The Design and Implementation of a Relational Programming System

David Michael Cattrall B.Sc.

Submitted for the Degree of Doctor of Philosophy

University of York

Department of Computer Science

October 1992

Contents

List of Figures

List of Tables

Dedication

To May and Laurie for all their love and the first eighteen years,
To Lol for encouraging me to think for myself,
To Louise for all her love and affection

Acknowledgements

So many people to thank, so little space ...

Thank you to everyone in the York functional programming group and the computer science department as a whole for providing a friendly and relaxed work environment.

Special thanks must go to my supervisor Colin Runciman for his unfailing guidance. He was always encouraging, full of ideas and never failed to give constructive feedback on my work. Particularly I must thank him for his perseverance in those early days when I took so much more interest in badminton and socialising than in computer science.

In the f.p. group I must especially thank David Wakeling for being such a valuable sounding board. Every idea in this thesis took shape while being bounced off him. He always gave an intelligent, objective response to my ideas.

Several people on the more social front of life and computer science must be thanked.

Louise Carney distracted me away from writing thesis when my sanity depended on it and coped with me in 'work mode'.

Iain Checkland was always ready to go for a beer, chat about women and organise the badminton club.

Simon Dobson was always ready to go for a beer and chat about computer science and women.

Andy Brown was always ready to go for a beer and go for women.

Paul Butcher for invaluable chats over beer or tea, typically about type systems.

Others who were always ready to buy a round: Clive Adams, Paul Andrews, Daphne Saines and Tim Blanchard.

Finally I must thank the person who invented badminton, whoever he or she may have been, for inventing the world's greatest and most social sport. I have made many great friends while playing baddas at York both playing staff matches and running the student club. There are too many people to mention them all, but those who deserve a special mention for the good times are: Iain Checkland, Andy Brown, Mark Wilkinson, Akil Chomoko, Ashar Kahn, Kev Gilholm, Kev Patel, Richard 'Dicky' Davies, Jo MacLeod, Alpesh Doshi, Anthony Jones, Mark Lawrie, Lisa Hogg, Anne Edlin, Nicky Monoghan, Harriet Somers, Lindsey Brown, Andy Vickers and Jayne Graham.

Thanks to you all for the great games and fun nights out!

Declaration

A brief description of the Drusilla system appears in PLILP' 92 — the fourth symposium on programming language implementation and logic programming [21]. This paper contains material present in chapters 3,4 and 5 of this thesis.

A 'paper of the thesis' will be submitted to FPCA '93.

This thesis presents algorithms that are used in the implementation of Drusilla, all of which have been implemented as programs in Miranda[1]. All Drusilla programs presented have been tested in the Drusilla system. To quote Ian Toyn [113]: 'How easy this thesis would have been, and how wrong would it have been, had they not'.

[1]Miranda is a trademark of Research Software

Abstract

The declarative class of computer languages consists mainly of two paradigms — the logic and the functional. Much research has been devoted in recent years to the integration of the two with the aim of securing the advantages of both without retaining their disadvantages. To date this research has, arguably, been less fruitful than initially hoped. A large number of composite functional/logical languages have been proposed but have generally been marred by the lack of a firm, cohesive, mathematical basis. More recently new declarative paradigms, equational and constraint languages, have been advocated. These however do not fully encompass those features we perceive as being central to functional and logic languages. The crucial functional features are higher-order definitions, static polymorphic typing, applicative expressions and laziness. The crucial logic features are ability to reason about both functional and non-functional relationships and to handle computations involving search.

This thesis advocates a new declarative paradigm which lies midway between functional and logic languages — the so-called relational paradigm. In a relational language program and data alike are denoted by relations. All expressions are relations constructed from simpler expressions using operators which form a relational algebra. The impetus for use of relations in a declarative language comes from observations concerning their connection to functional and logic programming. Relations are mathematically more general than functions modelling non-functional as well as functional relationships. They also form the basis of many logic languages, for example, Prolog.

This thesis proposes a new relational language based entirely on binary relations, named Drusilla. We demonstrate the functional and logic aspects of Drusilla. It retains the higher-order objects and polymorphism found in modern functional languages but handles non-determinism and models relationships between objects in the manner of a logic language with notion of algorithm being composed of logic and control elements. Different programming styles — functional, logic and relational — are illustrated.

However, such expressive power does not come for free; it has associated with it a high cost of implementation. Two main techniques are used in the necessarily complex language interpreter. A *type inference system* checks programs to ensure they are meaningful and simultaneously performs automatic representation selection for relations. A *symbolic manipulation system* transforms programs to improve efficiency of expressions and to increase the number of possible representations for relations while preserving program meaning.

Chapter 1

Introduction

> Children starve while boots costing many thousands of dollars leave their mark upon the surface of the moon *Watchmen*

1.1 Merging Functional and Logic Programming

The functional and logic programming paradigms have both found their niche in computer science. In many respects they are very similar — both are declarative, have a firm mathematical basis, and permit problems to be reasoned about at a very high-level — but certain significant differences exist between them, their semantics are quite incompatible, and they offer expressive power in contrasting areas. This has provided much impetus to research into combining aspects of both paradigms into a single framework. The aim of this is to secure the advantages of both in a single language without retaining any of the disadvantages. Such a language, if it could be created, would be extremely expressive. Unfortunately this has proved to be a difficult problem — as Hudak [43] observes, many proposals have been made but none are completely satisfactory, especially in the context of higher-order functions and lazy evaluation.

This thesis advocates a new declarative paradigm, relational programming. Although this paradigm is significant in its own right, this thesis aims to show that it may be regarded as a generalisation of functional programming that encompasses aspects of logic programming.

1.2 Functional Programming

Bird and Wadler [9] state that programming in a functional language consists of building definitions and using the computer to evaluate expressions. The primary role of a programmer is to construct a function to solve a given problem. Each function definition gives a name to an expression that is built from function applications. The output of a functional program is a pure mathematical function of its inputs. Functional programs use variables to denote values in the mathematical sense as opposed to the conventional procedural programming sense where variables denote storage locations containing different values at different

times. The focus is on describing data values by expressions. Examples of modern functional languages are LML, Miranda[1] and Haskell.

Some of the recognised good points of functional languages are:

Higher-order functions give good scope for software reuse.

Lazy evaluation allows infinite data structures to be handled.

Static polymorphic type inference detects many program errors at compile time.

Applicative expression construction naturally controls program flow. Data values are communicated between definitions by function application.

Some of the bad points are:

Non-functional relationships are difficult to express.

Non-determinism and search based computations are poorly handled.

Functions are directional — they can only be used to map arguments to results.

1.3 Logic Programming

Hogger [42] states that a logic program consists of sentences expressing knowledge relevant to the problem that the program is intended to solve. The sentences are logical assertions describing relationships between entities. It is the task of the programmer to formulate these sentences about the problem domain. The programmer can then ask questions about this problem and the computer can automatically derive answers by reasoning about the assertions. Reasoning is achieved by manipulating these sentences using logical inference. The most common example of a logic language is Prolog. Some of the main good points of logic languages are:

Predicates (relations) allow both functional and non-functional relationships between entities to be easily expressed.

Non-determinism and search based computation is naturally controlled.

Predicates are polymodal — input and output terms are not predetermined.

Some of the failings of logic languages are:

Predicates must be first-order because unification, which is used as the parameter passing mechanism, requires equality to be defined for all terms.

Program control flow is difficult to handle.

No applicative expression construction — logical variables are needed to communicate data values between terms.

[1]Miranda is a trademark of Research Software

1.4 Relational Programming

The term relational programming is often used synonymously with logic programming. However, the context in which this thesis uses it is quite different.

Relational programming is similar to functional programming — it consists of building definitions and using the computer to evaluate expressions. The primary role of a programmer is to construct a *relation* to solve a given problem. Each *relation definition* gives a name to an expression that is built from function applications where each function is one of a fixed set of operators that forms a *relational algebra*. A relational program is a pure mathematical relation that relates inputs to outputs. However, relational programming should be a generalisation of functional programming since mathematically relations are generalisations of functions.

Relational programming is also similar to logic programming — each relation describes some relationship between entities and each relation definition forms a sentence expressing knowledge relevant to the problem that the program is intended to solve. The programmer formulates these sentences about the problem domain, then asks questions about the problem and the computer automatically derives answers by reasoning about the assertions.

This thesis is an exploration of the feasibility of relational programming as a new declarative paradigm and attempts to answer two main questions:

1. Can relational programming be implemented in its full generality?
2. To what extent does a relational language merge aspects of functional and logic programming into a single, unified framework?

The main results of the research are that a relational language can be implemented and does indeed combine aspects of functional and logic programming. However the implementation requires computationally expensive static analysis, and power of expression in a fully relational language is limited by the absence of application for functional relations and data structure constructor functions.

1.5 Structure of Thesis

Chapter 2 reviews previous attempts to merge functional and logic programming and the use of relations in other languages. This chapter may be read in isolation — it justifies the direction taken in the thesis.

Chapter 3 introduces a new relational programming language named Drusilla. A typed relational calculus is formulated as a mathematical model for this language and used to define the relational operators that are primitive to Drusilla. This chapter can be read on its own, but it does refer to material in chapter 2.

Chapter 4 describes possible representations for relations and an algorithm, based on Milner type inference, that automatically infers relation representations. This algorithm forms the heart of the Drusilla implementation.

Chapter 5 explains how algebraic manipulation can be used to give representations to those expressions that typed representation inference cannot represent. Chapter 5 should be read after chapters 3 and 4.

