Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 95 (2004) 209–225
www.elsevier.com/locate/entcs

Mechanising the Alphabetised Relational Calculus

Gift Nuka [1] Jim Woodcock[2]

Computing Laboratory
University of Kent
Canterbury, United Kingdom

Abstract

In Hoare and He's unifying theories of programming, the alphabetised relational calculus is used to describe and relate different programming paradigms, including functional, imperative, logic, and parallel programming. In this paper, we give a formal semantics of the alphabetised relational calculus, and use our definition to create a deep embedding of the calculus in Z. This allows us to use one of the standard theorem provers for Z, in order to provide mechanised support for reasoning about programs in the unifying theory.

Keywords: Relational Calculus, Verification, UTP.

1 Introduction

Hoare and He [7] introduce unifying theories of programming(UTP), where the alphabetised relational calculus (*ARC*) is used as the basis for unifying the science of programming. In their unifying theory, programs, specifications, and designs are all represented as predicates defining relations between an initial observation involving undashed variables, and a later observation involving dashed variables, in a manner reminiscent of Z's schemas.

ARC adds a theory of alphabets to Tarski's relational calculus language. In this paper, we present a study of *ARC* and its formal semantics, providing an opportunity to reason formally, as well as to gain a greater understanding

[1] Email: G.S.Nuka@kent.ac.uk
[2] Email: J.C.P.Woodcock@kent.ac.uk

doi:10.1016/j.entcs.2004.04.013

of the calculus. This work contributes towards reasoning about programs in the unifying theories, by providing formal verification frameworks and tool support.

We implement the syntax and formal semantics of *ARC* in Z, and use a conventional Z theorem prover to prove theorems about *ARC* and to verify its algebraic laws. We choose to implement *ARC* in Z because we aim to support *Circus* [18,19], an integration of Z [15] and CSP [6], whose semantics is based on the unifying theories and expressed in Z.

The rest of the paper is organised as follows. Section 2 gives a brief introduction to UTP and *ARC*. Section 3 presents the *ARC* model with its syntax and semantics. In section 4, we present the model's implementation in Z/EVES and the relevant algebraic laws. All the proofs for the laws given in this section are given in the full report [10]. Section 5 presents related research, and in Section 6 we give our conclusions.

2 Unifying Theories of Programming

The theoretical basis of the unifying theories of programming is the alphabetised relational calculus: the classical relational calculus of Tarski [16], with an extension giving all relations an alphabet of free variables. Different programming paradigms are expressed in this common notation, and specifications, designs, and implementations are all programs that are represented by relations between initial and final observations. We associate to each program an alphabet, the names of variables that record observations of a particular program state. Variables that are not decorated denote initial observations, while decorated variables denote final observations.

Certain variables are shared with the environment of a program; for example, the variables *okay* and $okay'$ model a class of relations called *designs*. If the observation *okay* is true of a program, then that program has started its execution; while if $okay'$ is true, then the program has terminated or not diverged and the final values of its free variables can be observed.

One notable feature is the treatment of common ideas in the theory; for example, non-determinism in general is treated as a disjunction, sequential composition as relational composition, the conditional as a boolean connective, and parallelism as a restricted kind of conjunction. Healthiness conditions are used to denote feasibility of a program.

The Alphabetised Relational Calculus

The study of the calculus of relations was pioneered by De Morgan, Pierce, and Schröder and was axiomatised by Tarski [16] in the 1940s. The theory

of relational algebras evolved from the axiomatisation of the calculus of relations by Tarski. A *relational algebra* is an algebraic structure that contains a boolean algebra, associative binary operators, an identity element, and forms a distributive lattice. Recently, Maddux has presented an historical study of relational algebras and axiomatisations of the calculus of relations in a modern context [8].

Classical relational calculus has been a very useful framework for the study of mathematics and theory of computer science. *ARC* is based on Tarski's calculus, but great importance is attached to free variables appearing in the predicate. An *alphabetised relation* is a pair $(\alpha p, p)$, where predicate p is the predicate containing no free variables other than those in the alphabet αp. This makes calculations simpler, a property that is valuable in reasoning about programs and in program development.

