : Y).$$

iv. If A and B are arbitrary subsets of D, then

$$(R^t : A) : B = (R : B) : A.$$

v. If A and B are subsets of D, then

$$((R^t)^B) : A \subset (R : A)^B.$$

Proof of Theorem 1.

i. This is simply an extension of the binary case.

ii. This follows from the fact that a cyclic transposition of a set with thre elements needs to be applied three times to return the original set.

iii. Obvious.

iv. The left hand side is equivalent to

$$(\exists z)(z \in B \wedge (\exists y)(y \in A \wedge R^t xzy)).$$

The right hand side is equivalent to

$$(\exists y)(y \in A \wedge (\exists z)(z \in B \wedge Rxyz)).$$

Since

$$R^t xzy \leftrightarrow Rxyz,$$

both expressions are equivalent.

v. By (34), the equation can be reduced to

$$\overline{\overline{R^t} : B} : A \subset \overline{\overline{R : A} : B}.$$

The left hand side is equivalent to

$$(\exists z)(z \in A \wedge (\forall y)(y \in B \rightarrow Rxyz)).$$

The right hand side is equivalent to

$$(\forall y)(y \in B \rightarrow (\exists z)(z \in A \wedge Rxyz)).$$

Since the second follows from the first the theorem is proved. Note that this property cannot be strengthened to equality, since both expressions are not equivalent.

[5] Peirce (1902).

PRODUCTION RULE		SEMANTIC FUNCTION	
$VP \to$	$TVP + EQ + NP$	$[VP] =$	$[TVP] : [NP]$
$VP \to$	$TVP + UQ + NP$	$[VP] =$	$[TVP]^{[NP]}$
$TVP \to$	TV	$[TVP] =$	$[TV]$
$TVP \to$	$DV + EQ + NP + P$	$[TVP] =$	$[DV] : [NP]$
$TVP \to$	$DV + UQ + NP + P$	$[TVP] =$	$[DV]^{[NP]}$
$VP \to$	$DV + EQ + NP + P + EQ$ $+TN + P + Dem + NP$	$[VP] =$	$dom(([DV]; [TN]) \cap (D^3 \restriction I)) : [NP]$
$VP \to$	$DV + EQ + NP + P + EQ$ $+TN + P + Pers$	$[VP] =$	$dom(([DV]; [TN]) \cap (D^3 \restriction I)) : [NP]$
$VP \to$	$DV + EQ + NP + P + UQ$ $+TN + P + Pers$	$[VP] =$	$dom(([DV]; [TN]) \cap (D^3 \restriction I)) : [NP]$
$VP \to$	$DV + EQ + NP + UQ + NP'$	$[VP] =$	$([DV] : [NP'])^{[NP]}$
$VP \to$	$DV + UQ + NP + EQ + NP'$	$[VP] =$	$([DV]^{[NP']}) : [NP]$
$VP \to$	$DV + EQ + NP + EQ + NP'$	$[VP] =$	$([DV] : [NP']) : [NP]$
$VP \to$	$DV + UQ + NP + UQ + NP'$	$[VP] =$	$([DV]^{[NP']})^{[NP]}$

Table 5. Ternary Extension of Relational Grammar

3 Grammar Extension

To derive semantic trees for English expressions we propose the grammar of Table 5. Familiar grammatical categories are referred to by the following additional symbols: TVP = transitive verb phrase, DV = ditransitive verb, Dem = demonstrative pronoun, and Pers = personal pronoun.

A lexicon for the extended grammar would be as follows:

P	*to*
Dem	*that*
DV	*give, betray, ...*
Pers	*him, her, it, them*

Ditransitive verbs differ from monotransitive verbs like, e.g., *own* by having two objects rather than one. One object is called the *direct object* (DO), the other object is called the *indirect object* (IO). A paradigm ditransitive verb is *give.* In the verb phrase *gives a flower to some lady* the direct object is *a flower* and the indirect object is *(to) a lady.* According to our extended grammar, the semantic tree for this verb phrase would be the one shown in Figure 14 where F and L are subsets of D denoted by the noun *flower* and *lady*, respectively, and $G = \{< x, y, z >\}$ is a ternary relation on D denoted by the ditransitive verb *give* where x denotes the giver, y denotes the receiver, and z denotes the object given.