Chapter 6 describes the implementation of Drusilla explaining how the program analysis and manipulation techniques described in chapters 4 and 5 interact and how expressions are evaluated.

Chapter 7 evaluates the success of relational programming as embodied in Drusilla. The extent to which it merges functional and logic programming is examined. It is essential to read chapter 3 before this chapter, and it is beneficial to have read chapters 4, 5 and 6 if the implementation evaluation is to be appreciated.

Chapter 8 draws conclusions about the research work described in the thesis.

Chapter 2

Related Work

2.1 Introduction

This chapter discusses language development work that is related to relational programming. Section 2.2 surveys the approaches previously taken to to merging functional and logic programming. Section 2.3 assesses the relevance of the relational calculus and algebra based languages used for querying relational databases. Section 2.4 discusses the use of relations in programming languages. Section 2.5 ties together the literature and considers the direction to be taken in the development of relational programming.

2.2 Merging Functional and Logic Programming

Many attempts have been made to merge functional and logic programming and numerous new languages proposed. DeGroot and Lindstrom [31] give a comprehensive review. This survey groups the approaches into a number of categories.

2.2.1 Dual Interpreters

This method juxtaposes a functional language interpreter and a logic language interpreter. Each interpreter is capable of invoking the other. This allows part of a program to be written in a functional language and part written in a logic language.

LOGLISP, developed by Robinson [87] in the late 1970's, was the first approach to integration. LOGLISP is based on the idea of implementing Kowalski's logic programming within Lisp.

The University of Salford Lisp/Prolog system [4] takes a similar approach. The motivation behind this system is the belief that Prolog is best implemented in conjunction with a more conventional language. Lisp is regarded as a more natural mate than any imperative language.

The main problem with these systems is that they are not really unified — fairly complex mechanisms must be defined to interface the two language components.

These languages simply give the sum of the functional and logic parts. The aim of integration is to produce a language that is more than the sum of the parts.

2.2.2 Symmetric Functional and Logic Combinations

This approach combines functional and logic components into a single framework to avoid having two separate languages and a necessarily complex semantic interface between them.

APPLOG [24] integrates Prolog and Lisp directly, extending Prolog to embrace applicative expressions. This allows ideas to be expressed in either an applicative or logical style. Cohen claims the result is a powerful language with features such as function application, logical inference and pattern directed invocation.

RF-Maple [115] is a union of R-Maple (a concurrent logic language which uses explicit quantifiers) and F-Maple (a functional language). Voda claims that quantifiers remove the need for the Prolog cut construct. RF-Maple tries to strike a balance between control over program execution and program meaning — programs being, in theory, formulas of predicate calculus. Logic programs in R-Maple are based on the *generate and test* paradigm of problem solving — a program typically having the form:

```
find x in { G(|x) ; T(x) }
```

This has the declarative reading:

$$\exists x \ \{ \ G(x) \ \& \ T(x) \ \}$$

`find` is a quantifier, `G(|x)` is the generator and `T(x)` is the test.

RELFUN [10] integrates functions and relations at the level of their definitions through recursion equations and Horn clauses. All RELFUN definitions are generalised Horn clauses (facts and rules) called valued clauses. These allow arbitrary terms, not only goals, as the premises of rules assigning a value to each resolution of a goal with a clause. For relation definitions, valued clauses behave as logic language definitions except that on success they return the value `true` in addition to binding possible request variables; on fail they yield the value unknown. If used for function definitions valued clauses behave similarly to directed conditional equations or conditional term rewriting rules. The difference between functions and relations is that the value returned from a function can be an arbitrary term, not just a truth value. Functions, like relations, may bind request variables and evaluate non-deterministically.

The designers of **LEAF** [6] (Logic, Equations, And Functions) argue that semantic compatibility should make the integration of functional and logic languages conceptually simpler. Semantic compatibility requires the two languages to operate on the same data and to share both the basic control mechanism (rewriting) and the basic parameter passing and return mechanism (pattern matching or unification). The logic component combines Horn clause logic and equational theories with constructor functions. The functional component is essentially the same but relations are changed to tuple-valued functions with the addition of

mode declarations and some syntactic conditions which guarantee determinancy. Component integration is achieved by allowing mutual invocation.

DIALOG [111] facilitates the construction of interactive theorem proving systems in the medium of Lisp. The functional component allows the user to enter functional definitions in the modern pattern matching style. The logic programming component uses many sorted logic. Interplay between the functional and logic parts is achieved by allowing variables in the logic component to be instantiated to identifiers that represent functions defined in the functional component. Use of such a variable causes the function it represents to be called.

Uniform [54], a language based upon augmented unification, is an attempt to combine features of Lisp, actor languages and logic languages into a single framework. All Uniform programs are an extension of the unification process. The language works in a continual *read, unify, print* loop. The user types a set of assertions and then types in expressions which are unified with those assertions. Unification is augmented — two expressions unify if they unify syntactically or if their equivalence can be deduced from assertions stating what is equal to what. As a result of this the same program may be used as a function, an inverse function, a predicate, a pattern or a generator. However, Uniform, has so little control that it is often forced into combinatorial searches.

These languages have no clear underlying mathematical model and their separate language components require a complex interface. As a result their semantics are more complex than those of a conventional functional or logic language.

2.2.3 Extensions to Functional Programming

As the problems of juxtaposing two different interpreters were realised much research was focused on extending functional languages with various constructs designed to incorporate powers associated with logic languages — features like logical variables, unification, non-determinism and logical inference.

Darlington [27] proposes an extension of the set abstraction (or list comprehension) construct found in many functional programming languages. This extension involves removing the requirement to base every abstraction on a predefined set and moving to **absolute set abstraction** where the members of the set are defined implicitly by a set of conditions. The conditions are equations involving functional expressions. For example:

```
split l → {l1,l2 | append l1 l2 = l}
```

Here `l1` and `l2` act as logical variables. The aim of this extension is to improve the handling of non-determinism in functional languages.

The presence of unification in logic languages gives certain expressive power not possessed by functional languages. For this reason it is included in the language **HASL** [2] (a descendent of SASL) in the form of a new expression which Abramson calls a *one-way unification based conditional binding*. The limitation of one way unification is such that in

```
A {- B
```

where {- is the left crossbow operator, only A may contain variables to be instantiated. This operator is embedded in the expression

```
A {- B => C ; D
```

which means unify B with A and if unification succeeds, the value of the expression is the value of C with any of the variables of A occurring in C replaced by the bindings established by unification. Otherwise, if the unification fails, the value of the expression is the value of D totally unaffected by any of the bindings involved in the failed unification. Although this new expression form adds unification to a functional language it does not appear to give any significant increase in expressive power.

Qute [95] uses unification, rather than pattern matching, as its parameter passing and variable binding mechanism. By virtue of unification, Qute can handle incomplete data structures as a logic language might. Functions are regarded as first class citizens — they can be passed as arguments to other functions or returned as values of expressions. This suggests that higher-order unification may have to take place which is known to be undecidable.

Fresh [103] is akin to Qute; it is a higher-order functional language that incorporates logic language features: unification, non-ground data structures, nondeterminism and negation-as-failure. Logical variables are included in the language as first-class data structures — they can be used to partially define data structures. Non-determinism is introduced by a construct called *disjunction* which allows multiple results to be returned from an expression. The results of a disjunction, `e | f`, are the results of `e` followed by the results of `f`. Orthogonal to this are two constructs for the elimination of multiple results. Horn clauses and predicates can be expressed in Fresh. Predicates become functions that either fail or yield the atom `true` as a result. Such functions have precisely the same semantics as the the corresponding Prolog predicates. Fresh deals with the problem of higher-order unification by hiding functions behind names called 'designators'. By treating designators like atoms unification can cover all objects.

LML[1] [12] is perhaps the best attempt to date to merge functional and logic programming. This is like any typical modern functional programming language, but its distinctive feature is the presence of a data type of theories, whose objects represent logic programs. This amalgamates into a single framework the expressive power of both the functional and logic paradigms. These theories are ordinary data values which can be manipulated by suitable operators. They are denoted as collections of clauses that are defined with reference to the same data types that functions manipulate. Clauses are of the form:

$$A \ :- \ B_1, \ \ldots \ , B_n$$

Here A is an atom $p(t_1, \ \ldots \ , t_n)$ where p is a predicate name and t_i are terms. Each B_i is either a literal or a universal or existential quantification over a conjunction of literals. A literal is a positive atom $q(t_1, \ \ldots \ , t_n)$ or a negative atom $\sim q(t_1, \ \ldots \ , t_n)$.

[1] LML is an acronym for *Logical Meta Language* and should not be confused with Lazy ML

Rather than using negation-as-failure as most logic languages do, LML defines negative information using intensional negation. Here the negation connective (~) applied to an atom `p(t)` is viewed as a particular kind of positive atom with the special predicate name `~p`. The clauses defining `~p` can be systematically derived from the clauses defining its counterpart `p`.

The operators on theories can be used to query them, collect their answers and compose them. A set mechanism is used to get results from theories. For example, using the theory `Peano`:

```
val Peano = ( plus (zero,x,x).
              plus (succ(x),y,succ(z)) :-
              plus (x,y,z).  )
```

a query of the form:

```
{(x,y) | plus (x,y,3) wrt Peano}
```

yields

```
{(0,3), (1,2), (2,1), (3,0)}
```

The intensional operators are `union` and `intersection`.

P union Q is an expression that denotes the theory obtained by putting the clauses of theories P and Q together.