The alphabet of the relation contains undashed and dashed variable names; the undashed names (input alphabet) are disjoint from the dashed ones (output alphabet).

$$\langle in\alpha(p), out\alpha(p)\rangle \text{ partition } \alpha(p)$$

The undashed variables represent an initial observation and the dashed variables represent the final observation. In some cases the relation is homogeneous; that is, the output variables are input variables decorated with a dash.

$$out\alpha(p) = in\alpha'(p)$$

The signature of the calculus consists of primitive operators, as well as those defined in terms of the primitive ones. Operators such as **true**, negation, disjunction, and existential quantification are primitive, while sequential composition and conditionals are not. General properties of most of these operators include commutativity, associativity, idempotence, and absorption. Other useful operators include the conditional, sequential composition, assignment, non-determinism, skip, miracle, and abort. Other operators can always be added to the calculus as necessary. A detailed account of these operators appears in [7,10].

3 The ARC Model

We present a formal model that captures the semantics of the *ARC* as in [7]. We use predicates to represent the properties of a program, an alphabet to represent the allowed variables, and bindings to represent a set of observations for a particular predicate. The syntax for the predicate is similar to that of

the predicate calculus, and we define two main semantic functions: α, giving the alphabet of the predicate; and β, giving the set of observations that satisfy the predicate.

We use the Z notation to model *ARC*. Z [15,3,14,20] is a notation developed at Oxford University based on predicate calculus and Zermelo-Fraenkel set theory. It is state-oriented and provides simple notations using mostly standard mathematical concepts, and has a structuring mechanism called the schema calculus.

3.1 Syntax

We assume an infinite set of names denoting variables and an infinite set of values, given as *Name* and *Value* respectively. We present the syntax of a predicate in Figure 1 below,

$Pred ::= true_A$	truth		$op ::= \vee$	disjunction
$\mid false_A$	falsity		$\mid \wedge$	conjunction
$\mid Name\ comp_A\ Term$	expression		$\mid \Rightarrow$	implication
$\mid \neg Pred$	negation		$\mid \Leftrightarrow$	equivalence
$\mid Pred\ op\ Pred$	binary operators		$\mid \bigtriangledown$	exclusive or
$\mid Pred \lhd Pred \rhd Pred$	conditional			
$\mid \exists\, n \bullet Pred$	existential quantifier			
$\mid \forall\, n \bullet Pred$	universal quantifier			
$\mid Pred[Name \setminus Term]$	substitution		$comp ::= =\mid\leq\mid\geq$	

Fig. 1. The syntax of a predicate

where n and m range over *Name*. In the rest of the paper we will use p, q, r, and b for predicates, a, A, α, α_1, and α_2 as sets from the set of names, and n, m, x, y, and z for names from the set of names *Name*.

Predicates are generally defined as in classical predicate calculus, except

that predicates carry their alphabets; the constants $true_A$ and $false_A$ in *ARC*, carry a particular alphabet A. The syntax for primitive predicate expressions also requires that we mention the associated alphabet; for $n \in Name$ and $A \in \mathbb{P}\, Name$, $n =_A t$ says that variable n and term t from alphabet A have same values. A *term* is either a *Name* or a *Value* or an expression involving a term. Other predicates are defined conventionally, with negation ($\neg$) being unary, and disjunction ($\vee$), conjunction ($\wedge$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), exclusive or ($\bigtriangledown$) being binary. The conditional is defined in the format in [7], i.e., $p \lhd b \rhd q$. Quantification follows standard notation with implicit typing. Substitution is defined slightly different, where $p[n \setminus m]$ is a substitution of m for n in predicate p.

3.2 Semantics

Alphabets.

Each predicate in *ARC* has an associated alphabet which is made explicit in the predicate or can be calculated. An alphabet denotes the names of variables that have been introduced by the predicate. We represent it as a function from predicates to sets of names. Some predicates have their alphabet explicitly marked with them. For example the alphabet of predicates *true*, *false*, and primitive expressions such as *equality* are always specified. Negating a predicate p does not change its alphabet, such that

$$\alpha(p) = \{n, m\} \Rightarrow \alpha(\neg p) = \{n, m\}$$

The alphabet of a binary predicate is the union of the alphabets of the operands. Quantification removes a variable from the alphabet: we can no longer observe its value.

$$\alpha(\exists\, n \bullet p) = \alpha(p) \setminus \{n\} \qquad \alpha(\forall\, n \bullet p) = \alpha(p) \setminus \{n\}$$

For example, if p is the predicate $\forall\, n \bullet m =_{\{m\}} 1 \wedge n =_{\{n\}} 2$, then $\alpha(p) = \{m\}$. Substitution $p[n \setminus m]$ allows one name m to be systematically substituted for free occurrences of the other name n. It removes n from the alphabet, but introduces the name m instead.

$$\alpha(p[n \setminus m]) = (\alpha(p) \setminus \{n\}) \cup \{m\}$$

If n is bound to some quantifier in p, then substitution does not occur and $n \notin \alpha(p)$

$$\alpha(p[n \setminus m]) = (\alpha(p) \setminus \{n\}) \cup (\{m\} \lhd \{n\} \subseteq \alpha(p) \rhd \alpha(p))$$

Decoration.