For the expression *betray a woman to a man* our grammar derives the denotation

$$(B : W) : M$$

where M is the subset of D denoted by *man*, W is the subset of D denoted by *woman* and B is a ternary relation on D denoted by *betray.* That this denotation provides the correct denotation follows from the fact that it is equivalent to the set

$$\{x|(\exists y)(y \in M \land (\exists z)(z \in W \land Bxyz)\}. \quad (35)$$

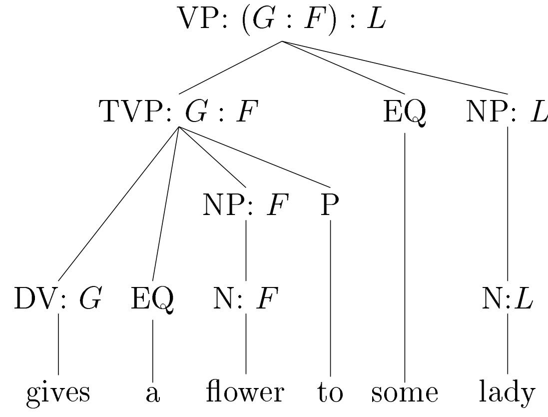


Figure 14.

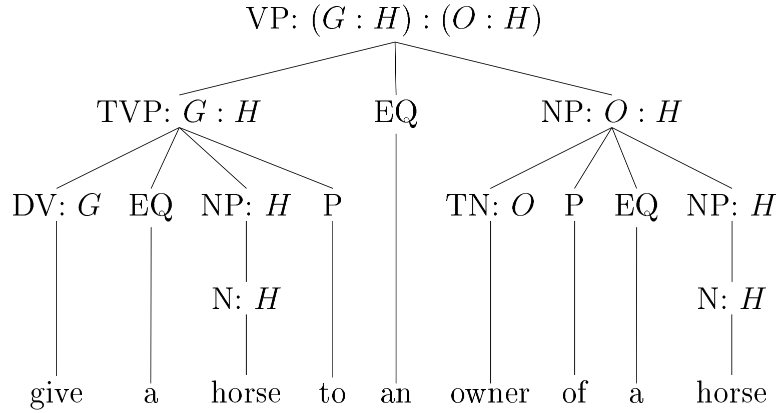


Figure 15.

For the expression *betray every woman to every man* our grammar derives the denotation

$$(B^W)^M. \tag{36}$$

Applying our definition, we have

$$(B^W)^M = \overline{(\overline{B^W} : M)} = \overline{(\overline{\overline{(\overline{B} : W)}} : M)} = \overline{(\overline{B} : W) : M}. \tag{37}$$

An element z of this set fulfils the condition

$$(\forall z)(z \in M \to (\forall y)(y \in W \to Bxyz)), \tag{38}$$

and this captures the intuitive meaning of the verb phrase in question.

Our grammar also derives semantic trees for expressions with binary relations occurring in argument position. An example would be the tree in Figure 15 where H is a set denoted by *horse*, O is a binary relation denoted by *owner* and G is a ternary relation as in (14). For the expression

$$\textit{betray each man to an enemy of every man} \tag{39}$$

our grammar derives the term

$$(B^M) : (A^M) \tag{40}$$

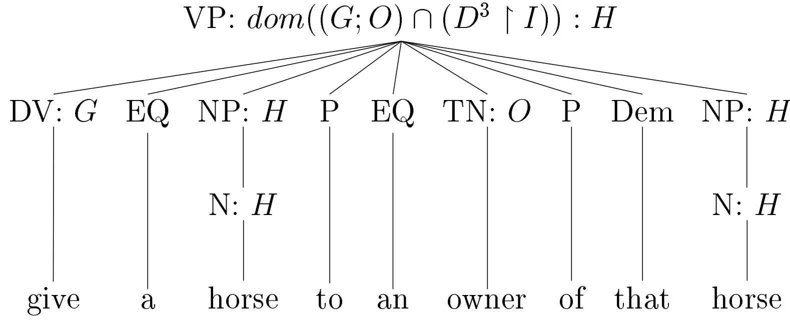


Figure 16.

where B is the ternary relation denoted by the ditransitive verb *betray*, A is a binary relation denoted by the transitive noun *enemy*, and M is the set denoted by the common noun *man*. By definition, this term is equivalent to

$$\overline{(\overline{B} : M)} : \overline{(\overline{A} : M)}. \tag{41}$$

By simple computation this expression will turn out to be equivalent to

$$\{x|(\exists y)(\forall z)(z \in M \to yAz \wedge Bxyz)\}, \tag{42}$$

which is an appropriate translation of (39).