P intersection Q is obtained in the following way:

if $P(t_1, \ldots, t_n)$:– *Body*1 is a clause of `P`
$P(t_1, \ldots, t_n)$:– *Body*2 is a clause of `Q`
θ is the most general unifier of $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$
then $P(t_1, \ldots, t_n)$:– *Body*1, *Body*2 θ is a clause of `P intersection Q`

The `union` and `intersection` operators provide higher-order operations over logical relationships.

The theory datatype of LML provides a clean extension to functional programming and introduces a mechanism that gives the power of expression normally associated with a logic language. However, the logic theories can only be used for computation in the manner of set-valued functions in set abstractions.

Extending functional languages with new constructs that increase expressive power intuitively appears to be a good idea. However, the extensions are only successful if they do not conflict with the underlying mathematical model — the $\lambda - calculus$ — and do not complicate the semantics. Pattern matching is used as the parameter passing mechanism in conventional functional languages and is derived from the beta-rule of the $\lambda - calculus$. Unification is a generalisation of pattern matching and hence of this rule. This implies a change in the mathematical model and semantics. It also restricts a language making it first-order unless function names are used to provide syntactic equality for functions as in Fresh. The introduction of non-determinism by permitting multiple results to be returned from expressions also conflicts with the $\lambda - calculus$ since it allows introduction of non-functional definitions.

2.2.4 Extensions to Logic Languages

One alternative to extending a functional language with logic features is to extend a logic language with functional features. The approaches discussed here incorporate functions, functional notation, strong typing and lazy evaluation into logic programming.

Prolog with rewrite rules is suggested by Newton [79]. Functional notation is added to Prolog in the form of a new database rule which allows the user to specify conditional rewrite rules of the form:

$$a \Rightarrow b :- c$$

This clause states: if a term unifies with a then it can be rewritten to b if the Prolog goal(s) c are true. Such rules are allowed to depend on Prolog clauses in order to permit the user to switch back and forth between functional and relational clauses. This extension of Prolog is designed to be a conservative one. The meaning of Prolog programs is preserved. It merely adds a primitive form of functional notation for term rewriting.

Prolog with equality is advocated by Kornfield [57] — an extension of Prolog that allows the inclusion of assertions about equality. When an attempt is made to unify two terms that do not unify syntactically an equality theorem of the form `equal(s,t)` may be used to attempt to prove the two terms equal and hence allow them to unify. Besides improving the power of Prolog such equality theorems can be used to augment Prolog with functional notation. This allows composition of functions without the need to introduce temporary variables to glue successive relations together.

FUNLOG [108] allows executable functions to be used in Horn clauses and *reduced by need.* A FUNLOG program consists of a set of equations and a set of Horn clauses. The equations act as term rewriting rules. Each Horn clause is of the form:

$$A :- B_1, \ \ldots \ , B_n$$

where a predicate B_i in the clause body can be of the form $M =_E N$ where M, N are terms and $=_E$ means E-unification (i.e. unification using equational theories).

Parameter passing in Horn clauses is by unification. If two terms do not unify syntactically then E-unification is invoked. The equations are applied to the terms as reduction rules in an attempt to make them unify. This will only terminate if the equations defining the functions are noetherian and confluent. This use of unification appears to complicate the semantics of the language.

Eqlog [37] unifies Horn clause logic programming with (equality based) first-order functional programming. It is based on the smallest logic containing these — Horn clause logic with equality. Both functions and predicates are allowed. Functions are computed by reduction and queries to predicates are computed in a Prolog-like fashion with unification and backtracking. According to Goguen, Eqlog extends the logic programming paradigm without sacrificing logical rigour. Besides functional programming it provides strong typing, user definable abstract

types and generic modules. Eqlog appears to be very large and complex and as far as the author knows has never been implemented.

TABLOG [70, 71] is another language based on first-order predicate logic with equality. Rather than having resolution as its proof system it is based on the Manna-Waldinger [72] deductive tableau proof system. This is a generalisation of the resolution rule of inference to non-clausal logic. This generalisation, however, sacrifices completeness. A TABLOG program is a list of assertions in quantifier free first-order logic with equality that allows a mixing of logical and functional styles of programming. Two example definitions for deletion of an element from a list are:

```
delete (x,[]) = []
delete (x,y:u) = (if x = y then  delete (x,u)
                           else  y : delete (x,u)).

delete (x,x:u) = delete (x,u).
x ≠ y → delete (x,y:u) = y : delete (x,u).
```

The use of unification as a parameter passing mechanism makes TABLOG inherently first-order.

The conservative approaches of Newton and Kornfield add only a little functional notation to logic programming and as such give a limited increase in expressive power. The more ambitious approaches of Goguen and Malachi are more credible from the perspective of merging functional and logic paradigms. However, both Eqlog and Tablog are very large and complex and inherently first-order. The underlying theorem proving engine of Tablog lacks completeness — a problem that results from the generalisation of logic programming to non-clausal logic. This means it is possible to write Tablog programs that will not execute.

2.2.5 Alternative Declarative Paradigms

Researchers have tried to merge functional and logic programming via use of new, alternative, declarative language paradigms. The idea behind this is that functional and logic programming may be seen as two separate aspects of a different paradigm.

Equational Languages

Equational programming languages were first presented by Hoffman and O'Donnell [41]. Equations can be used to program all the computable functions and provide a convenient notation for programming. Expressions in an equational language, as in a functional language, can be evaluated by reduction. However, reduction can only be guaranteed to lead to a normal form if the equations are confluent and noetherian.

Dershowitz and Plaisted [32] advocate **condition directed** equations. For functional programming equations are used as pattern directed rewrite rules. Each returns as output a simplified term equal to its given input term. Given

an input expression rewrite rules may be applied in any order. If a subterm of the given expression pattern matches the left hand side of an equation then that subterm may be replaced by the right hand side of that equation. For logic programming, rules are used to solve goals by a process called *narrowing*. If the left hand side of a rule unifies with any subterm of a goal, then the goal is narrowed by applying the narrowing substitution to the goal and then applying the rule to rewrite the subterm.

Unicorn [5] introduces a language mechanism called *constraining-unification*. Given a particular input expression and a set of axioms, constraining unification produces a set of *transformation constraints* that any allowable transformation of the input expression must satisfy. Each axiom is viewed as producing a constraint on the operations rather than on the variables used in the axiom. For example the axiom

```
centigrade (X) = plus (32,times (9,divide (X,5)))
```

states the desired relationship between quantities of `centigrade` and quantities of `fahrenheit`. It can be used as a rewrite rule for converting from centigrade to fahrenheit or to convert from fahrenheit to centigrade. Constraining-unification is a generalisation of the mechanisms used in logic and rewrite rule programming. Equations here provide functional style rewrite rules. They must be first-order since constraining unification may be applied. Constraint unification gives a logic programming mechanism for solving equations but no rule of inference is involved.

Jayaramann and Silbermann [51] propose **rewrite rules and equations** as the unifying language constructs for first-order functional and Horn clause logic programming. A program in this framework is a set of rewrite rules and equations that need to be solved. The rewrite rules are an extension of those found in non-canonical term rewriting systems in that they permit conditions, expressed as a set of equations having logical variables, on the right hand side of a rule. The operational semantics are unified by outermost reduction, as used in functional languages, and object refinement, which solves an equation by progressively reducing its two expressions and by refining objects bound to its logical variables. This contrasts with the approaches that juxtapose functions and predicate clauses resulting in two radically different programming styles. Rewrite rules can be used to express functions directly and, by extending them with conditions, to express Horn logic relations. These are identical to the conditional rewrite rules advocated by Dershowitz [32].

Constraint Functional Programming

Darlington's idea of absolute set abstraction has recently been developed into a new paradigm called **constraint functional programming** (CFP) [28]. Here constraints are associated with function definitions. In a CFP system functional programming is achieved by evaluating expressions on the computational domains. At the same time logic programming is achieved by solving constraints over these domains. A constraint is a declarative statement of the relationships

between objects in the domain, and is also a computational device enforcing these relationships. Thus functional and logic programming are integrated with a uniform semantic base. The satisfiability of constraints can be computed by some built-in solver together with a general goal reduction procedure. The expressive power of general logical inference and non-deterministic computation comes from the integrated system with constraint solving. For example, a CFP function for generating all the permutations of a list:

```
permu @ [a] -> set [a]
permu [] -> {[]}
permu (a:l) -> {u::(a:v) with u,v | u::v∈permu l}
```

2.2.6 Comparison of the Integration Approaches

The dual interpreters approach discussed in section 2.2.1 does not integrate the aspects of functional and logic programming into a single language. Rather two separate languages with some communicating interface is required.

The combining of functional and logic features into a single language, as discussed in section 2.2.2, presents similar problems. Despite the features being in the same language some communication is required because different evaluation mechanisms are used for different parts of a program. There is a direct conflict of semantics between the logic and functional parts.

The approaches extending functional or logic programming discussed in section 2.2.3 and section 2.2.4 appear more sensible. However, finding an extension that is simultaneously powerful and implementable is difficult. Some of the suggested languages are very large and complex. Also changing the parameter passing mechanism in functional languages from pattern matching to unification eliminates the possibility of higher-order definitions.

The idea of a new declarative paradigm, one which naturally incorporates aspects of functional and logic programming would seem preferable. The paradigms discussed in section 2.2.5 however do not match the criteria advocated for a successful merge in chapter 1.

2.3 Relational Database Query Languages

One of the biggest uses of relations to date has been in the field of relational databases and their query languages. Relational query languages are therefore of interest to any designer of a relational programming language.

