# Widening the Representation Bottleneck: A Functional Implementation of Relational Programming

Dave Cattrall & Colin Runciman
University of York, York, YO1 5DD, England
{dmc,colin}@minster.york.ac.uk

**Abstract**

Relational programming is a generalisation of functional programming that includes aspects of logic programming. We describe a relational language, Drusilla, that retains the lazy, polymorphic and higher-order aspects of functional languages and the flexible handling of non-determinism and search based computation of logic languages. As a result it offers certain economy of expression not found in functional or logic programming. However a complex implementation, using a combination of polymorphic type inference and automatic program transformation to select appropriate representations, is needed to support this language.

## 1 Introduction

At FPCA 1981 [14] and in subsequent papers, [15, 16, 17], MacLennan introduced *relational programming* — a programming paradigm with a number of novel features:

- entire relations are manipulated as data;
- program definitions are represented as relations;
- a set of relational operators (themselves relations), are available for manipulating both data and program.

Relational programming should, in principle, be a generalisation of functional programming because mathematically the concept of a relation is more general than that of a function.

MacLennan implemented his relational language RPL [2] through an explicit compromise of the relational abstraction — a fixed relation representation scheme split relational programming into two worlds: a world of *intensional* relations and a world of *extensional* relations. Intensional relations corresponded to computable functions and extensional relations were data structures. This scheme directly inhibits freedom of expression because it splits the relational operators into two classes: one applicable solely to intensional relations and one applicable solely to extensional relations.

MacLennan's 'four relational programs' [17] provide evidence of this. For each operation over relations it is often necessary to give more than one definition — one for each representation. For example, consider MacLennan's word frequency program shown in Figure 1. Two domain restriction operators must be defined — one for restricting an intensional relation (`restrict`), and one for restricting an extensional relation (`->`). Even then, both operators require the restricting relation to be extensionally represented.

```
s =  {(1,"to"),(2,"be"),(3,"or"),
      (4,"not"),(5,"to"),(6,"be")}
freq r = unimg (inv r) | size
unimg r = (dom r)  restrict (r elimg)
elimg t x = rng [un x -> t]
```

Figure 1: Word frequency program in RPL

A program in RPL is a collection of function definitions. Since functions are deterministic non-determinism is not naturally handled. Only intensional operators are applicable to program code. Most operators are extensional, and only applicable to data. As a result of this intensional/extensional divide the expressive power of RPL is greatly compromised. RPL is really a functional language with a few FP-style combining forms for functions and a collection of operators for manipulating relational data structures.

The aim of our Drusilla system is to select relation representations automatically, to provide an implementation of relational programming that is more faithful to MacLennan's original conception. It uses a generalisation of Milner's [18] type inference algorithm combined with automatic program transformation to achieve this.

Section 2 describes the design of the relational language Drusilla. Section 3 describes the techniques used in its implementation. Section 4 evaluates the resulting system. Section 5 reviews related work on relational languages and automatic representation selection. Section 6 draws some conclusions.

## 2 Design of the Relational Language Drusilla

### 2.1 Ideas Underlying The Design

The Drusilla language is declarative, applicative, and purely relational — there are no side effects or imperative features of any kind. A program is a collection of relation definitions and a query to be evaluated. The order in which definitions are given has no significance.

We adopt the slogan: *all the world is a binary relation.* The built-in operators are applicable to relations, irrespective of how they are defined. There is no functional application of user-defined relations because it cannot generally be determined whether a relation is functional. However all the built-in operators known to be functional may be used either as relational values or as functional operators.

The built-in relational operators of Drusilla are defined in Tables 1, 2 and 3.

| Cardinality of a relation<br>`[card]` :: (A ↔ B) ↔ num<br>card r = *primitive to calculus* |
|---|
| Domain of a relation<br>`[dom]` :: (A ↔ B) ↔ (A ↔ un)<br>x (dom r) Unit ⇔ ∃ y . x r y |
| Range of a relation<br>`[rng]` :: (A ↔ B) ↔ (B ↔ un)<br>y (rng r) Unit ⇔ ∃ x . x r y |
| Inverse of a relation<br>`[inv]` :: (A ↔ B) ↔ (B ↔ A)<br>x (inv r) y ⇔ y r x |
| Set form of relation<br>`[set]` :: (A ↔ B) ↔ ((A × B) ↔ un)<br>(x,y) (set r) Unit ⇔ x r y |
| Complement of a relation<br>`[neg]` :: (A ↔ B) ↔ (A ↔ B)<br>x (neg r) y ⇔ ¬ x r y |
| Relation containership<br>`[cont]` :: (A ↔ B) ↔ (A × B)<br>r `[cont]` (x,y) ⇔ x r y |

Table 1: Primitive unary operators

| Relation intersection<br>`[/\]` :: (A ↔ B × A ↔ B) ↔ (A ↔ B)<br>x (r `/\` s) y ⇔ x r y ∧ x s y |
|---|
| Relation union<br>`[\/]` :: (A ↔ B × A ↔ B) ↔ (A × B)<br>x (r `\/` s) y ⇔ x r y ∨ x s y |
| Relation composition<br>`[;]` :: (A ↔ B × B ↔ C) ↔ (A ↔ C)<br>x (r;s) y ⇔ ∃ z . x r z ∧ z s y |
| Parallel composition<br>`[||]` :: (A ↔ B × C ↔ D) ↔<br>((A × C) ↔ (B × D))<br>(a,c) (r `||`s) (b,d) ⇔ a r b ∧ c s d |
| Dual composition<br>`[#]` :: (A ↔ B × A ↔ C) ↔ (A ↔ (B × C))<br>x (r # s) (y,z) ⇔ x r y ∧ x s z |
| Domain restriction<br>`[<<]` :: (A ↔ un × A ↔ B) ↔ (A ↔ B)<br>x (s `<<` r) y ⇔ x s Unit ∧ x r y |
| Range restriction<br>`[>>]` :: (A ↔ B × B ↔ un) ↔ (A ↔ B)<br>x (r `>>` s) y ⇔ x r y ∧ y s Unit |