The alphabet of a program consists of two sets of variables, the decorated variables referred to as input alphabet (such as x, n) and output alphabet - the dashed variables (such as x', n'). We model variable decoration using the total injection *dash*

$$dash : Name \rightarrowtail Name$$

Bound and Free Variables.

The semantic function σ defines the set of bound variables in a predicate, which is disjoint from the alphabet of the predicate. We need to distinguish between bound and free variables occurring in a predicate so that we can reason about substitution and quantifiers.

$$\sigma : Pred \rightarrow \mathbb{P}\, Name$$

Predicates $true_A$, $false_A$, and primitive expressions like *equality* do not have any bound variables. All variables that appear in them are free. The bound variables of the negation of a predicate are the same as those bound variables of the predicate itself. The set of bound variables of conjunction, disjunction, implication, equivalence are the union of the respective bound variables of the two predicates involved. In a conditional $p \lhd b \rhd q$ the set of bound variables is the union of those variables bound in p, b and q. For every predicate p, b and q

$$\sigma(p \lhd b \rhd q) = \sigma(p) \cup \sigma(b) \cup \sigma(q)$$

Quantification introduces a bound variable, the one quantified, while substitution in general does not change the bound variable set. For every predicate p and q

$$\sigma(\exists\, n \bullet p) = \sigma(p) \cup \{n\}$$
$$\sigma(\forall\, n \bullet p) = \sigma(p) \cup \{n\}$$
$$\sigma(p\,[n \setminus m]) = \sigma(p)$$

If a variable is bound then that variable cannot be substituted. Whenever a new variable would make an already existing variable become bound, then the original variable is renamed. Consider the predicate

$$p \;\widehat{=}\; \exists\, n \bullet n = 1 \wedge m = 0, \qquad \alpha(p) = \{m\}$$

then

$$p[n \setminus m] \;=\; \exists\, n \bullet n = 1 \wedge m = 0, \qquad \alpha(p[n \setminus m]) = \{m\}.$$

The occurrence of n is bound by the existential quantifier, so the substitution does not have any effect on the predicate. In a different case, consider we are substituting for n, in the predicate p above, then

$$p[m \setminus n] \mathrel{\widehat{=}} \exists\, x \bullet x = 1 \wedge n = 0, \qquad \alpha(p[m \setminus n]) = \{n\}.$$

In this case, a new variable x is introduced, which is not observable as n was not previously and since m was free, the new variable n is also free.

Bindings set.

An observation of a particular program can be expressed by a predicate. This can be represented by a set of pairs where the first element in each pair is a variable from the alphabet of the predicate and the other element being a constant value assigned to the particular variable. We represent an observation as a partial function, *Binding*.

$$Binding == Name \pfun Value$$

We model it as a partial function, since some variables from the alphabet set may not be associated with any particular value, or the alphabet of the predicate may not be the entire *Name* set. Observations expressed by a predicate are defined by the semantic function β.

$$\beta : Pred \to \mathbb{P}\, Binding$$

We call $\beta(p)$, for all predicates p, the set of *bindings*, that is a set of observations, but we will use both terms (observations and bindings) interchangeably. The set of bindings for predicate $true_A$ are all the pairs involving alphabet A. The bindings of $true_A$ represent the universal function set with respect to alphabet A. The predicate $false_A$ gives no observations and its set of bindings is empty. Given term t, a set of names A and $n \in Name$

$$\beta(true_A) = \{\, b : Binding \mid \mathrm{dom}\, b = A \,\}$$
$$\beta(false_A) = \emptyset$$

For expressions involving terms, the set of bindings depends on the expression operator $\odot$ which is used.

$$\beta(m \odot_A t) \;=\; \{\, b : Binding \mid \mathrm{dom}\, b = A \wedge m \in \mathrm{dom}\, b \wedge n \wedge b(m) \odot val(t) \,\}$$

where $val(t)$ gives the value after evaluation of term t. Negating a predicate p gives the set of bindings that are not in the set of *bindings* for p: the difference between the universal bindings set for $\alpha(p)$ and the bindings of p.

$$\beta(\neg p) = \beta(true_{\alpha(p)}) \setminus \beta(p)$$

Extension set.