2.3.1 Functional Query Languages

FQL [16] is based on a functional programming system similar to Backus [3]. Instead of explicit control structures a few operators or functional forms are used to construct new functions or database queries from existing functions. Consider a very simple query:

```
!EMPLOYEE o *NAME
```

Informally this reads "take a the sequence of all employees and create a sequence of their names". The function !EMPLOYEE generates a sequence of employees. The function NAME takes an EMPLOYEE as argument and returns a character string as result. The operator * takes a function that acts on some type of entity and makes it act on a sequence of that entities of that type. Here NAME acts on EMPLOYEE, *NAME acts on a sequence of EMPLOYEES. The two functions are composed by the composition operator 'o'.

One query optimisation used by FQL is *memoisation*. This prevents a function from being recomputed by storing result values as they are computed. This changes the representation of the function from the intensional to the extensional. Also only extensionally represented functions can be stored in the database. This means that the FQL system must cope with both extensionally and intensionally represented functions.

The functional query language **FDL** [84] extends the functional database model to computational completeness while also supporting the persistence of any function whether extensionally or intensionally defined. All functions whether used for data modelling purposes or for computation, are treated uniformly with respect to their definition, evaluation, update and persistence. Also functions can be partly extensionally defined and partly intensionally defined.

As we shall see later these systems are related to relational programming systems that must process both extensionally and intensionally represented relations. Moreover all relations should ideally be treated uniformly, irrespective of their representation as they are in FDL.

2.3.2 Calculus-based Query Languages

The concept of basing query languages on relational calculus was first suggested by Codd [22]. The query language QUEL was one of the earliest implementations of Codd's proposal and is very close to his proposed predicate calculus notation.

QUEL uses the notion of a *Tuple Variable*, which can be instantiated to reference any tuple in a given relation. For this reason it is called a *tuple-calculus* language. The attribute values for a given tuple are obtained from the tuple variable by using a record selector notation similar to that of Pascal. For example if variable S ranges over an employee relation then *name* and *salary* attributes would be denoted by S.Name and S.Salary.

There are four commands in QUEL: RETRIEVE, REPLACE, DELETE and APPEND. In general queries in QUEL take the form:

```
RANGE OF var IS rel
{ RANGE OF var IS rel }
RETRIEVE [INTO rel]
(rel.var {, rel.var})
WHERE pred
```

Here braces denote repetition and square brackets enclose optional items. An example query is:

```
RANGE OF S IS EMPLOYEE
RETRIEVE S.Salary
WHERE S.Name = 'Smith'
```

The query language SQL (Structured Query Language) may be regarded as an extension of QUEL with facilities for handling sets. Simple queries in SQL are like those in QUEL but it is also possible to write more complex queries which involve the formation of intermediate sets. Set union and set difference operations can be used on these sets which gives some of the features of relational algebra. The syntax of SQL queries is of the form:

SELECT [UNIQUE] *varlist* FROM *rel_list* WHERE *pred*

Here *varlist* is one or more attribute values, which are taken from tuples referenced by tuple variables ranging over the relations in *rel_list*. The tuples are selected by the predicate in *pred*.

SQL allows use of *nested sub-queries* which effectively generate unnamed relations. The operators that may be applied to a sub-query S are based on those which one could apply to a set. For example set union (S1 UNION S) or set membership (X IN S).

An example query in a football database is given by Gray [39]:

"All groups that included Scotland or played some matches in Rosario"

This could be written in SQL as:

```
(SELECT UNIQUE Group
 FROM GROUP_PLAC
 WHERE Team = Scotland
 UNION
(SELECT UNIQUE Group
 FROM STAD_ALLOC
 WHERE Stadium = 'Rosario')
```

2.3.3 Algebra-based Query Languages

The relational algebra used in database query languages is an equivalent notation to the relational calculus. It is based on function application and the evaluation of algebraic expressions.

The basic operations of relational algebra were first suggested by Codd [22]. They are described concisely by Date [29]:

SELECT: Extracts specified tuples from a specified relation.

PROJECT: Extracts specified attributes from a specified relation.

PRODUCT: Builds a relation from two specified relations consisting of all possible concatenated pairs of tuples, one from each of the two specified relations.

UNION: Builds a relation consisting of all tuples appearing in either or both of two specified relations.

INTERSECT: Builds a relation consisting of all tuples appearing in both of two specified relations.

DIFFERENCE: Builds a relation consisting of all tuples appearing in the first but not the second of two specified relations.

JOIN: Builds a relation from two specified relations consisting of all possible concatenated pairs of tuples, one from each of the two specified relations, such that in each pair the tuples satisfy some specified condition.

DIVIDE: Takes two relations, one binary and one unary, and builds a relation consisting of all values of one attribute of the binary relation that match (in the other attribute) all values in the unary relation.

These operations are discussed in more detail in chapter 3.

The output of each of these algebraic operations is another relation — that is the operations form a relation algebra that is closed. As a result of the closure property relation expressions may be nested — that is expressions in which the operands are themselves represented by expressions instead of just names.

It should be understood that the eight operations do not constitute a minimal set, nor were they ever intended to. In fact of the eight, only five are primitive: restriction, projection, product, union, and difference. The other three — intersect, join and divide — can be defined in terms of those five. For example, the natural join is a projection of a restriction of a product as explained by Date [29].

Gray [39] introduces a query language, called ASTRID, which uses these relational algebra operations. We present an example query for a world cup football database. The query is to list all groups in which Scotland played in 1978:

```
GROUP_PLAC
selected_on[Year = ''1978'' and Team = ''Scotland'']
projected_to GROUP
```

2.3.4 Comparison of Algebra and Calculus

Equivalence of Calculus and Algebra

Codd [23] proved that the algebra is at least as powerful as the calculus. He did this by giving an algorithm — *Codd's reduction algorithm* — by which an arbitrary calculus expression can be reduced to a semantically equivalent expression of the algebra. Conversely, it is possible to show that any algebraic expression can be reduced to a calculus equivalent, and hence the calculus is at least as powerful as the algebra. It therefore follows that the two formalisms — calculus and algebra — are logically equivalent.

A language is said to be *relationally complete* if it is at least as powerful as the relational calculus — that is if its expressions permit the definition of any relation definable by the expressions of the relational calculus.

Relational completeness may be regarded as a basic requirement of expressive or selective power for database languages in general. However, relational completeness does not necessarily imply any other kind of completeness. For example, it is also desirable for a language to provide *computational completeness.* i.e. support all of the computational operators found in arithmetic. The calculus and algebra defined in this chapter are not complete in this sense.

Advantages and Disadvantages of the Formalisms

Given the two formalism are equivalent in power which is best to use?

Gray [39] states that the principal advantage of the algebra is that it is closed under the relational operations. In calculus-based languages certain difficult queries have to be formulated by asking sub-queries and storing the results. The extra constructs needed to formulate these results are to some extent 'outside' the calculus. However, the algebra does not need these extra constructs since the notion of using intermediate relations is already there, and thus queries of arbitrary complexity can be built up. The crucial thing is that the user can give each intermediate result a name to remember it by if he chooses; by contrast, calculus notation tends to produce complex nested expressions that are unnamed, and it is thus harder to read and understand.

Advocates of the calculus usually claim it is less 'procedural' than relational algebra, because it describes the result in terms of a collection of predicates whilst the algebra gives a succession of operations to be applied to give the desired result. However the relational algebra has the property of referential transparency which allows one to make substitutions and to rewrite the operations in many equivalent forms; thus the description is more flexible than it looks. In practice when transforming queries, it seems just as easy, if not easier, to work with algebraic rather than calculus notation.

A Computationally Complete Algebra

The above description of the good points of relational algebra by Gray encourages the development of a programming language based on relational algebra. Perhaps more importantly the equivalence of calculus and algebra suggests that a programming language based on algebra could be of similar expressive power to a language based on relational calculus. This bodes well for the notion of encompassing logic programming within relational programming since logic programming is typically based on relational calculus.

The concept of relational completeness is of importance. A relational language should, at the very least, have analogues of the five primitive algebraic operations. Ideally, if the algebra is to be rich in terms of expressive power, it should have direct analogues of all eight. Of course computational completeness must be satisfied as well.

2.4 Related Work on Relational Systems

2.4.1 Popplestone's Relational Programming

Popplestone's [83] idea for relational programming is more of a logic programming language than a true relational one — the language computation mechanism applies forward inference to predicate calculus. Conclusions are inferred from premises rather than Prolog-style backward inference which starts with a conclusion and tries to find ways of inferring it. Rather than a 'backtrack search' this approach uses operations on tables of data and is identical in many respects to the work of Codd on databases. From one perspective this work is an extension of Codd's into the realm of general purpose computing. For example given the relation ON

```
ON = {(1,2) (2,3) (3,5)}
```

and the relation ABOVE defined by the clauses

```
ON (x,y)  =>  ABOVE (x,y)
ON (x,y) & ABOVE (y,z)  =>  ABOVE (x,z)
```

It is possible to find a value for ABOVE by cycling round the two clauses. From the first clause we derive:

```
ABOVE ⊇ {(1,2) (2,3) (3,5)}
```

Then using second clause produce new rows:

```
ABOVE ⊇ {(1,2) (2,3) (3,5) (1,3) (2,5)}
```

Further rows can be generated by using the second clause again, and no more rows can be added by further use of the clauses, so a fixed point has been reached.

```
ABOVE = {(1,2) (2,3) (3,5) (1,3) (2,5) (1,5)}
```

The main criticism of this work is that Popplestone makes no mention of higher-order relations or general relational operators for manipulating and composing relations. To the author's knowledge Popplestone's work has not been implemented.