Table 2: Primitive binary operators

| Domain anti-restriction<br>`[<-]` :: (A ↔ un × A ↔ B) ↔ (A ↔ B)<br>s `<-` r ≡ neg s `<<` r |
|---|
| Range anti-restriction<br>`[->]` :: (A ↔ B × B ↔ un) ↔ (A ↔ B)<br>r `->` s ≡ r `>>` neg s |
| Relation difference<br>`[\]` :: (A ↔ B × A ↔ B) ↔ (A ↔ B)<br>r \ s ≡ r `/\` neg s |
| Relation overide<br>`[@]` :: (A ↔ B × A ↔ B) ↔ (A ↔ B)<br>r @ s ≡ dom s `<-` r `\/` s |
| Image of a set under a relation<br>`[img]` :: (A ↔ B × A ↔ un) ↔ (B ↔ un)<br>r img s ≡ rng (s `<<` r) |

Table 3: Non-primitive Operators

### 2.2 Expressions and Types

A **basic value** is a (real) number, character string or `Unit`. The corresponding types are *num string* and *un*.

A **tuple** is a sequence of elements of mixed type separated by commas and enclosed in parentheses, e.g.

```
("Jones",Unit,39).
```

If $t_1 \ \ldots \ t_n$ are types then $(t_1 \times \ldots \times t_n)$ is the type of tuples with objects of these types as components. For example, the above tuple has type (string × un × num). Such product types allow binary relations to model n-ary relations. An n-ary relation is a set of n-tuples. If each n-tuple is split into two tuples, the first regarded as a domain value and the second as a range value, then that relation becomes a set of pairs of tuples and hence binary.

An expression that evaluates to a relation is termed a *relation designation.*

An **elementary relation designation** is a program definition, a formal parameter, an operator as a first-class relational value denoted syntactically by use of square brackets (e.g. `[inv]`, `[;]`), or a relation extension of the form:

$$\{(d_1, r_1), \ \ldots \ , (d_n, r_n)\}$$

Each pair $(d_i, r_i)$ is of the same type and denotes a mapping from domain value $d_i$ to range value $r_i$.

Sets are not manipulated as structures distinct from relations. Any set with elements of type t is identified with a binary relation of type $t \leftrightarrow un$. For example, the set of numbers {1,2,3} is represented by the relation:

{(1,Unit),(2,Unit),(3,Unit)}.

Relations defined in this way may be used as data structures or as program code relating inputs to outputs.

**Compound relation designations** are formed from simpler ones by application of operators to operands, e.g. `inv r` (the inverse of relation r), `r ; s` (the composition of relations r and s).

If $t_1$ and $t_2$ are types, then $t_1 \leftrightarrow t_2$ is the type of a relation with domain type $t_1$ and range type $t_2$. Higher-order relations are permitted. A relation that is a value in the domain of another relation is termed a *value-relation.*

Polymorphism is introduced via type variables (upper case letters), which are understood to be universally quantified.

### Relational Sections

A relation may be specialised by partial instantiation of its domain and/or range. For example, the expression `[- 1]` used in the definition of `ackA` in Figure 2 denotes the predecessor relation, which holds between any number and that number minus one. This is syntactic sugar for the more explicit specialisation syntax `(_,1)[-]`.

We use the notation $v :: t$ to denote that value $v$ is of type $t$. If

$$\mathtt{R} :: (t_1 \times \ldots \times t_n) \leftrightarrow u$$

then for any $S \subset \{1, \ldots, n\}$ given $\forall\ i \in S,\ v_i :: t_i$, then an expression of the form

$$(c_1, \ \ldots \ , c_n)R \quad where \quad \begin{array}{l} c_i = v_i, \ if\ i \in S \\ c_i = \_\ , \ otherwise \end{array}$$

is a relation of type

$$\prod_{1 \le i \le n,\ i \notin S} t_i \leftrightarrow u$$

A similiar convention applies when the range type is a product and is to be specialised.

```
ack = ackA @ ackB @ ackC.
(_,y) ackC  {(0,y+1)}.
(x,_) ackB  {(0,1)} ; (x-1,_)ack.
(x,_) ackA  [-1] ; (x,_)ack ; (x-1,_)ack.
```

Figure 2: Ackermann's function as a relation

### 2.3 Definitions

A program is a collection of definitions of which there are three forms.

**Value definitions** bind an identifier to an expression that can denote any type of Drusilla value including of course a relation. Example value definitions are `fact` and and `fib` in Figure 3.

```
-- recursive definition of factorial
fact = id # ([- 1] ; fact) ; [*] @ {(0,1)}.

-- recursive definition of fibonacci
fib = [- 1] ; fib  # ([- 2] ; fib) ; [+]
            @ {(0,0),(1,1)}.
```

Figure 3: Recursively defined mathematical functions

**Fully parameterised definitions** have the form:

*domainTuple relationName definingExpression*

Identifier *relationName* is bound to a relation between values in a domain, denoted by a tuple of formal parameters, *domainTuple*, and an expression, *definingExpression*, which defines the range value that is related to a given domain value. Such definitions are really functions but are treated as any other relation by Drusilla. The `get` definitions shown in Figure 4 illustrate how this form of definition can extract tuple elements. This program shows how such definitions can be used to perform database style projection operations.

```
-- workers relation
workers = {((1,"Simon","Oxford","13k"),Unit),
           ((2,"Dave","York","13k"),Unit),
           ((3,"Paul","Cambridge","17.5k"),Unit),
           ((4,"Tim","Reading","18k"),Unit)}.

-- tuple extraction relations
(code,name,place,pay) getCode code.
(code,name,place,pay) getName name.
(code,name,place,pay) getPlace place.
(code,name,place,pay) getPay pay.
(code,name,place,pay) getNamePay (name,pay).
(code,name,place,pay) getCodePlace (code,place).

-- projection relations over workers
projectCode = getCode img workers.
projectName = getName img workers.
projectPlace = getPlace img workers.
projectPay = getPay img workers.
projectNamePay = getNamePay img workers.
projectCodePlace = getCodePlace img workers.
```