Our grammar also derives semantic trees for expressions with anaphoric pronouns in Figure 16. Notice that the Peirce product and conversion operations are not sufficient here and some additional operation is required. In a similar fashion, a semantic tree can be derived for the expression *betray a man to an enemy of him.* The root denotation of this tree is

$$dom((B; A) \cap (D^3 \restriction I)) : M. \tag{43}$$

Any element x of this set fulfils the condition

$$(\exists y)(y \in M \wedge (\exists z)(Bxyz \wedge zAy)) \tag{44}$$

which captures the intuitive meaning of the verb phrase. Correspondingly, the expression

$$\textit{betray a man to every enemy of him} \tag{45}$$

would by our grammar be assigned the denotation

$$dom(\overline{(\overline{B}; A) \cap (D^3 \restriction I)}) : M. \tag{46}$$

This term is equivalent to the set

$$\{x|(\exists y)(y \in M \wedge (\forall z)(zAy \to Bxyz))\}, \tag{47}$$

which is in line with our intuition about the meaning of (45).

In Table 1 a grammar was given for a fragment of English that is large enough to derive many of Peirce's English examples to illustrate his operations and their use to construct complex terms. In the next section we use our fragment to check Peirce's constructions.

4 Peirce's Relatives

Our grammar is able to derive semantic trees for most of the terms with ternary relations considered in Peirce (1870). A term expressing a ternary relation is called a "conjugate term" by Peirce. Peirce illustrates his relational terms by examples from everyday English. Peirce made occasional blunders in his notation as had been pointed out before.[6]

In order to be able to correctly assess the terms proposed by Peirce we need to explain some of his notation. Peirce uses juxtaposition and exponentiation in case the first term is a relation and the second term is relational or absolute. So xy may correspond to standard relational composition in case both x and y are binary relations, or to the Peirce product in case y is an absolute term and x is a binary relation. Similarly, x^y may correspond to standard involution if x is a binary relation and y is an absolute term, or to generalized involution in case x is a ternary relation and y is a binary relation or absolute.

Relations as Arguments Peirce also considers the case of a binary relation occurring in argument position like, e.g., in

$$\textit{giver of a horse to an owner of a horse.} \tag{48}$$

In Peirce's notation, this corresponds to the term $gohh$. This is equivalent to the root denotation of the tree of Figure 15. But in the case of

$$\textit{betrayer of each man to an enemy of every man} \tag{49}$$

Peirce appears to have got it wrong. The term he proposed is ba^m. On our account, the denotation would be (40). Notice that the respective denotations are not equivalent.

This term can be analyzed either by $(ba)^m$ or by $b(a^m)$. Recall that by juxtaposition of two terms x and y, Peirce denotes either relational composition[7] or Peirce product.[8] Assume juxtaposition denotes composition. Then ba denotes a ternary relation and $(ba)^m$ denotes a binary relation. This cannot be correct, since (4) is an absolute term and should denote a set. Assume therefore that juxtaposition denotes the Peirce product. Then $b(a^m)$ denotes a binary relation too. Consider now the possibility that ba denotes the Peirce product. Then ba denotes a set. And if ba denotes a set then $(ba)^m$ is not defined. Notice that a^m always denotes a set. But if a^m denotes a set, b a ternary relation and $b(a^m)$ denotes the Peirce product of b and a^m, then $b(a^m)$ denotes a binary relation. But this is not correct, since $b(a^m)$ is supposed to denote a set. Similar remarks hold for other terms with three quantifiers proposed by Peirce.

Anaphora Some of Peirce's terms involve anaphoric pronouns. For instance, for the expression *betrayer of a man to every enemy of him*, the term $b^a m$ is proposed by Peirce.[9] This is not correct. For $b^a m$ is equivalent to

$$\{x|(\exists y)(y \in M \wedge (\forall z)(xAz \rightarrow Bxzy)\}, \tag{50}$$

and (50) is not equivalent to (47).

[6] Cf. Brink (1978) and Martin (1978).
[7] cf. Brink (1978: 288).
[8] cf. Martin (1978: 27).
[9] Peirce (1870: 378 and 426).