2.4.2 Wile's Relational Data Structures

Wile [118, 119] argues the case for incorporating relational access in programming languages generally. He has implemented a set of macros in Lisp to allow definition, update and queries of abstract relations. Rather than being the repositories of 'bulk data' these relations are 'lightweight' being used as a common abstraction of a wide variety of conventional program data structures.

The relations used are general n-ary relations and a relation's schema is specified by giving name and the types of its arguments. For example:

```
defrelation grade-for (student course grade)
```

Facts can be asserted, retracted and updated:

```
assert (grade-for 'John-Jones 'cs101 'A)
retract (grade-for 'John-Jones 'cs101 'A)
update (grade-for 'John-Jones 'cs101 'A-)
```

Here a single slot (field of the relation) is identified for change based on the declaration of the key slots — here grade is changed from A to A-.

The true power of the relational approach comes from the retrieval language, where access through any fields is permitted. The selection ? refers to any element satisfying the relation. The symbol ?? refers to all elements satisfying the relation. For example:

```
(grade-for 'John-Jones ?? 'A-)
(grade-for 'John-Jones 'cs101 ?)
```

The first query selects all the courses in which John Jones received an A-; the second query selects John Jones grade for course cs101.

The system implementation has a large set of operators for manipulating and retrieving relationships. In addition to simple assertions and retractions several forms iterate over sets of items, inserting, retracting or changing tuples containing them. This language extension mechanism was designed to incorporate relational data structures, rather than to permit a relational style of programming as a whole.

2.4.3 The Relational Language GREL

The work by Legrand [62] is slightly closer to our idea of relational programming. That is relations are used in the same way as functions, but he makes the observation that relations are more general than functions because they allow us to consider point-to-set computations as well as point-to-point computations.

However Legrand uses n-ary relations rather than binary relations. An n-ary function f may be defined as an $(n+1) - ary$ 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. He uses set-valued functions to represent relations. The result of applying such a function is a stream of values that are returned one at a time to the calling relation/function. Legrand observes that relations are useful in the design of programs that are non-deterministic or multi-valued. When application of the function f to arguments $a_1, \ldots, a_n$ gives a multi-set of results $b_1, b_2, \ldots$ the function evaluation is denoted by

$$(f : a_1, \ldots, a_n) \rightarrow b_1\ b_2\ b_3$$

Legrand has implemented his ideas as the relational language GREL which is the development of a functional language, GRAAL, towards relations. Function evaluation in GREL remains the same as in GRAAL but special forms and functions are added to build non-functional relations. The most important of these is the `union` construct for applying a number of functions to the given arguments and collecting their multiple results:

$$(union\ f_1\ \ldots\ f_n)\ :\ a_1\ \ldots\ a_x\ \rightarrow\ \begin{array}{llll} y_{1,1} & \cdots & y_{1,p_1} \\ y_{2,1} & \cdots & y_{2,p_2} \\ & \cdots & \\ y_{n,1} & \cdots & y_{n,p_n} \end{array}$$

where $f_i\ :\ a_1,\ \ldots\ ,a_x\ \rightarrow\ y_{i,1}\ \cdots\ y_{i,p_i}$

One of the most important applications of GREL, Legrand says, is for programming in logic. A relation appears as the procedure corresponding to the translation of a Horn clause of logic.

Legrand compromises relational abstraction by using lists as the basic structure and by fixing relation representations as set-valued functions. As a result relations must always be used in a directional manner and it is the programmer's responsibility to collect the alternative results. Ideally the programmer should be permitted to reason about relationships and leave processing of multiple results to the system.

2.4.4 The Relational Algebra of Möller

Möller [76, 77] presents a relational language which he designed specifically to have a number of useful algebraic properties. The language is based on n-ary heterogeneous relations.

Tuples

Each element of a relation is a tuple denoted as a sequence of components. The empty tuple is denoted with ε and tuple concatenation by $\bullet$. Tuple concatenation is associative with ε as the neutral element:

$$u \bullet (v \bullet w) = (u \bullet v) \bullet w$$
$$\varepsilon \bullet u = u \bullet \varepsilon = u$$

All tuples are considered to be flat, i.e. nesting of tuples is irrelevant. In particular the singleton tuple is not distinguished from the value it contains:

$$(u) = u$$

Concatenation is lifted to sets of tuples as well:

$$U \bullet V = \{u \bullet v : u \in U, v \in V\}$$

If U and V consist of singleton tuples only then $U \bullet V$ corresponds to the cartesian product $U \times V$. Concatenation of sets of tuples is associative with ε as the neutral and $\emptyset$ as the zero element.

$$U \bullet (V \bullet W) = (U \bullet V) \bullet W$$
$$\varepsilon \bullet U = U \bullet \varepsilon = U$$
$$\emptyset \bullet U = U \bullet \emptyset = \emptyset$$

The reverse of a tuple u = $x_1 \bullet \ldots \bullet x_n$ is a the tuple $u^{-1} = x_n \bullet \ldots \bullet x_1$ The reversal operation is extended pointwise to sets of tuples:

$$U^{-1} = \{u^{-1} : u \in U\}$$

Relations

A *relation* is a subset $R \subseteq Q_1 \bullet \ldots \bullet Q_n$ of the cartesian product of certain sets $Q_1, \ldots, Q_n$. The *type* of R is $Q_1 \bullet \ldots \bullet Q_n$, the *domain* (dom R) is Q_1 and the *codomain* (cod R) is Q_n. The *arity* of R is the length of the tuples in R: ar R = n. There are only two 0-ary (nullary) relations viz. ε and $\emptyset$. A nullary relation R may serve as a pre- or postcondition for another relation S, with ε representing truth and $\emptyset$ representing falsehood.

$$R \bullet S = S \bullet R = \begin{cases} S & if\ R = \varepsilon \\ \emptyset & if\ R \neq \varepsilon \end{cases}$$

Operations Over Relations

The *converse* (or *inverse*) of a relation R is denoted by R^{-1} by virtue of the fact that tuple reversal is extended to sets of tuples.

The *relation composition* of two relations R, S with $ar\ R$, $ar\ S > 0$ and cod R = dom S = Q is defined by

$$R ; S = \bigcup_{x \in Q} \{u \bullet v : u \bullet x \in R \wedge x \bullet v \in S\}$$

The *diagonal* of a set P ⊆ Q is:

$$I_P = \{x \bullet x : x \in P\} = \bigcup_{x \in P} x \bullet x \subseteq P \bullet P$$

The diagonal of a domain and codomain of a relation are left and right identities respectively whereas $\emptyset$ is a zero element.

$$I_{domR} ; R = R = R ; I_{codR}$$
$$\emptyset ; R = \emptyset = R ; \emptyset$$

Möller shows that interesting special cases of composition arise when one of the relations has arity 1. Suppose R ⊆ dom S = Q for some S with arity ar S > 0

$$\begin{aligned} R\,;\,S &= \bigcup_{x \in Q} \{u \bullet v : u \bullet x \in R \wedge x \bullet v \in S\} \\ &= \bigcup_{x \in Q} \{\varepsilon \bullet v : \varepsilon \bullet x \in R \wedge x \bullet v \in S\} \\ &= \bigcup_{x \in Q} \{v : x \in R \wedge x \bullet v \in S\} \\ &= \bigcup_{x \in R} \{v : x \bullet v \in S\} \end{aligned}$$

In other words, R ; S is the image of set R under relation S. Likewise, for T $\subseteq$ cod S, the set S ; T is the inverse image of T under S.

Möller defines the natural join used in relational databases theory: given two relations R, S with cod R = dom S = Q, their *join* R $\bowtie$ S, consists of all tuples that arise from 'gluing' tuples from R that end in with a certain element to tuples from S that start with the same element. The set of tuples in R ending with x is given by R ; x • x, while the set of tuples in S starting with x is given by x • x ; S.

$$R \bowtie S = \bigcup_{x \in Q} R\,;\, x \bullet x \bullet x\,;\, S$$

This is closely related to composition. Whereas R ; S just states whether there is a path from a point x to a point y via some point z $\in$ Q, the relation R $\bowtie$ S consists of exactly those paths x • z • y.

Interesting special cases arise when one of the relations involved in a join has arity 1. Suppose R $\subseteq$ dom S for some S with $ar\ S > 0$ and dom S = Q:

$$R\,;\,S = \begin{cases} \varepsilon & x \in R \\ \emptyset & x \notin R \end{cases}$$

and hence

$$\begin{aligned} R \bowtie S &= \bigcup_{x \in Q} R\,;\, x \bullet x \bullet x\,;\, S \\ &= \bigcup_{x \in R} \varepsilon \bullet x \bullet x\,;\, S \\ &= \bigcup_{x \in R} x \bullet x\,;\, S \end{aligned}$$

In other words, R $\bowtie$ S is the restriction of relation S to set R. Likewise, for T $\subseteq$ cod S, the set S $\bowtie$ T is the corestriction of S to T.

Assessment

Möller's principal interest is in the algebra of relations and he demonstrates a number of interesting properties for a beautifully simple view of relations. It would be desirable for any implemented relational programming language to have such properties. There would appear to be no reason why some at least

should not be retained although some may be impossible due to consideration of computability. For example, it will not be possible to pattern match tuple elements of a relation if that relation is represented by a function. Möller has not yet developed any implementation of his system.

2.4.5 The Relational Language Ruby

Ruby [101] is a language, based on binary relations, designed by Mary Sheeran for use in describing hardware algorithms and circuits. A Ruby circuit description is a binary relation between signals. For example

```
a R b
```

relation R relates domain signal a to range signal b. A signal is a data value that is either atomic or a tuple or a list of signals. Relations are useful for modelling the components of circuits: they are the simplest interpretation of what is happening.