Figure 4: Example relation projections

**Partially parameterised definitions** have the form

*domainTuple relationName definingExpression*

Identifier *relationName* is again bound to a relation. Values in the domain of *relationName* are syntactically described by a tuple, *domainTuple*. This tuple contains *formal* and *anonymous* parameters. Some components of this tuple are *named* by the formal parameters, the rest are left *anonymous* (denoted by underscores).

Examples of partially parameterised definitions are `ackA`, `ackB` and `ackC` in Figure 2.

## 3 The Implementation of Drusilla

### 3.1 The Representation Bottleneck

MacLennan's fixed representation scheme splits relational programming into two worlds: a world of intensional relations and a world of extensional relations. This *representation bottleneck* keeps the programmer thinking in terms of computable functions and data structures, compromising relational abstraction.

MacLennan [16] refers to his operator division as the 'elimination of (ad-hoc) polymorphism' — operators are only defined for one representation even if they are implementable and useful for others. If the representation bottleneck is to be removed then each relational operator must be defined for all possible representations of its arguments. A single symbol should be used for each operator. The system must automatically select representations and resolve operator overloading to preserve relational abstraction.

### 3.2 Relation Representations

Drusilla is implemented in Miranda[1] and uses a variety of relation representation techniques.

#### Extensional Representation

An *extensional representation* stores the relation elements in a data structure. This method can only be used if the programmer defines the relation in extension or gives some formula for generating the relation elements. Such a formula can generate an infinite extensional relation because Miranda is lazy. The extensional representation used is a list of pairs of related domain and range values. For example the set of weekend days is represented by the list:

```
[("Sat",Unit),("Sun",Unit)]
```

We term such lists *association lists*, denoted by the symbol $AL$.

#### Intensional Representation

**Set-valued functions** can be used to represent both functional and non-functional computable relations. If a relation `R` relates domain value $x$ to range values $y_1, \ldots y_n$ then the set-valued function, `f`, used to represent `R` maps `x` to the set $\{y_1, \ldots, y_n\}$. For example, the union of two relations `S` and `T` represented respectively by the functions `f` and `g`:

$$\text{x } (\text{S} \cup \text{T}) \text{ y} \Leftrightarrow \text{y} \in \text{f x} \vee \text{y} \in \text{g x}$$

We denote set-valued functions by the symbol $SF$.

**Characteristic functions** are boolean-valued functions that can be used to represent both functional and non-functional computable relations. If a relation `R` relates domain value $x$ to a range value $y$ then the characteristic function, `f`, used to represent `R` maps the pair `(x,y)` to `True`. If $x$ and $y$ are not related under `R`, then `f` maps the pair `(x,y)` to `False`. For example, the union of two relations `S` and `T` represented respectively by the functions `f` and `g`:

$$\text{x } (\text{S} \cup \text{T}) \text{ y} \Leftrightarrow \text{f (x,y)} \vee \text{g (x,y)}$$

We denote characteristic functions by the symbol $CF$.

### 3.3 A Type System for Representation Inference

The *typed representation inference system* of Drusilla widens the representation bottleneck. It is a generalisation of Milner's type system [18] that infers ad-hoc as well as parametric polymorphism. Type systems normally infer a type for every expression and every subexpression within a program. Typed representation inference yields not only a type but also suitable representations for each expression and subexpression. The analysis not only ensures a program is type correct but also automatically selects representations.

The typed representations of basic values are fixed: *num*, *string* and *un*.

If $t_1 \ldots t_n$ are typed representations then $(t_1 \times \ldots \times t_n)$ is the typed representation of tuples with objects of these typed representations as components.

If $t_1$ and $t_2$ are typed representations, and $rep \in \{\text{AL}, \text{SF}, \text{CF}\}$ then $rep[t_1 \leftrightarrow t_2]$ is the typed representation of any relation with representation $rep$ and domain values with typed representation $t_1$ and range values with typed representation $t_2$.

Polymorphism is denoted by typed representation variables of which there are two forms: *ordinary* and *equality*. An ordinary variable can unify with the typed representation of any value. An equality variable can *only* unify with the typed representation of those values for which equality is defined. Equality variables are distinguished from normal variables by an '=' superscript.

Each Drusilla primitive operator is associated with a *set* of typed representation inference rules. Each rule within this set captures some argument-result representation constraint which dictates what the result representation will be when that operator is applied to an argument of a certain representation. However this representation may be automatically coerced later if necessary. An association list may often be coerced to a set-valued function or to a characteristic function and a set-valued function function may often be coerced to a characteristic function.