Peirce proposed the term *goh* as a denotation for[10]

$$\textit{giver of a horse to the owner of that horse.} \tag{51}$$

Martin pointed out correctly that this is wrong but did not give a correct term for (51).[11] Recall that our grammar derives a semantic tree in Figure 16 for a structure that is closely related. If we assume the denotation for *own* to be a left-unique binary relation, then the tree in Figure 16 would also be a semantic tree for (51).

Scope Peirce sharply distinguishes two notions of *give*:[12]

g_1: *giver of —- to —-*
g_2: *giver to —- of —-*

This distinction corresponds to a difference in syntactic structures with g_1 occurring in a structure with the direct object preceding the indirect object like in, e.g.

$$\textit{give a flower to every lady,} \tag{52}$$

and g_2 occurring in a structure with the indirect object preceding the direct object like in, e.g.,

$$\textit{give every lady a flower.} \tag{53}$$

More important than the relative order of the direct and indirect objects is the scope of direct and indirect objects. In principle, two situations can be distinguished: either the direct object is in the scope of the indirect object as is the case in (52) or the indirect object is in the scope of the direct object. The first situation is called the patient analysis. The second situation is called the recipient analysis. It is often claimed that (53) has the same meaning as (52).[13]

According to Peirce the meaning of bm^w would be *betrayer of all women to a man.*[14] Notice that bm^w is equivalent to

$$\{x|(\forall y)(y \in W \rightarrow (\exists z)(z \in M \wedge Bxyz))\}. \tag{54}$$

On this analysis, the indirect object *man* falls inside the scope of the direct object *women*, which runs against common linguistic intuition. But the denotation of the phrase *betrayer of all women to a man* should rather be

$$\{x|(\exists z)(z \in M \wedge (\forall y)(y \in W \rightarrow Bxyz))\}. \tag{55}$$

Our grammar accounts for this fact by introducing the order DV DO IO in two steps, but introducing the order DV IO DO in one step and assigning

$$(B^W) : M \tag{56}$$

as a denotation for the phrase *betrayer to a man of all women.* This denotation is identical to the one provided for the phrase *betrayer of all women to a man.*

[10] Peirce (1870: 370).
[11] Martin (1978: 29).
[12] Peirce (1870: 370).
[13] Keenan and Faltz (1985: 193).
[14] Peirce (1870: 378).

5 Concluding Remarks

In this paper, we have (i) extended the notion of relational grammar such that it is able to accommodate ternary relations, (ii) illustrated this notion by a fragment of English that deals with transitive and ditransitive phrases, (iii) pointed out certain inadequacies in terms proposed by Peirce, and (iv) given correct interpretations for terms that had been pointed out to be flawed. In addition we would like to point out that our grammar extends the set of syllogisms considerably. For instance, it will be able to identify the argument

Some man gave every lady a rose
Every rose is a flower

Every lady was given a flower

as a valid syllogism of English. With additional rules introducing negative particles *no* and *not* we may end up with about 88 different syllogistic forms.

Our notion of an extended relation algebra as a structure closed with respect to certain operations resembles the notion of a "bonding algebra" proposed by Herzberger.[15] Herzberger proposed a structure closed with respect to relational composition, major permutation, minor permutation, bonding, and relative complement, where major permutation shifts the first argument into final position, minor permutation switches the first two arguments and bonding identifies the last two arguments of a relation. In line with Peirce, Herzberger does not distinguish between the operations of relational composition and Peirce product. This may be satisfactory in the case where only binary relations and sets are considered. However, the operations can be well distinguished: if R is a ternary relation and S is a binary relation, then $R;S$ will return a ternary relation but $R:S$ will return a set. Moreover, the operations turn out not to be sufficient. Some notion of union or intersection is required as is a notion of restriction. We would otherwise not be able to derive an appropriate structure for the tree in Figure 16.