Ruby is an example of an *constructive* language. That is programs are built up piecewise from smaller programs. Higher-order functions called *combining forms* or *combinators* for short are used for this construction. Programming in the constructive style is a great aid to formal manipulation — something which Ruby takes advantage of.

Data Structures in Ruby

The two forms of structuring data in Ruby are lists and tuples. Tuples can be of any length and may be nested. Some examples of tuples are:

```
(a), (a,b), (a,(b,c))
```

Lists are denoted by enclosing the items in angled brackets. The basic operation over lists is append (^). The following are examples of lists:

<>	the empty list
<x>	the singleton list containing element x
$\langle x_1, x_2, \ldots, x_n \rangle$	a list containing the elements x_i
<x>^xs	a list with head x and tail xs

The distinction between lists and tuples is blurred. For example there is no difference between a triple of things that have a common type and a list of things that have that type and which happens to be three elements long.

Relational Operators in Ruby

Circuit descriptions in Ruby are built up hierarchically using higher-order functions. These functions are operators over relations and hence form a relational algebra. It is left to the circuit designer to decide which operators to define but the operators typically used are composition (;), inverse ($^{-1}$), parallel composition ([,]), union (+) and conjugate (\). These operators are defined:

$$
\begin{array}{lcl}
x\ (R\ ;\ S)\ y & \Leftrightarrow & \exists\ z.\ x\ R\ z\ \wedge\ z\ S\ y \\
x\ (R^{-1})\ y & \Leftrightarrow & y\ R\ x \\
(a,b)\ [R,S]\ (c,d) & \Leftrightarrow & a\ R\ c\ \wedge\ b\ S\ d \\
a\ (R\ +\ S)\ b & \Leftrightarrow & a\ R\ b\ \vee\ a\ S\ b \\
R \setminus S & = & S^{-1}\ ;\ R\ ;\ S
\end{array}
$$

The relational algebra is closed under these operators since each operator application forms a new relation.

Ruby is associated with a collection of transformation rules that are based on algebraic laws over relational expressions. An initial circuit description that is inefficient or even impossible to implement can be transformed using these rules.

Sheeran [101] uses Ruby to specifying hardware algorithms for butterfly networks of chips. Jones and Sheeran [53] again study butterfly circuits but derive simpler, more elegant algorithms. In [52] they give a detailed description of Ruby and its use for describing and refining circuit descriptions.

Rossen [91] describes a framework intended to allow the system designer to capture a circuit description as a relation and, through stepwise refinement, construct a circuit suitable for automatic layout generation. This framework is based on a theorem prover that can be used to prove the correctness of equivalences used in the system process. He gives a formal definition of Ruby to implement this system support. Rossen [90] gives an example synthesis of a circuit from a specification using this theorem prover. This construction gives a formal proof of correctness of the circuit with respect to its specification.

Hutton [49] observes that while programming in a relational framework has much to offer over the functional style in terms of expressiveness, computing with relations is less efficient and more semantically troublesome. Computation is less efficient because relations have no notion of data-flow. Relations are not as well-behaved semantically as functions. For example, the fixed-point approach to recursion does naturally extend to the relational world.

He proposes a blend of functional and relational styles by identifying *causal relations* which retain the bi-directional properties of relations but retain the efficiency and semantic foundation of the functional style. A relation is *causal* if it is possible to identify an 'input' part of the relation which uniquely determines the 'output' part. Unlike functions however the input part of a causal relation is not restricted to its domain, nor its output to the range; indeed inputs and outputs may be interleaved throughout the domain and range. A causal relation may have many such functional interpretations.

This weakening of the functional constraint permits bi-directional communication between components. The standard relation composition operator (;) can be used to combine any two components regardless of whether they communicate bi-directionally or not. It was this observation that originally led Sheeran to consider using relations rather than functions.

However there are two problems with causal relations:

- Causal relations are not closed under composition — composition of two causal relations may produce a relation that is not causal.

- There are causal programs that involve non-functional data flow. For example (and^{-1} ; and) is equivalent to the identity relation over Booleans but operationally has non-functional flow between the two primitives.

An Assessment of Ruby

Ruby illustrates how relations may be used in a specification language. The work by Rossen shows that such a relational language can be implemented. However 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, rather than relations, are used as the basic data structure.

The operators used in Ruby appear to be powerful and widely applicable to relations. This implies that they would be of use in any relational programming language.

The emphasis on algebraic laws for derivation of efficient algorithms is an interesting idea and one that can be used for a full programming language.

2.4.6 The Relational Language RPL

The work by MacLennan on relational programming is pioneering. He developed a programming language, RPL, from a relational algebra and designed a prototype interpreter for it. His work formed a basis for the research described in this thesis.

Origin and Development of RPL

MacLennan initially presented his concept of a programming language based entirely on relations in the papers [65, 66] and later produced a more detailed internal report [67]. This theoretical concept he developed into a working programming language called RPL for which some of his students built an interpreter [13][2]. From his experience with RPL MacLennan presented four sample relational programs in a report [68] which he later refined into a paper [69].

MacLennan had a view of relational programming which appears unique in the literature. It was to be 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, which are themselves relations, are available for manipulating both data and programs;
- a paradigm that subsumes functional programming.

[2]This author, however, has been unable to secure copies of either this thesis or the interpreter.

It should be possible to express clearly in a relational language anything that can be expressed clearly in a functional language. Furthermore, some things that are awkward to express functionally may be naturally expressed relationally because mathematically the concept of a relation is more general than that of a function — all functions are relations but not all relations are functions. The evolution of functional programming into relational programming is a generalisation that increases expressive power.

There should be a number of similarities between functional and relational programming. Functions and relations have both been studied and developed in mathematics and functional and relational programs can both be derived and processed via algebraic manipulation.

MacLennan first published work on relational programming in 1981. Around this time Backus FP [3] was just emerging on the scene in functional programming and was proving to be very influential. The number of papers on FP published in the same journals as MacLennan's work testifies to this. Backus' view of functional appears to have been the one that most influenced MacLennan. MacLennan was really attempting to generalise FP, and its treatment of functions, to relations. The operators MacLennan uses are combining forms for relations akin to the combining forms for functions in FP.

The Underlying Mathematical Model

This section describes the mathematical model that underlies RPL and MacLennan's mathematical view of relations in general. It is based on the earlier MacLennan papers [65, 66] in which he presented the mathematical basis of relational programming, prior to the implementation of RPL.

Sets are the most fundamental objects. A set is an unordered collection of items that contains no repetitions. An n-ary relation may be perceived as a set of n-tuples where each tuple is an element of the relation. Thus if R is a n-ary relation then

$$R(x_1, x_2, \ldots, x_n) \Leftrightarrow (x_1, x_2, \ldots, x_n) \in R$$

MacLennan restricts his view of relational programming to binary relations. Any binary relation, R, may be regarded as a set of pairs:

$$x\, R\, y \Leftrightarrow (x, y) \in R$$

More general n-ary relations may be modelled by allowing x or y themselves to be pairs. For example, arithmetic plus may, be thought of as a binary relation defined:

$$(x, y)\,(+)\, z \Leftrightarrow z = x + y$$

A relation is considered to be a function if there is only one output value associated with any given input value. For a relation expression x R y where R is the relation, x is considered to be the input and y the output. Such a relation is generally referred to in computer science as being deterministic. MacLennan refers to a relation having this functional property as being right univalent. The formal mathematical definition of right univalence given by MacLennan [65] is :

$$F \in \text{run} \Leftrightarrow \forall x\, y\, z\, [x\, F\, y \wedge x\, F\, z \Rightarrow y = z]$$

The set of all right univalent relations (functions) is a subset of the set of all relations.

Relational operators are a special case of set operators because a relation is a set of pairs. For example the union of two sets:

$$x \in (S \cup T) \Leftrightarrow x \in S \vee x \in T$$

The union of two binary relations is a special case in which x is a pair `(y,z)`:

$$\begin{aligned}(y,z) \in (S \cup T) &\Leftrightarrow (y,z) \in S \vee (y,z) \in T \\ &\Leftrightarrow y\, S\, z \vee y\, T\, z \\ &\Leftrightarrow y\, (S \cup T)\, z\end{aligned}$$

The Universe of Values

Although MacLennan refers to his theory of of relations as being typeless it is possible to identify an universe of typed values in which all RPL values lie.

There are three primitive types in this universe, or domain: real numbers, Booleans and characters. More complex types are constructed from pairs, sets and intensional relations over the domain. These take the form :

$$\text{pair} = (\text{domain},\text{domain})$$
$$\text{set} = \{\text{domain}\}$$
$$\text{intensional relation} = (\text{domain} \rightarrow \text{domain})$$

Thus the whole domain may be defined as:

`domain ::=`	`Bool`	boolean values
	`Numbers`	numerical values
	`Characters`	ASCII characters
	`(domain,domain)`	pairs of values
	`{domain}`	sets of values
	`(domain → domain)`	functional relations

Representing Relations

For the implementation of relational programming some representation is needed for each relation defined in the program.

There are two views of relation representation: the programmer's view and the implementation view. In RPL these two are one and the same; the method of representation, for any relation, *must* be known to the programmer. This is because the choice of representation for a particular relation is the sole determiner of which relational operators can be applied to it and which cannot.

In RPL there are two main different ways in which relations can be represented:

Extensional representation: the relation elements are explicitly stored in a list data structure. Such relations are the data the program manipulates.

Intensional representation: the relation elements are not stored, rather the relation is represented by the corresponding computable function. This though forces all intensional relations to be functional.