Disjoining and conjoining predicates results in extending the observations described by one predicate to the possible observations from the alphabet of the other predicate. To model the bindings of such predicates we use an *extend* function. Given a set of observations $s \in \mathbb{P}\, Binding$ we can extend this set to a new set by enlarging the domain of every element in s.

$$\begin{array}{|l} extend : \mathbb{P}\, Binding \times \mathbb{P}\, Name \times \mathbb{P}\, Name \to \mathbb{P}\, Binding \\ \hline \forall\, s : \mathbb{P}\, Binding;\ \alpha_1, \alpha_2 : \mathbb{P}\, Name;\ b : Binding \bullet \\ \qquad b \in extend(s, \alpha_1, \alpha_2) \Leftrightarrow \alpha_1 \lhd b \in s \land \operatorname{dom} b = \alpha_1 \cup \alpha_2 \end{array}$$

where $\lhd$ is the Z domain restriction operator.

An extension set $extend(s, \alpha_1, \alpha_2)$ gives a map extension for observations in s. Consider a bindings set for a predicate p, $\beta(p)$; extending this set from $\alpha(p)$ to a set of variables A, where $A \in \mathbb{P}\, Name$, can be represented as $\beta(p)_{+A}$. This is defined as follows

$$\beta(p)_{+A} \mathrel{\widehat{=}} extend(\beta(p), \alpha(p), A)$$

The restricted set.

We use the restricted set as another way of representing the Z domain anti-restriction (domain removal).

$$\begin{array}{|l} nrestrict : Name \times \mathbb{P}\, Binding \to \mathbb{P}\, Binding \\ \hline \forall\, s : \mathbb{P}\, Binding;\ n : Name \bullet nrestrict(n, s) = \{\, b : Binding \bullet \{n\} \mathrel{\lhd\!\!\!-} b \,\} \end{array}$$

We represent the restricted set for $\beta(p)$ and variable n as $\beta(p)_{-n}$. This is defined as

$$\beta(p)_{-n} \mathrel{\widehat{=}} nrestrict(n, \beta(p))$$

This notation is more compact, and we use it to represent removal (anti-restriction).

Disjunction.

When two predicates p and q are disjoined, we may get observations from one predicate or the other. We represent this as a union of two extension sets. We extend $\beta(p)$ to $\alpha(q)$ and also extend $\beta(q)$ to $\alpha(p)$.

$$\beta(p \vee q) = \beta(p)_{+\alpha(q)} \cup \beta(q)_{+\alpha(p)}$$

Conjunction.

If predicates p and q have disjoint alphabets then the set of observations of their conjunction is the union of the observations of p and observations of q. However, if their alphabets are not disjoint, then the variables that are shared between them should agree on their values. If they do not agree, then the bindings of their conjunction is empty. We define this as follows

$$\beta(p \wedge q) = \beta(p)_{+\alpha(q)} \cap \beta(q)_{+\alpha(p)}$$

Quantification.

Existential quantification removes the bound variable from the observable set and we use the domain anti-restriction to represent this.

$$\beta(\exists\, n \bullet p) = \beta(p)_{-n}$$

Universal quantification is the generalisation of the existential quantification and is defined in terms of the existential quantifier.

$$\beta(\forall\, n \bullet p) = \beta(\mathit{true}(\alpha(p) \setminus \{n\})) \setminus \beta(\neg p)_{-n}$$

Substitution replaces one variable, n for another variable m in the set of observable variables of a predicate p, in a way that m is associated with all values previously associated with n. We represent this set by conjunction of p and equality between n and m.

$$\beta(p[n \setminus m]) = \beta(p \wedge (m =_A n))_{-n} \text{ where } A = \alpha(p) \cup \{m\}$$

Other predicates.