#### Example

Consider the relation composition operator, `[;]`. Its type inference rule

$$\frac{x :: P \leftrightarrow Q \qquad y :: Q \leftrightarrow R}{x \ ; \ y :: P \leftrightarrow R}$$

is combined with the representation constraints shown in Figure 5, to form the set of typed representation inference rules presented in Figure 6.

| r \ s | AL | SF | CF |
|---|---|---|---|
| **AL** | AL | AL | CF |
| **SF** | SF | SF | CF |
| **CF** | CF | $\perp$ | $\perp$ |

Figure 5: representation constraints for relation composition

Each typed representation rule reflects the natural computational constraints of type and representation for composition. □

#### The Algorithm

The typed representation inference algorithm proceeds as a standard polymorphic typechecker [5]. It begins with the leaves of the expression tree and proceeds bottom up moving toward the root. The abstract syntax of Drusilla expressions is defined so that each compound designation consists of an operator applied to one subexpression. Whenever an operator node is reached each of its typed representation inference rules is, in turn, applied to the subexpression. Each application of an inference rule involves unification which generates a substitution for type variables. This yields a set of alternative possible substitutions and hence a set of alternative possible typed representations for the expression, rather than one principal type.

Typed representation analysis selects representations by building a new version of the expression tree for each substitution generated. Each substitution assigns a type and a representation to type variables in its associated expression and hence to the relations those variables are attached to.

[1] Miranda is a trademark of Research Software

$$\frac{\text{r} :: \text{AL}[\text{A} \leftrightarrow B^{=}] \quad \text{s} :: \text{AL}[B^{=} \leftrightarrow \text{C}]}{\text{r ; s} :: \text{AL}[\text{A} \leftrightarrow \text{C}]}$$

$$\frac{\text{r} :: \text{AL}[A^{=} \leftrightarrow \text{B}] \quad \text{s} :: \text{CF}[\text{B} \leftrightarrow \text{C}]}{\text{r ; s} :: \text{CF}[A^{=} \leftrightarrow \text{C}]}$$

$$\frac{\text{r} :: \text{SF}[\text{A} \leftrightarrow \text{B}] \quad \text{s} :: \text{SF}[\text{B} \leftrightarrow \text{C}]}{\text{r ; s} :: \text{SF}[\text{A} \leftrightarrow \text{C}]}$$

$$\frac{\text{r} :: \text{AL}[\text{A} \leftrightarrow \text{B}] \quad \text{s} :: \text{SF}[\text{B} \leftrightarrow \text{C}]}{\text{r ; s} :: \text{AL}[\text{A} \leftrightarrow \text{C}]}$$

$$\frac{\text{r} :: \text{SF}[\text{A} \leftrightarrow B^{=}] \quad \text{s} :: \text{AL}[B^{=} \leftrightarrow \text{C}]}{\text{r ; s} :: \text{SF}[\text{A} \leftrightarrow \text{C}]}$$

$$\frac{\text{r} :: \text{SF}[\text{A} \leftrightarrow \text{B}] \quad \text{s} :: \text{CF}[\text{B} \leftrightarrow \text{C}]}{\text{r ; s} :: \text{CF}[\text{A} \leftrightarrow \text{C}]}$$

$$\frac{\text{r} :: \text{CF}[\text{A} \leftrightarrow \text{B}] \quad \text{s} :: \text{AL}[\text{B} \leftrightarrow C^{=}]}{\text{r ; s} :: \text{CF}[\text{A} \leftrightarrow C^{=}]}$$

$$\frac{\text{r} :: \text{CF}[\text{A} \leftrightarrow \text{B}] \quad \text{s} :: \text{SF}[\text{B} \leftrightarrow \text{C}]}{\text{r ; s} :: \perp_{TR}}$$

$$\frac{\text{r} :: \text{CF}[\text{A} \leftrightarrow \text{B}] \quad \text{s} :: \text{CF}[\text{B} \leftrightarrow \text{C}]}{\text{r ; s} :: \perp_{TR}}$$

Figure 6: Typed representation inference rules for relation composition

When the inference algorithm reaches the root node of the expression tree a set of syntactically equivalent resolved expression trees has been generated. Each tree has its own unique typed representation and associated substitution for type variables. Every relation and subexpression within each tree has a typed representation.

### Example

Figure 7 shows a Drusilla translation of MacLennan's word frequency program shown in Figure 1. The typed representations inferred for this program are shown in Figure 8.

```
s = {(1,"to"),(2,"be"),(3,"or"),
     (4,"not"),(5,"to"),(6, "be")}.
freq = [inv] ; unimg ; [; [card]].
(r) unimg  dom r << (_,r)elimg.
(e,r) elimg  rng ({(e,Unit)} << r).
```

Figure 7: Word frequency program in Drusilla

The word table relation `s` is naturally represented as an association list. The algorithm first inverts (`s`) to create a new relation between words and their positions (also represented as an association list). The unit image (`unimg`) of this relation generates a relation between each word and the set of positions in which that word occurs. The value-relation in `unimg` must map domain elements to range elements. Therefore it can be represented by either an association list or a set-valued function but not by a characteristic function. Relation `freq` derives the frequencies by taking the cardinality of each occurrence set. The value-relation for `freq` must be represented by an association list so that it can be inverted and its unit image taken. If instead the value-relation was represented by a set-valued or characteristic function then its inverse would be represented by a characteristic function and therefore could not be used by `unimg`.

| | |
|---|---|
| s :: | AL[num $\leftrightarrow$ string] |
| elimg :: | SF[($O^{=} \times$ AL[$O^{=} \leftrightarrow$ P]) $\leftrightarrow$ AL[P $\leftrightarrow$ un]]<br>SF[(G $\times$ SF[G $\leftrightarrow$ J]) $\leftrightarrow$ AL[J $\leftrightarrow$ un]] |
| unimg :: | SF[AL[$U^{=} \leftrightarrow$ V] $\leftrightarrow$ AL[$U^{=} \leftrightarrow$ AL[V $\leftrightarrow$ un]]]<br>SF[SF[X $\leftrightarrow$ T] $\leftrightarrow$ SF[X $\leftrightarrow$ AL[T $\leftrightarrow$ un]]] |
| freq :: | SF[AL[I $\leftrightarrow$ $C^{=}$] $\leftrightarrow$ AL[$C^{=} \leftrightarrow$ num]] |

Figure 8: Typed representations for word frequency program

The unit image of a relation is calculated by restricting its element image (`elimg`) to its domain (`dom`). The image of an element under a relation is the set of range elements to which it relates. The value-relation for `elimg` must generate range elements from a given domain element and is therefore represented by an association list or set-valued function.