Relational grammar is not compelled to distinguish two variants of a ternary relation depending on the order of their arguments. On the contrary, Peirce's assumption of two different notions for *give* is rather unnatural from the standpoint of natural English where one and the same form is used throughout. If we assume only one predicate for *give* we would then have to derive g_2 from g_1 or g_1 from g_2.[16]

Unlike most conventional linguistic approaches our grammar is semantically driven rather than syntactically driven. The sentences

give a horse to an owner of a horse
give a horse to an owner of that horse

exhibit an almost identical syntactic structure. The only difference is that one structure has an existential quantifier *a* where the other structure has the demonstrative pronoun *that*. Linguists have speculated that quantifiers and demonstrative pronouns belong to one and the same syntactic category of determiners. Under this assumption one should expect that the semantic trees for these expressions are very similar. But under our anlysis, this turns out not to be the case. The respective semantic trees are given in Figure 15 and in Figure 16. The semantic tree for the expression with the anaphoric pronoun *that* is much flatter than the tree for the expression without the anaphoric pronoun. But this is

[15] Herzberger (1981).
[16] This is in fact done in Böttner (To appear).

not surprising since the anaphoric reference requires information given at some location of the tree to be available at a distant location of the tree. It is an open question whether relational semantics has to stay with the flat tree of Figure 16 or can be tailored to fit better a more hierarchical structure.

The flat-tree problem is inherited by any standard one-dimensional representation. Peirce himself proposed a two-dimensional representation better known under the name of existential graphs. Existential graphs have become a major focus in the design of systems of knowledge represention in computer science under the name of conceptual graphs.[17] The problem will be to find uniform procedures to map the variant forms of a natural language syntax to two-dimensional graph structures.

References

[1] Aubert, K. E. (1955) On the foundations of the theory of relations and the logical independence of generalized concepts of reflexivity, simmtery, and transitivity. *Archiv for Mathematik og naturvidenskab* 52, 9-56.

[2] Böttner, M. (1992) Variable-free semantics for anaphora. *Journal of Philosophical Logic* 21, 375-390.

[3] Böttner, M. (1994) Open problems in relational grammar. In *Patrick Suppes. Scientific Philosopher*, Vol. 3, ed. by P. Humphreys, Dordrecht: Kluwer, 319-335.

[4] Böttner, M. (1997) Natural Language. In *Relational Methods in Computer Science*, ed. by Brink, C., Kahl, W. and G. Schmidt, New York: Springer, 229-249.

[5] Böttner, M. (To appear) *Relationale Grammatik*. Tübingen: Niemeyer.

[6] Brink, C. (1978) On Peirce's notation for the logic of relatives. *Transactions of the Charles S. Peirce Society* 14, 285-304.

[7] Brink, C., Kahl, W. and G. Schmidt (1997) *Relational Methods in Computer Science*. New York: Springer.

[8] Carnap, R. (1929) *Abriss der Logistik*. Wien: Springer.

[9] Copi, I. M. and F. Harary (1953) Some Properties of n-Adic Relations. *Portugaliae Mathematica* 12, 143-152.

[10] Herzberger, H. G. (1981) Peirce's Remarkable Theorem. In *Pragmatism and Purpose. Essays presented to Thomas A. Goudge*, ed. by L. W. Sumner, J. G. Slater and F. Wilson. University of Toronto Press, 41-58.

[11] Keenan, E. L. and Faltz, L. M. (1985) *Boolean Semantics for Natural Language*. Dordrecht: Reidel.

[12] Martin, R. M. (1978) Of lovers, servants, and benefactors. *Journal of Philosophic Logic* 7, 27-48.

[17] Sowa (1993).

[13] Peirce, C. S. (1870) Description of a Notation for the Logic of Relatives, resulting from an Amplification of the Conceptions of Boole's Calculus of Logic. *Writings of Charles Peirce, 1867-1871*, Bloomington: Indiana University Press, 1984, 359-429.

[14] Peirce, C. S. (1882) Brief Description of the Algebra of Relatives. *Writings of Charles Peirce, 1879-1884*, Bloomington: Indiana University Press, 1986, 328-333.

[15] Peirce, C. S. (1902) Relatives. *Dictionary of Philosophy and Psychology* Vol. 2, ed. by J. M. Baldwin. MacMillan: New York, London, 447-450.

[16] Sowa, John F. (1993) Relating Diagrams to Logic. In *Conceptual Graphs for Knowledge Representation*, ed. by G. Mineau, B. Moulin and J. F. Sowa. Springer, 1-35.

[17] Suppes, P. (1973) Semantics of context-free fragments of natural languages. In *Approaches to Natural Language* ed. by K.J.J. Hintikka et al. Dordrecht, 370-94.

[18] Suppes, P. (1976) Elimination of quantifiers in the semantics of natural language by use of extended relation algebras. *Revue Internationale de Philosophie* 117-118, 243-59.