Bindings of implications are defined in terms of the bindings of disjunction. And those of equivalence in terms of conjunction and implication, with

exclusive-or defined in terms of the negation of equivalence.

$$\beta(p \Rightarrow q) = \beta(\neg p \lor q)$$
$$\beta(p \Leftrightarrow q) = \beta((p \Rightarrow q) \land (q \Rightarrow p))$$
$$\beta(p \bigtriangledown q) = \beta(\neg(p \Leftrightarrow q))$$
$$\beta(p \lhd b \rhd q) = \beta((b \land p) \lor (\neg b \land q)) \quad \texttt{iff} \quad \alpha(b) \subseteq \alpha(p) = \alpha(q)$$

4 Implementation in Z/EVES

The *ARC* model is in the Z notation and is implemented in Z/EVES: a theorem prover that uses the EVES system [11,12,4], that was built to support proof for ZF set theory. Z paragraphs are translated to the first order logic that EVES uses, and then translated back to Z for the user.

ARC predicates are introduced as a free type definition. We define a function *alpha* for the alphabet and all its axioms are introduced as rules in Z/EVES

$$alpha : Pred \rightarrow \mathbb{P}\, Name$$
$$\langle\langle rule\ alphaTruth \rangle\rangle \forall\, a : \mathbb{P}\, Name \bullet alpha(truth(a)) = a$$

We use the Z/EVES definition option *rule* so that we allow automatic rewriting; that is any occurrence of the left hand side is replaced by the right hand side in a proof without user intervention. We used this option for all the axioms for *alpha*. We use predicates $truth_A$ and $falsity_A$ in Z/EVES for $true_A$ and $false_A$ to differentiate from Z/EVES inbuilt *true* and *false*.

Z allows an axiomatic definition with several predicates conjoined, but Z/EVES is not able to apply rewriting rules using such a definition; each predicate in an axiomatic definition that needs to be used for rewriting has to be defined separately.

We introduce observations in a predicate (*Binding*) as an abbreviation definition.

$$Binding == Name \pfun Value$$

This introduces a set type *Binding*. However, in Z/EVES, to allow the prover to automatically infer about types for such definition, we introduced an assumption rule.

$$theorem\ grule\ BindingType$$
$$Binding \in \mathbb{P}(Name \times Value)$$

The Z/EVES option *grule* is used so that Z/EVES can automatically introduce this assumption whenever a *Binding* type is encountered in proofs.

The bindings function β is defined in the manner of the *alpha* function

$$bindings : Pred \rightarrow \mathbb{P}\, Binding$$

$$\langle\!\langle grule\ bindingsTruth \rangle\!\rangle \forall\, a : \mathbb{P}\, Name \bullet bindings(truth(a)) = \{\, b : Binding \bullet \operatorname{dom} b = a \,\}$$

The *extend* function was used without modification, while we substituted *nrestrict* for *dreduce* as below

$$dreduce : Name \times \mathbb{P}\, Binding \rightarrow \mathbb{P}\, Binding$$

$$\langle\!\langle disabled\ rule\ dreduceDef \rangle\!\rangle \forall\, s : \mathbb{P}\, Binding;\ n : Name \bullet dreduce(n, s) = \{\, b : Binding \mid b \in s \bullet \{n\} \lhd b) \,\}$$

Algebraic Laws

The operators we have defined enjoy several algebraic properties, which can be proved using predicate logic and set theory. A detailed list of these properties is found in [9]. Proofs for these properties are sometimes non-trivial and we employ several sub-lemmas on the function *extend* and *dreduce*. For example, double extension reduces to single extension if the alphabets are related by a subset inclusion.

Lemma 4.1 (Double Extension)

$$(\beta(p)_{+\alpha_1})_{+\alpha_2} = \beta(p)_{+(\alpha_1 \cup \alpha_2)}$$

Other properties include distributivity of *extend* with respect to union, intersection and difference of the bindings argument. Proof for these lemmas involve applying the definition for *extend*.

Lemma 4.2 (Distributivity of extend)

$$(\beta(p) \setminus \beta(q))_{+\alpha_1} = \beta(p)_{+\alpha_1} \setminus \beta(q)_{+\alpha_1}$$

Lemma 4.3 (Bindings of Same Domain)

$$\beta(p)_{+\alpha(p)} = \beta(p)$$

Basic Laws

The properties relating to primitive operators true, disjunction ($\vee$), and conjunction ($\wedge$) include idempotency, commutativity, associativity and absorp-

tion. We examine these properties further.

Idempotence.