The typed representation system has inferred the appropriate representation constraints and the appropriate operator definitions. As a result, whereas in RPL two domain restriction operators are needed, one for extensional relations (`->`), the other for intensional relations (`restrict`), in Drusilla only one (`<<`) is required. □

### 3.4 Algebraic Manipulation of Drusilla Expressions

Although typed representation inference selects representations for expressions, some expressions may be unrepresentable and hence unusable.

As MacLennan [16] notes, a relational language is amenable to symbolic manipulation. Laws of equivalence between relational expressions are well documented and laws specific to the operators used in Drusilla can easily be formulated.

Algebraic manipulation may be used to transform problematic expressions to forms that can be represented. This preserves the mathematical meaning of a relational program but changes its operational behaviour.

Rewrite rules are extracted from laws, which are equations of the form:

```
expL = expR
```

Here `expL` and `expR` are two relational expressions whose equivalence has been asserted by the implementor. Expressions in laws typically consist of a number of free variables glued together by the relational operators. An example law is:

```
(r >> s) ; t = r ; (s << t)
```

Given an expression that has the undefined typed representation, $\perp_{TR}$, how can we tell whether a particular rewrite rule will improve the representation of that expression? We apply typed representation inference to laws to generate this information. For this purpose we first create a relation definition from each law, `expL = expR`:

```
pars lawExample (expL,expR)
```

The defining expression is the tuple (expL,expR) and the tuple of formal parameters (pars) contains L ∪ R where L and R are the sets of free variables occurring in expressions expL and expR respectively. Analysis of such a definition yields a set of definitions each of which has its own typed representation, including typed representations for the law expressions expL and expR and all their subexpressions.

We want rewrite rules with the typed representation undefined on the left hand side but defined on the right hand side.

### Example

Analysis of the law

```
inv s ; inv r = inv (r ; s)
```

reveals that the rewrite rule

```
inv s ; inv r  →  inv (r ; s)
```

generates a defined typed representation for the following cases:

| s | r | inv s;inv r | inv (r;s) |
|---|---|---|---|
| SF[W ↔ X] | SF[V ↔ W] | $\perp_{TR}$ | CF[X ↔ V] |
| CF[T ↔ U] | SF[S ↔ T] | $\perp_{TR}$ | CF[U ↔ S] |

However the reverse rule

```
inv (r ; s)  →  inv s ; inv r
```

never improves representation. □

### Rewriting Expressions

An expression C is an instance of the expression A on the left hand side of a rewrite rule, $A \rightarrow B$, if there exists a substitution $\sigma$ for the free variables in A such that:

$$\sigma \mathtt{A} = \mathtt{C}$$

Suppose C has the undefined typed representation $\perp_{TR}$. If the typed representation of C and all its subexpressions match the corresponding subexpressions in $\sigma$A then there exists a substitution $\delta$ for the typed representation variables in $\sigma$A such that

$$\delta\ (\sigma A)^{TR} \quad = \quad C^{TR}$$

and expression C can be rewritten as $\delta(\sigma$B). This rewrites the unrepresentable expression C to a mathematically equivalent expression that has a defined typed representation. Moreover, there is no need to re-apply typed representation inference to the new expression.

The manipulation strategy used in the Drusilla system is based on meta-level inference [3]. We use a number of *methods* for rewriting expressions. Each method combines rewrite rules to transform some class of expression. Some conventional rewriting strategies consider a space limited only by the different ways in which an expression can be rewritten. Such search spaces suffer combinatorial increase in size when new rewrite rules are introduced. By contrast the search space of meta-level inference is limited by the ways in which rules can be usefully combined. Therefore as the number of rules increases there is less growth in search space size.

In turns out that only a few rules improve representation. Also, in practice, few expressions are without representation since comparatively few operators are undefined for comparatively few argument representations. Moreover, all of the programs in this paper are executable in the Drusilla system without any manipulation.

### 3.5 Architecture of the System

A program is initially parsed into abstract syntax using Fairbairns's 'let form follow function' [8] style of parser construction.

The maximally strong components of the program reference graph are then isolated. A *reduced reference graph* is obtained by replacing each node in the original reference graph with the component to which it belongs. The reduced graph is used in subsequent analyses.

Types are first inferred using Milner's [18] algorithm. Any program that does not typecheck is rejected.

Typed representation inference is applied to the maximal strong components in the reverse of their depth-first ordering. Algebraic manipulation is applied to any defining expression that has the undefined typed representation, $\perp_{TR}$. If this manipulation is successful then a defined typed representation is created for the expression; otherwise analysis ceases and the program is rejected as unusable.

If the program is successfully analysed then it can be interpreted — expressions are interactively evaluated in a *read, evaluate, print* loop. Alternatively, a Drusilla program can be compiled into a Miranda program. The ability to compile demonstrates that typed representation 'unravels' all relation representations at progam analysis time.

Drusilla programs run considerably faster when compiled than when interpreted. For example, the Drusilla solution to the eight queens problem, shown in Figure 12, when interpreted directly takes several hours to produce all 92 solutions, but when compiled produces all solutions in a few minutes — about twice as long as the time taken by a conventional Miranda solution.

## 4 Evaluation of the Drusilla System

We have described new implementation techniques for relational programming, designed to widen the representation bottleneck and have explained how they are used in the Drusilla system. Their use complicates the implementation but can be justified if expressive power is significantly increased as a result.