The Relational Operators

MacLennan defined numerous relational operators for RPL. Those operators are divided into two groups:

Extensional operators can manipulate relations that are extensionally represented. These are finite, extensional sets and relations — those sets and relations whose elements are explicitly stated.

Intensional operators can manipulate relations that are intensionally represented. They are combining forms for computable functions.

An Assessment of RPL

MacLennan corrupted the mathematical view of relations in his implementation of RPL, with the divide between intensional and extensional relations.

The extensional operators are functions for manipulating two kinds of data structure: one representing sets and one representing sets of pairs. They preserve the view of relations as a sets of pairs but are not applicable to intensional relations. This implies the intensional relations can no longer be thought of as sets. The intensional operators defined are really just combinators — combining forms for ordinary computable functions. They are not applicable to extensional relations and hence are not general relational operators.

A program in RPL is a collection of definitions each of which is an intensional relation represented by a computable function. Since functions are deterministic non-determinism is not naturally handled. The only operators applicable to program code are the intensional of which there are comparatively few. Most operators are extensional and hence only applicable to the data of the program. 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 a form of relational data structure.

If relational programming is to succeed as a new declarative paradigm then it should be implemented in more generality than RPL and brought closer to MacLennan's original conception.

2.5 Conclusion

Section 2.2 concluded that if functional and logic programming are to be successfully merged then some new declarative paradigm is required.

The work on relational query languages discused in section 2.3 shows that it is possible to use relations for high-level computation. The work on algebra-based query languages has highlighted a number of relational operators which are known to form an algebra as powerful as Codd's relational calculus. Moreover, the arguments for relational algebra given by Gray [39] suggest that algebra forms a rich basis for languages.

Section 2.4 described research into relational languages which, although still in its infancy, is developing in a variety of directions. The relational data structures of Wiles, the set-valued functions of Legrand, the emphasis of algebra by Möller and in Ruby, and on generality of relational programming by MacLennan. These ideas have the potential to form a powerful new declarative paradigm but techniques more sophisticated than those used to implement GREL or RPL are required.

Chapter 3

The Relational Language Drusilla

3.1 Introduction

Chapter 2 concluded that more sophisticated techniques are needed to realise relational programming as MacLennan originally perceived it. The relational language used and its new implementation should satisfy certain criteria:

- The language should be based on a mathematical model that can describe both program and data.
- The relational operators used should be general purpose in nature and equally applicable to program code (intensional relations) and data (extensional relations).
- It should be possible to combine extensionally and intensionally represented relations within expressions.

This chapter proposes a new relational language, named Drusilla, which will be used to investigate implementation techniques and possible relational programming styles.

In section 3.2 we define a typed relational calculus which forms the underlying mathematical model of Drusilla. In section 3.3 this model is used to define the primitive relational operators of Drusilla. In section 3.4 we give an overview of Drusilla and present some example programs.

3.2 The Underlying Mathematical Model

3.2.1 Mathematical Preliminaries

It is useful to begin by defining some basic concepts that are fundamental to our discussion of relational programming. Some of these definitions are taken from Gallier's book [36].

Tuples

Given two sets A and B (possibly empty) their *cartesian product*, A × B, is the set of ordered pairs:

$$\{(x,y) \mid x \in A,\ b \in B\}$$

These sets A and B may themselves be cartesian products of other sets — the elements within pairs may themselves be pairs. This concept may be generalised — an *n-tuple* (or *tuple* for short) is formed from the cartesian products of n sets. For example, A = $A_1 \times \ldots \times A_n$, is the set of ordered n-tuples:

$$\{(a_1, \ldots, a_n) \mid a_i \in A_i,\ 1 \leq i \leq n\}$$

Again these sets may be constructed from tuples of other sets and hence nested tuples may be created. There is no distinction made between between a one-tuple and the element it contains.

Binary Relations

A *binary relation*, R, between sets A and B is any subset (possibly empty) of A × B. Given such a relation, the *domain* of R is the set:

$$\{x \in A \mid \exists y \in B, (x,y) \in R\}$$

The *range* of relation R is the set:

$$\{y \in B \mid \exists x \in A, (x,y) \in R\}$$

Sanderson [94] observes that a relation, R, which holds between sets A and B, may be viewed in three different ways:

1. It may be seen as a *set* — a subset of A × B. Adopting this viewpoint enables us to use set-theoretic notation, writing, for example, (x,y) ∈ R or R = S ∪ T (where S and T are also relations).

2. It may be seen as a *logical relation*. Thus we may write x R y as the equivalent of (x,y) ∈ R and use the predicate calculus in defining new relations. The above is read *x has the relation R to y*. This technique is adequate for setting up a complete relational calculus and forms the basis of the Drusilla calculus.

3. When R is viewed as a *relator* we are thinking in terms of applying it to an element of the set A, to produce an element of set B as a result; thus x R y is verbalised as *application of R to x may yield y*. The element pair (x,y) is referred to here as a *map* in R.

A relation R between sets A and B is *functional* (or *deterministic* or *right univalent*) if and only if:

$$\forall x \in A,\ y, z \in B,\ \ (x,y) \in R \wedge (x,z) \in R \Rightarrow y = z$$

The *cardinality* of a set is the number of elements in that set. Therefore the cardinality of a binary relation is the number of element pairs in that relation.

3.2.2 A Typed Relational Calculus For Drusilla

Sanderson [94] states that a relational calculus provides a formalism for writing expressions whose values are relations. Drusilla is based on an extension of Tarksi's [110] calculus of binary relations that introduces types.

Calculus Expressions

The relational operators built into the Drusilla language are themselves binary relations and as such should be treated in the calculus as any other relation. However, it should be possible to use those operators that are known to be functional to construct new relations. Therefore, such operators are attributed both relational and functional forms and may be used to construct expressions similar to Tarski's [110].

The most basic forms of expression are called *relation designations* of which there are two types:

Elementary relation designations are user-defined relations and operators in their relational forms, denoted syntactically by use of square brackets e.g. `[inv]` (relation inverse), `[;]` relation composition.

Compound relation designations are formed from simpler ones by prefix application of unary operators or infix application of binary operators. This is the functional form of operator use from which it is possible to obtain expressions such as `inv r` (the inverse of relation `r`), `r ; s` (the composition of relations `r` and `s`).

A *calculus expression* is created from predicate formulae over terms that are constructed from relation designations. This use of predicate calculus corresponds to Sanderson's relation viewpoint. Calculus expressions will be used to define the Drusilla operators but are not actually part of the Drusilla language. Relation designations form part of both the calculus and the Drusilla language.

Calculus Types

Types are introduced into the calculus to permit Drusilla to be statically, polymorphically typed in the manner of a modern functional language.

Cardelli and Wegner [20] make the observation that types, in the sense of conventional functional languages, correspond to sets of values and Cardelli [19] states that a mathematical model for such types is normally given by mapping every type expression into a set of values (the values having that type). By definition, every binary relation holds between two sets of values. The calculus is typed by regarding such sets as types. The sets of values associated with types are those values Drusilla may manipulate. The types in our calculus, and the sets of values associated with them, are given in table 3.1.

There are three basic calculus types: *num*, *string* and *un*. The type *num* comprises integer and real numbers. The type *string* includes any string of ASCII characters. There is just one value of type *un* called the `Unit` element.

Domain Type	*Corresponding Set of Values*
num	natural numbers $\{0, 1, 2, \ldots\}$
string	strings of ASCII characters {"Drusilla", "1", "afh!#@", ...}
un	unit element {Unit}
$(t_1 \times \ldots \times t_n)$	set of all n-tuples $\{(x_1, \ldots, x_n) \mid x_i \in t_i\}$
$t_1 \leftrightarrow t_2$	set of all relations between types t_1 and t_2
A ... Z	polymorphic type variables — any set of values

Table 3.1: The types in the Drusilla calculus

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 type of `(Unit,"hello",36)` is (un × string × num). Such product domains introduce n-tuples which can be used to make binary relations model n-ary relations. An n-ary relation is, conceptually, a set of n-tuples. If each n-tuple is split into two tuples, the first is 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.

If t_1 and t_2 are types, then $t_1 \leftrightarrow t_2$ is the type of any relation with domain of type t_1 and range of type t_2. By allowing relations to hold between domains that contain relations we permit introduction of higher-order relations into the calculus.

Polymorphism is introduced via type variables (upper case letters), which are understood to be universally quantified. A type variable may optionally be followed by an integer number; this prevents any limitation on the number of type variables permitted within an expression (otherwise polymorphic 27-tuples, for example, would not be allowed!).

All expressions denote a value drawn from one of the above types and hence may be given a *calculus type*. For example the type of arithmetic plus is written:

```
[+] :: (num x num) <-> num
```

Sets in the Calculus

If the relational calculus is to be based solely on binary relations then it should not be concerned with manipulating sets of values as structures. However it should preserve Sanderson's three viewpoints of relations the first of which is that a binary relation is a set of pairs. How can this paradox be resolved? The solution is to introduce the unit domain and identify any set of values with a binary relation from the element type to the unit type. For example, the set of numbers {1,2,3} may be represented by the relation {(1,Unit),(2,Unit),(3,Unit)}. Therefore sets become special cases of binary relations and set theoretic operators (e.g. union, intersection and difference) become special cases of their relational equivalents. Similarly the cardinality operator rather than denoting the number of elements in a set now denotes the number of element pairs in a relation. As a

further aid section 3.3 introduces a special operator, `set`, which is used to give the set view of a relation.