To prove that $p \wedge p = p$ in our theory we need to show that this holds with respect to alphabets as well as with respect to the bindings, that is

Law 1 $\beta(p \wedge p) = \beta(p)$ *and* $\alpha(p \wedge p) = \alpha(p)$

The Z/EVES goal for this law is

$$bindings(conj(p, p)) = bindings(p) \ and$$
$$alpha(conj(p, p)) = alpha(p)$$

We have to show that $\wedge$ is idempotent with respect to β and also with respect to α. The proof is trivial, but, for illustration, we show it here

use $bindingsConj[q := p]$;
prove

Theorem proving in Z/EVES is a sequential process, with the goal predicate being transformed after each step until you get the predicate *true*. The graphical interface provides a user with a small set of commands, and you can also use the command window to edit or type in commands. Theorem proving can be tuned to be fully user-driven or semi-automatic by using particular labels ($grule, rule, frule$) to definitions and theorems. A user has a choice to disable a rule if automatic rewriting is not required.

In the proof script above, we use the following definition of bindings of a conjunction.

$$\langle\langle grule\ bindingsConj\rangle\rangle \forall p, q : Pred \bullet bindings(conj(p, q)) =$$
$$extend(bindings(p), alpha(p), alpha(q)) \cap$$
$$extend(bindings(q), alpha(q), alpha(p))$$

This makes the goal rewritten to a new goal

$$p \in Pred$$
$$q \in Pred$$
$$\Rightarrow extend(bindings(p), alpha(p), alpha(p)) \cap$$
$$extend(bindings(p), alpha(p), alpha(p)) = bindings(p)$$

which can be easily proved by inbuilt theorems involving sets. Set intersection is idempotent. The extension of bindings of predicate p to $\alpha(p)$ does not change the bindings set using *lemma 4.3*, which has been implemented in

Z/EVES as an automatic rewrite rule. The goal then reduces to the following, which is further reduced to the predicate true by applying `prove`, which applies the rewriting rules to

$$p \in Pred$$
$$q \in Pred$$
$$\Rightarrow extend(bindings(p), alpha(p), alpha(p)) = bindings(p)$$

to get the goal reduced to $bindings(p) = bindings(p)$.

To be of use in proof each definition has to be labelled, and so *bindingsConj* is the label for the conjunction definition above. Z/EVES stores this as a theorem and depending on what proof option you give it, the theorem can be used as a rewrite rule or introduced as an assumption in the proofs. In this case *grule* in the definition means that this definition can be introduced as an assumption in proofs.

It is trivial to prove the theorems on the alphabet. We have implemented the model in such a way that theorems with respect to the alphabet can be automatically discharged by just using `prove`. This is done by labelling alphabet definitions as *rules*.

Commutativity.

Conjunction and disjunction are commutative in relational calculus and in alphabetised relational calculus with respect to our formalisation.

Law 2 $\beta(p \wedge q) = \beta(q \wedge p)$

The proof for this law requires application of the definition of bindings of conjunction (*bindingsConj*) and instantiation of the necessary parameters. Often, Z/EVES requires such instantiation of variables. Similarly, we can prove that disjunction is commutative, by applying the definition of disjunction (*bindingsDisj*).

Law 3 $\beta(p \vee q) = \beta(q \vee p)$

Associativity.

Conjunction and disjunction are associative.

Law 4 $\beta(p \wedge (q \wedge r)) = \beta((p \wedge q) \wedge r)$

Law 5 $\beta(p \vee (q \vee r)) = \beta((p \vee q) \vee r)$

Proof for these laws require repeated application of the definitions of respective operators (conjunction and disjunction).

Absorption laws.

We can reduce certain compound predicates involving conjunction and disjunction.

Law 6 $\beta(p \wedge (p \vee q)) = \beta(p)$ iff $\alpha(q) \subseteq \alpha(p)$

Law 7 $\beta(p \vee (p \wedge q)) = \beta(p)$ iff $\alpha(q) \subseteq \alpha(p)$

The requirement $\alpha(q) \subseteq \alpha(p)$ is necessary, since in applying β to a disjunction or a conjunction we extend the alphabets of both predicates p and q. We can also prove absorption by using distributivity of conjunction over disjunction.

Distributive Laws.

Conjunction distributes over disjunction as in relational calculus.