### 4.1 A Comparison of RPL and Drusilla

Compare the RPL word frequency program in Figure 1 with the Drusilla translation shown in Figure 7. In RPL two domain restriction operators are needed — one for restricting extensional relations (->), the other for restricting intensional relations (restrict) and the programmer must decide which operator is appropriate. However, in Drusilla only one domain restriction operator (<<) is required. This is a direct result of the typed representation analysis shown in Figure 8. It can be seen from the analysis that each of the Drusilla definitions elimg and unimg corresponds to two separate definitions in RPL — one for each representation combination.

By hiding relation representations from the programmer, The system better preserves relational abstraction. The use of set-valued functions rather than point-to-point functions means that non-determinism can be better handled.

### 4.2 Functional Aspects of Drusilla

Functional language expressions are built from function applications. Drusilla expressions are similarly constructed

from applications of built-in operators. The operators form a fixed set of combining forms as advocated by Backus [1] in his functional language FP but they combine relations rather than functions.

Lists are central to all functional languages but are not present in Drusilla. However, any list of length $n$ can be modelled in Drusilla as a *sequence* — a relation between the first $n$ natural numbers and the elements of the list. Each number dictates the position of the element related to it in the sequence. For example the list [10,20,30,40] would be represented by the relation:

```
{(1,10),(2,20),(3,30),(4,40)}
```

The standard functional language list manipulating operations can be defined for sequences in Drusilla as shown in Figure 9. The basic construction operation is `cons` and the basic destruction operations are `hd` and `tl`. Definition `append` relates a pair of sequences to the sequence that results from appending them and `length` relates a sequence to the number of elements it contains. Higher-order operations such as `filter`, `foldl`, `foldr` and `map` can also be defined. All definitions are polymorphic as in a functional language.

```
hd = [cont](1,_).
tl = {({},Unit)} <- tl2.
(s) tl2  dom s << [1 +] ; s.
(el, seq) cons {(1,el)} \/ inv (inv seq ; [1 +]).
(_,t) append  {({},t)} \/
              (hd # (tl ; (_,t)append) ; cons).
length = {({},0)} \/ (tl ; length ; [+ 1]).
(p,_) filter  id @ (hd # (tl ; (p,_)filter) ;
              (snd @ (fst ; p << cons))).
(f,_) map {({},{})} \/
          (hd ; f # (tl ; (f,_)map) ; cons).
(op,x,_) foldr  {({},x)} \/
          (hd # (tl ; (op,x,_)foldr) ; op).
(op,x,_) foldl  {({},x)} \/
          (hd ; (x,_)op # tl ; (op,_,_)foldl).
(x) id x.
(x,y) fst x.
(x,y) snd y.
```

Figure 9: List operations defined for sequences

All Drusilla expressions are evaluated lazily — no subexpression is evaluated until its value is needed to produce a result for the whole expression. Compound designations are lazily evaluated in the same way as function applications in a functional language — each operator is evaluated before the relation(s) it manipulates and those relations are only evaluated as much as the operator requires. At the implementation level, Drusilla inherits laziness from Miranda. Relational laziness permits non-strict definitions, e.g. `fst` and `snd` in Figure 9 are non-strict in their second and first arguments respectively. It also allows manipulation of infinite data structures, e.g. `nats` and `squares` in Figure 10.

### 4.3 Non-determinism and Relational Abstraction

Reasoning about non-determinism is difficult in functional languages because their underlying mathematical model can only describe deterministic (functional) relationships. A

```
nats = {(Unit,0)} \/ (nats ; [+ 1]).
squares = inv nats << sq.
(n) sq  n * n.
```

Figure 10: Examples of infinite relational data structures

functional program might represent a non-deterministic computation by lazily generating a list of results [25]. For example, in Figure 11 `perms` returns all possible permutations of a given list and `insert` returns all possible non-deterministic insertions of an element into a list.

However, Drusilla is based on relations, which may denote non-functional relationships and hence have potential to describe non-deterministic (search based) computations. Figure 11 shows how `perm` may be viewed as a relation between two sequences. Similarly `insert` is a relation between a pair `(x,s)` and a sequence `t` which is generated by non-deterministically inserting element `x` into sequence `s`. This permits reasoning about the permutation and insert relationships without the need to map over lists. At the implementation level the Drusilla interpreter exploits a lazy list technique explained by Wadler [25] to handle non-determinism, but the relational abstraction permits lists of alternatives to be hidden from the programmer.

```
perms [] = [[]]
perms (h:t) =
   concat (map (insert h) (perms t))

insert x [] = [[x]]
insert x (h:t) =
   (x : h : t) : map (h :) (insert x t)
```

```
perm = {({},{})} \/ (hd # (tl ; perm) ; insert).

(x,_) insert (x,_)cons \/
             (hd # (tl ; (x,_)insert) ; cons).
```

Figure 11: Permutations solutions in Miranda (upper) and Drusilla (lower)

This simplicity together with higher-order, polymorphic relations permits concise definition of some non-deterministic programs. For example, `perm` could be defined more concisely as the fold right (`foldr` in Figure 9) of non-deterministic `insert`:

```
perm = (insert,{},_)foldr
```

A more substantial example is the Drusilla solution to the 8-queens problem presented in Figure 12.

### 4.4 Logic Aspects of Drusilla

In Prolog relationships between objects are expressed using predicate calculus (a form of *relational calculus*) and a rule of inference *deduces* new relationships from existing ones. A relational language uses a *relational algebra* not a relational calculus and new relationships are constructed from existing ones by *algebraic operations*. The presence of operations for direct construction of new relationships means that object

```
queens = [- 1] ; queens # newQueen -> attack
               ; relAdd @ {(0,{})}.
newQueen = colPositions ; [cont]
(n) colPositions {(n,1),(n,2),(n,3),(n,4),
                  (n,5),(n,6),(n,7),(n,8)}.
(_,pos) attack  [cont] ; (_,pos)check.
(r,(x,y)) relAdd  r \/ {(x,y)}.
((i,_),(m,n)) check {(n,Unit)} \/
    ([- n] ; {(i-m,Unit)} \/ {(m-i,Unit)}).
```

Figure 12: Drusilla solution to eight queens

names are used less in construction of expressions. Arguably this also clarifies the logic of relationships in a program.