Law 8 $\beta(p \wedge (q \vee r) = \beta((p \wedge q) \vee (p \wedge r))$

Similarly, disjunction distributes over conjunction

Law 9 $\beta(p \vee (q \wedge r) = \beta((p \vee q) \wedge (p \vee r))$

Proposition 4.4 *The set of predicates Pred, constants true and false, operators $\vee$ and $\wedge$, negation ($\neg$) form a Boolean algebra.*

Proof. We have already shown that alphabetised predicates with respect to $\wedge$ and $\vee$ are a distributive lattice. That is $\wedge$ and $\vee$ are associative, commutative and distributive. We will list laws to show that *false* and *true* are the minimum and maximum of the lattice. This completes the proof. ∎

Predicate *true* is the identity with respect to conjunction and it is a maximum element of the lattice.

Law 10 $\beta((p \wedge (\mathit{truth}(\alpha(p))))) \;=\; \beta(p)$

Law 11 $\beta((p \vee (\mathit{truth}(\alpha(p))))) \;=\; \beta(\mathit{truth}(\alpha p))$

and *false* is an identity with respect to disjunction.

Law 12 $\beta((p \wedge (\mathit{falsity}(\alpha p)))) \;=\; \beta(\mathit{falsity}(\alpha p))$

Law 13 $\beta(p \vee (\mathit{falsity}(\alpha p))) \;=\; \beta(p)$

Negation complements the lattice.

One property of predicate calculus is that negation complements the lattice. We show here that negation acts as a complement of our distributive lattice

Law 14 $\beta(p \wedge \neg(p)) \;=\; \beta(\mathit{falsity}_{\alpha(p)})$

Law 15 $\beta((p \vee \neg(p))) \;=\; \beta(\mathit{truth}_{\alpha(p)})$

Applying negation twice to a particular predicate results into the original predicate.

Law 16 $\beta(\neg\neg p) = \beta(p)$

De Morgan's laws.

Negation satisfies De Morgan's laws as follows

Law 17 $\beta(\neg(p \wedge q)) = \beta(\neg(p) \vee \neg(q))$

Law 18 $\beta(\neg(p \vee q))) = \beta(\neg(p) \wedge \neg(q))$

Predicate laws.

The universal and existential quantifiers are both idempotent

Law 19 $\beta(\forall\, n \bullet \forall\, n \bullet p) = \beta(\forall\, n \bullet p)$

Law 20 $\beta(\exists\, n \bullet \exists\, n \bullet p) = \beta(\exists\, n \bullet p)$

De Morgan's laws.

Quantification observes De Morgan's Laws.

Law 21 $\beta(\neg\, \exists\, n \bullet p) = \beta(\forall\, n \bullet \neg p)$

Law 22 $\beta(\neg\, \forall\, n \bullet p) = \beta(\exists\, n \bullet \neg p)$

Commutativity laws.

The universal and existential quantifiers are commutative.

Law 23 $\beta(\forall\, n \bullet \forall\, m \bullet p) = \beta(\forall\, m \bullet \forall\, n \bullet p)$

Law 24 $\beta(\exists\, n \bullet \exists\, m \bullet p) = \beta(\exists\, m \bullet \exists\, n \bullet p)$

5 Related Work

We are not aware of any other research on mechanisation of the alphabetised relational calculus, but several other implementations of relational algebras have been done and include proof systems like RALL (Relational Algebraic Language and Logic), RALF, ARA and a mechanisation of δRA. In our other work, we have implemented the model in ProofPower; a theorem prover for higher order logic that also supports Z.

RALL [17] is a proof assistant for relational algebras, particularly using a method that manipulates the atomicity property of relational algebras. The relation between complex algebras and atomic structures is stated in the correspondence theorem which states that any relation algebra can be embedded

as an atomic structure. RALL supports the full language of abstract element-free relational algebra and is implemented in the Isabelle/HOL theorem prover using HOL as the object logic. In our calculus, we do not use the atomicity property and require that our predicates are alphabetised.

RALF [2] is another mechanisation of the relational algebras. It has a graphical interface and provides interactive theorem proving by manipulation of relation-algebraic formula. As opposed to the other two mechanisations (*RALL and Display logic*), RALF is not built on any generic theorem prover and is stand-alone. This is a major setback for developers, as extending its proof system and libraries would be costly. It also uses an element-free algebra.

In [5], Dawson provides a mechanised proof system for a relational algebra, (δRA) using δRA Display logic [1]. Display logic is a proof system that is based on Gentzen sequent calculus and is a syntactic non-classical logic proof system. δRA is implemented in the general theorem prover Isabelle and built directly on Isabelle's meta-logic. Unlike δRA, our calculus is based on classical relational logic of Tarski.

ARA [13] (Automatic Theorem prover for Relational Algebras) is another proof assistant for relational algebras. It is a theorem prover that is based on Gordeev's Reduction Predicate Calculi for n-variable logic and allows first order variable proofs.

6 Conclusions

We have presented a theory of the alphabetised relational calculus and its formal definition in the Z notation. Our approach has been to embed both the syntax and the semantics into Z so that we can reason and prove theorems on the language. An alternative approach is direct translation of the semantics into equivalent Z denotations. In this approach the syntax of the language is not implemented in Z and thus making it difficult to reason and make reference to propositions on the whole language.

We intend to further develop and test our theory on several applications in the unifying theories of programming. Currently we are mechanising the fixed point theory in *ARC* and the UTP theory of shared variables. Mechanisation of the alphabetised relational calculus, we believe, will enable provision of proof support to applications and languages that use the C.A.R. Hoare and He Jifeng's unifying theories of programming.

Acknowledgement

The authors thank anonymous referees and Ana Cavalcanti for their helpful comments on this paper and Arther Hughes who read an earlier version of this paper.

References

[1] Belnap, N. D., *Display Logic*, Journal of Philosophical Logic **11** (1982), pp. 375–417.

[2] Berghammer, R. and C. Schmidt, *RALL: A Relation algebraic formula manipulation system and proof checker*, in: *Proc. 3rd Conference on Algebraic Methodology and Software Technology - AMAST '93*, Workshops in Computing **1249** (1993), pp. 407–408.

[3] Brien, S. and J. Nicholls, "Z Base Standard, Version 1.0. Technical Monograph TM-PRG-107," Oxford University Computing Laboratory, Oxford - UK, 1992.

[4] Craigen, D., S. Kromodimoeljo, I. Meisels, W. Pase and M. Saaltink, *EVES: an overview*, in: *Proceedings of the VDM 91 (Formal Software Develpment Methods)* (1991).

[5] Dawson, J. E. and R. Goré, *A Mechanised Proof System for Relation Algebra using Display Logic*, in: *Logics in Artificial Intelligence: European Workshop, JELIA'98* (1998).

[6] Hoare., C., "Communicating Sequential Processes," Prentice Hall International, 1985.

[7] Hoare, C. and J. He, "Unifying Theories of Programming," Prentice Hall, 1998.

[8] Maddux, R., *The origin of relation algebras in the development and axiomatization of the calculus of relations*, Studia Logica **6** (1991), pp. 423–455.

[9] Morgan, C. and J. Sanders, "Laws of the Logical calculi. Technical Report PRG-78," Programming Research group, Oxford, England, 1989.

[10] Nuka, G. and J. Woodcock, *Mechanising Alphabetised Relational Calculus*, Technical Report (2002).

[11] Saaltink, M., "Z and EVES. Technical Report," ORA Canada, 1991.

[12] Saaltink, M., *The Z/EVES system*, in: J. P. Bowen, M. G. Hinchey and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, Lecture Notes in Computer Science **1212** (1997), pp. 72–85.

[13] Sinz, C., *System description: ARA - an automatic theorem prover for relation algebras*, in: D. McAllester, editor, *Automated Deduction CADE-17*, number 1831 in LNAI (2000), pp. 177–182.

[14] Spivey, J., "Understanding Z: A Specification Language and its Formal Semantics," Cambridge University Press, 1988.

[15] Spivey, J., "The Z Notation: A Reference Manual," Prentice Hall International, 1992, second edition edition.

[16] Tarski, A., *On the calculus of relations*, Journal of Symbolic Logic **6** (1941), pp. 73–89.

[17] von Oheimb, D. and T. F. Gritzner, *RALL: Machine-supported proofs for Relation Algebra*, in: *Proceedings of CADE-14*, Lecture Notes in Computer Science **1249** (1997), pp. 380–394.

[18] Woodcock, J. and A. Cavalcanti, *Circus: A concurrent refinement language*, Technical Report (2001).

[19] Woodcock, J. and A. Cavalcanti, *A concurrent language for refinement*, in: *IWFM'01: 5th Irish Workshop in Formal Methods* (2001).

[20] Woodcock, J. and J. Davies, "Using Z Specification, Refinement, and Proof," Prentice Hall, 1996.