No rule of inference is present in Drusilla but such rules may be regarded as implementation mechanisms rather than as conceptual features. The conceptual feature is the ability to express relationships between objects at a high level while relying upon the system to assimilate them. The inference rule is just one assimilation technique for establishing whether relationships hold, while reduction, as used by Drusilla, is another.

Relation level negation in Drusilla is not negation-as-failure as found in Prolog. Instead the complement of a relation, like the inverse, may be represented as a characteristic function. Evaluation uses standard laws of relations:

$$\texttt{x (neg r) y} \Leftrightarrow \neg \texttt{x r y}$$
$$\texttt{x (inv r) y} \Leftrightarrow \texttt{y r x}$$

A Drusilla version of a typical logic program for inferring ancestral relationships in a family tree is shown in Figure 13.

```
parent = {("Drusus","Germanicus"),
          ("Antonia","Germanicus"),
          ("Germanicus","Caligula"),
          ("Agrippina","Caligula"),
          ("Germanicus","Drusilla"),
          ("Agrippina","Drusilla")}.
male = {("Drusus",Unit),("Germanicus",Unit),
        ("Claudius",Unit),("Caligula",Unit)}.
female = neg male.
father = male << parent.
mother = female << parent.
grandfather = father ; parent.
parentCouple = parent ; inv parent \ id.
sibling = inv parent ; parent \ id.
sister = female << sibling.
uncle = (brother ; parent) \/
          (parentCouple ; sister ; parent).
```

Figure 13: Ancestors Program in Drusilla

## 5 Related Work

### 5.1 Related Relational Language Work

Möller [19] suggests a programming style based on n-ary, heterogeneous relations. Sets are treated as unary relations and tuples are used to denote relation elements. Möller's main concern is with algebraic properties of relations. His approach allows code to be kept symmetric and hence avoids heavy use of the relation inverse. Also his join operation can be used to express the concept of relation restriction.

Legrand [12] uses n-ary rather than binary relations. A n-ary function `f` may be defined as a $(n+1)-ary$ relation `R` such that:

$$R = \{(x_1, x_2, \ \ldots \ , x_n, y) \mid y = f(x_1, \ x_2, \ \ldots \ , x_n)\}$$

Legrand notes that relations are more general than functions because they associate a possibly infinite set of values to their arguments and proposes GREL, which is the development of a functional language, GRAAL, towards relations. He uses set-valued functions to model relations. The result of applying such a function is a stream of values returned one at a time to the calling relation/function. Special forms and functions are included to build non-functional relations. The most important of these is the `union` construct, which applies a number of functions to given arguments and collects their multiple results. Legrand compromises relational abstraction by fixing relation representations as set-valued functions. Ideally the programmer should be permitted to reason about relationships and not have to consider the multiple results.

Ruby [24] is a language based on binary relations, designed for use in describing hardware algorithms and circuits. A Ruby circuit description is a binary relation between signals. Circuit descriptions in Ruby are built up hierarchically using operators that form a relational algebra. All the Ruby operators are either present in Drusilla or easily definable. Sheeran and Jones [24, 11] uses Ruby to specify hardware algorithms for butterfly networks of chips. Ruby is associated with a collection of transformation rules based on algebraic laws over relational expressions. Rossen [21, 20] describes a framework based on the Isabelle theorem prover that allows a Ruby circuit description that is inefficient or even impossible to implement to be transformed using these rules in a provably correct way. Ruby was designed as a domain specific language for high-level circuit descriptions, not as a general programming language. Also Ruby is not fully relational since tuples and lists are used as the basic data structure. The operators used in Ruby appear to be powerful and widely applicable to relations and would therefore be useful in any relational programming language. Also algebraic laws could similarly be used to derive efficient algorithms for a full programming language.

### 5.2 Related Representation Selection Work

An overview of representation selection up to 1978 is given by Low [13]. He observes that automatic data structure selection should allow the programmer to use abstract objects denoted by abstract data types without concern for their underlying representation. The data structures for implementing them should be chosen automatically from a number of alternative representations. Selection decisions should be based on information gathered from static analysis or the user. In Drusilla the programmer sees relations but not their representations. A relation typically has several possible representations which are found using typed representation analysis.

In the 1970's much work on automatic data structure selection was undertaken for implementation of the SETL language. Schwartz [23] and Schonenberg [22] describe the same compiler optimisations to allow selection of appropriate data structures for abstract objects, such as sets and mappings, in SETL programs. The SETL designers did not

consider computable functions as representations. Despite its 'very high level' aspirations the SETL language is very much imperative — a factor that complicates the analysis of programs. Consequently, a wide variety of analysis techniques is required — Schwartz [23] describes nine different types of analysis used by the compiler.

Paige [4] introduces a declarative, set-theoretic, high-level, programming language named SQ2+ (set queries with fixed points). Subtypings (type containments) are inferred from SQ2+ queries in a first-order, parametric, monomorphic subtyping theory. Type variables occurring in SQ2+ subtypings are uniquely interpreted as finite universal sets called *bases* which are, in principle, the same as in SETL. They are used to create aggregate data structures that avoid data replication and implement logical, associative access operations. This work adds type theory to the SETL basing approach to representation selection by establishing a link between monomorphic subtypes and representations. Our work identifies a similar link between (ad-hoc and parametric) polymorphic types and representation.

### 5.3 The Haskell Type Class System

Type classes [26] are an extension to the traditional Milner type system and have been included in the language Haskell [9]. They subsume many uses of ad-hoc polymorphism and enhance the expressive power of a polymorphic language. Would it be possible to use type classes to implement the ad-hoc polymorphism of Drusilla operators? A class `Relation` could be defined and contain a member function for each relational operator. Each instance of this class would apply to a particular representation and contain those operator functions that return results of that representation. When a class instance is defined its type is substituted for the type variable associated with that class. If this instance is of polymorphic type then it becomes monomorphic when its type is substituted for the class type variable. i.e. it must denote the same instance of that polymorphic type at each occurrence of the class type variable. If ad-hoc polymorphism is used to hide relation representations behind the operators that process them then the parametric polymorphism of those operators is lost. So Haskell type classes could be used to implement operator ad-hoc polymorphism but parametric polymorphism would be lost in the process.

## 6 Conclusion

Our new implementation techniques for relational programming allow Drusilla to have many desirable aspects of functional and logic languages: polymorphism, higher-order relations, lazy evaluation, non-determinism, reasoning about entity relationships. This is significant since Hudak [10] observes that previous functional-logic integrations have not been entirely satisfactory in the context of higher-order definitions and lazy evaluation.

The success of the more sophisticated implementation is due to typed representation inference which widens the representation bottleneck. The analysis better preserves relation abstraction and consequently simplifies reasoning about relationships and non-determinism. Typed representation inference is based on Milner type inference and may be used as a general mechanism for implementing ad-hoc polymorphism in programming languages.

Algebraic manipulation can be used to transform expressions that have no representation into mathematically equivalent expressions that can be implemented. Manipulation is helped by law analysis which isolates those laws which, when used as a rewrite rule in a specific direction, improve expression representation.

A brief outline of this system was given in [7] and a full description is given in Cattrall's D.Phil. thesis [6].

## 7 Acknowledgements

Thanks to SERC for a studentship that financed Cattrall's D.Phil. work, badminton and beer. Thanks also to the referrees and David Wakeling for comments on preliminary versions of this paper.

## References

[1] J. Backus. Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(9):613 – 641, August 1978.

[2] J. Brown and S. Mitton. Relational programming: Design and implementation of a prototype interpreter. Master's thesis, Naval Postgraduate School, Monterey, California, June 1985.

[3] A. Bundy and B. Welham. Using meta-level inference for selective application of rewrite rule sets in algebraic manipulation. *Artificial Intelligence*, 16:189 – 212, 1981.

[4] J. Cai, Ph. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Möller, editor, *Constructing Programs from Specifications*. North-Holland, 1991.

[5] L. Cardelli. Basic polymorphic type checking. *Science of Computer Programming*, 8(2), 1987.

[6] D.M. Cattrall. The design and implementation of a relational programming system. D.Phil. thesis YCST 93/01, Department of Computer Science, University of York, England, January 1993.

[7] D.M. Cattrall and C. Runciman. A relational programming system with inferred representations. In M. Bruynooghe and M. Wirsing, editors, *4th International Symposium, Programming Language Implementation and Logic Programming, Leuven, Belgium*, pages 475–476. Springer-Verlag, August 1992. LNCS 631.

[8] Jon Fairbairn. Making form follow function: An exercise in functional programming style. *Software — Practice and Experience*, 17(6):379 – 386, June 1987.

[9] Hudak, Wadler Peyton Jones, Fairbairn Boutel, Guzman Fasel, Hughes Hammond, Kieburtz Johnsson, Partain Nikhil, and Peterson. Report on the programming lanuguage haskell a non-strict, purely functional language version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, May 1992.

[10] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3), September 1989.

[11] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[12] R. Legrand. Extending functional programming towards relations. In H. Ganzinger, editor, *European Symposium On Programming '88*, pages 206 – 220. Springer-Verlag, 1988. LNCS 300.

[13] J.R. Low. Automatic data structure selection: An example and overview. *Communications of the ACM*, 21(5):376 – 385, May 1978.

[14] B. MacLennan. Introduction to relational programming. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 213 – 220. ACM Press, 1981.

[15] B. MacLennan. Overview of relational programming. *ACM SIGPLAN Notices*, 18(3):36 – 45, March 1983.

[16] B. MacLennan. Relational programming. Technical Report NPS52-83-012, Naval Postgraduate School, Monterey, California, September 1983.

[17] B. MacLennan. Four relational programs. *ACM SIGPLAN Notices*, 23(1):109 – 119, January 1988.

[18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[19] B. Möller. Join algebra. Papers of the 42nd meeting of I.F.I.P. Working Group 2.1, January 1991.

[20] L. Rossen. Proving (facts about) Ruby. In *Proceedings of the IVth Banff Higher Order Workshop*, November 15 1990.

[21] L. Rossen. Ruby algebra. In G.Jones and M.Sheeran, editors, *Workshop on Designing Correct Circuits*. Springer-Verlag, 1990.

[22] E. Schonberg and J.T. Schwartz. An automatic technique for selection of data representations in SETL programs. *ACM TOPLAS*, 3(2):126 – 143, April 1981.

[23] J.T. Schwartz. Automatic data structure choice in a language of very high level. *Communications of the ACM*, 18(12), December 1975.

[24] M. Sheeran. Describing butterfly networks in Ruby. In K. Davis and J. Hughes, editors, *Proceedings of the 1989 Glasgow Functional Programming Workshop*, pages 182 – 205. Springer-Verlag, 1990.

[25] P. Wadler. How to replace failure by a list of successes. *Proccedings of Functional Programming Languages and Computer Architecture*, pages 113–128, September 1985. LNCS 201.

[26] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages*, pages 60 – 76, January 1989.