3.3 The Drusilla Operators

The Drusilla primitive operators are defined in Table 3.2 using the typed relational calculus presented in section 3.2. The primitive operators are subdivided into two classes — unary (those that take one argument) and binary (those that take two arguments). The non-primitive operators are simply those that can be expressed in terms of the primitives. Primitive and non-primitive operators alike are all part of Drusilla and from now on the term *built-in* will be used to refer to both. The following subsections discuss the sources that influenced the choice of built-in operators.

3.3.1 The Relational Operators of RPL

MacLennan describes various relational operators in his papers [65, 66, 69] and gives a comprehensive description of the primitive and non-primitive intensional and extensional operators used by his relational language RPL in his internal report [67]. The Drusilla operators were based on these. The ones taken were those that satisfied certain criteria:

Fundamental: Those operators that are frequently used by MacLennan and seem fundamental to the expressive power of relational programming.

General: Operators should be generally applicable to relations. Many of MacLennan's operators were designed solely for manipulating specific relational data structures or functional relations. Also the operators should be defined for application to program and data.

Primitive: Cannot be easily or efficiently expressed in terms of other operators.

The generality consideration is necessary if MacLennan's division between intensional and extensional relations is to be excluded from Drusilla. However, there are exceptions — operators such a `dom`, `rng`, and `card` are taken although they can only be applied to data. They are included because without them certain expressive power is lost. There are only two Drusilla primitives not previously identified by MacLennan:

relation containership `[cont]` replaces MacLennans extensional relation application operator. There is no notion in Drusilla of applying an arbitrary relation to an argument, instead containership generates the element pairs of a relation. Section 3.4 discusses how it can be used to replace relation application.

set form `[set]` is used to give the set view of a relation. The normal set view of a relation is by definition:

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

Non-primitive Operators
Domain anti-restriction [<-] :: (A ↔ un × A ↔ B) ↔ (A ↔ B) s <- r ≡ neg s << r
Range anti-restriction [->] :: (A ↔ B × B ↔ un) ↔ (A ↔ B) r -> s ≡ r >> neg s
Relation difference [\] :: (A ↔ B × A ↔ B) ↔ (A ↔ B) r \ s ≡ r ⋀ neg s
Relation overide [@] :: (A ↔ B × A ↔ B) ↔ (A ↔ B) r @ s ≡ dom s <- r ⋁ s
Image of a set under a relation [img] :: (A ↔ B × A ↔ un) ↔ (B ↔ un) r img s ≡ rng (s << r)

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

Table 3.2: Drusilla operators

$$(x,y) \in R \Leftrightarrow x\,R\,y$$

As sets are not used directly in Drusilla, the set view is given by:

$$(x,y)\ (\text{set}\ R)\ \text{Unit} \Leftrightarrow x\,R\,y$$

3.3.2 Algebra-based Database Query Languages

Relational algebra has long been used as the basis for relational database query languages. Chapter 2 briefly described the basic algebraic operations and concludes that a relational programming language should have counterparts of these operations.

Each of these operations takes either one or two relations as its input and produces a new relation as its output. Codd defined eight such operators, two groups of four each. The first group consists of traditional set operations: union, intersection, difference and cartesian product (all modified slightly to take account of the fact that their operands are relations as opposed to arbitrary sets). The second group consists of specialised relational operations: select, project, join and divide. These operations are all generally applicable to n-ary relations. For the description of these operations it is useful to visualise each n-ary relation as a table of rows and columns in which each row is an n-tuple with n column elements. Since any given n-ary relation can be modelled by a binary relation, each operation is considered and the analogous operator(s) for binary relations in Drusilla identified. However the divide operation is not considered since it is not computable for binary relations that are represented by computable functions.

Union

The *union* operation is dyadic joining two relations together by taking the set union of their respective tuple sets. This may be visualised as taking the rows of the two tables representing the relations and putting them one after the other to form one long table. The union operator ($\backslash/$) is included in Drusilla.

Intersection

The *intersection* operation is dyadic joining two relations together by taking the set intersection of their respective tuple sets. If both relations are visualised as tables then this operation may be seen as forming a new table from those rows that are common to both of the tables. The intersection operator ($/\backslash$) is included in Drusilla.

Difference

The *difference* operation is dyadic joining two relations together by taking the set difference of their respective tuple sets. If both relations are visualised as tables then this operation may be seen as forming a new table from those rows that are

in the first table but not in the second. The difference operator (\) is included in Drusilla.

Cartesian Product

By defining *cartesian product* in the Drusilla calculus it is possible to derive an equivalent expression in the Drusilla language. This is done by systematically introducing Drusilla operators substituting them for their defining expressions. For example:

$$\begin{aligned}(a,b)\ (r\ \text{cartesianProduct}\ s)\ (x,y) &\Leftrightarrow a\ r\ b \wedge x\ s\ y\\ &\Leftrightarrow (a,b)\ (\texttt{set}\ r)\ \text{Unit} \wedge (x,y)\ (\texttt{set}\ s)\ \text{Unit}\\ &\Leftrightarrow (a,b)\ (\texttt{set}\ r)\ \text{Unit} \wedge \text{Unit}\ (\texttt{inv}\ (\texttt{set}\ s))\ (x,y)\\ &\Leftrightarrow \exists\, z.\ (a,b)\ (\texttt{set}\ r)\ z \wedge z\ (\texttt{inv}\ (\texttt{set}\ s))\ (x,y)\\ &\Leftrightarrow (a,b)\ (\texttt{set}\ r\ ;\ \texttt{inv}\ (\texttt{set}\ s))\ (x,y)\end{aligned}$$

$$\Rightarrow r\ \textit{cartesianProduct}\ s \equiv \texttt{set}\ r\ ;\ \texttt{inv}\ (\texttt{set}\ s)$$

If the programmer defined the relations r and s by explicitly listing their elements then the elements of this relation can all be evaluated.

If either of the relations r or s is represented by a computable function and their cartesian product is to be used as a *relator* then its application to a domain–range element pair of r would have to generate all the domain–range pairs of s. This however is obviously not computable. A similar, but computable relation, is given in Drusilla by parallel composition (| |) of relations. If this operator relation is viewed as a table then it is identical except for column ordering:

$$(a,x)\ (r\ ||\ s)\ (b,y) \Leftrightarrow a\ r\ b \wedge x\ s\ y$$

This is computable in relator form as it relates a pair of domain values to a pair of range values using the respective relations *r* and *s*.

Selection

The *selection* operation reduces the number of tuples in a relation by selecting only those that satisfy some predicate. Gray [39] states that any predicate can be replaced by the set of values of its arguments for which it is true. Therefore, any predicate p can be represented by a Drusilla relation r:

$$x\ r\ \text{Unit} \Leftrightarrow x\ p\ \text{True}$$

Since n-ary relations are modelled by binary relations selection must now have two predicates—one to select domain elements, the other to select range elements. Any given selection expression can then be translated into a calculus expression from which it is possible to derive a Drusilla language expression. Suppose selection is applied to relation s using predicates represented by relations f and g:

$$\begin{aligned} x\ (s\ \text{select}\ (f,g))\ y &\Leftrightarrow x\ s\ y \wedge x\ f\ \text{Unit} \wedge y\ g\ \text{Unit} \\ &\Leftrightarrow x\ (f << s)\ y \wedge y\ g\ \text{Unit} \\ &\Leftrightarrow x\ (f << s >> g)\ y \\ \Rightarrow s\ \text{select}\ (f,g) &\equiv f << s >> g \end{aligned}$$

The Drusilla operators analogous to selection are the domain and range restriction and anti-restriction operators (<<, >>, <-, ->).

Projection

The *projection* operation extracts specific fields from a relation, reducing the size of the relation tuples. If the relation is visualised as a table a projection reduces the number of columns. A given n-ary relation is modelled by a binary relation by splitting each n-tuple into a pair of tuples — one denoting a domain element, the other denoting a range element. The domain (`dom`) and range (`rng`) operators in Drusilla project, respectively, the domain and range fields and are therefore projection operators.

A more general form of projection can be performed using relations that extract tuple elements. An example of this is given in section 3.4.

In Drusilla, a relation can be specialised by partial instantiation of its domain and/or range. This is, in effect, a combination of selection and projection operations. It is a selection as it selects only those elements that include the specialising values. It is a projection as it reduces the relation tuple size. An example is specialising the addition relation `[+]` to derive the successor relation. Full projection would be impossible to compute for relations that are represented by computable functions because there may be an infinite number of domain-range element pairs. Relation specialisation will be discussed in more detail in section 3.4.

Join

The other dyadic operation is *join* which is defined for any two relations. It is defined in different ways depending on how many column names are common to both relations:

- If the two tables denoting the relations have no column names in common it behaves as a cartesian product operation and concatenates each row of the first table with each row of the second in turn.
- If the two tables have identical column names it behaves as a set intersection operation and produces a table of those rows that occur in both tables.
- If the two tables have some column names in common then it produces a table with all the column names from the first table together with any extra column names from the second one. Rows are selected from the first table and extra values are concatenated on from rows in the second table that have matching values in the common columns. This creates redundant columns and is referred to as *equi-join*. If the redundant columns are removed then it is called the *natural join*.

The action of join encompasses three different Drusilla operators. The notion of common column names is replaced in Drusilla by that of common domain / range types. The cartesian product of two relations of any arbitrary type has already been discussed. The intersection of two relations is permitted between any two relations that have the same domain and range types. The intersection operator (/\) has already been discussed. The relation composition operator (;) encapsulates the notion of natural-join for binary relations with the common column removed.

The dual composition operator (#) is also a form of join operation since it combines two relations on their common domain elements.

3.3.3 Binary Relations in Mathematics

Operations on binary relations have been much explored in mathematics and are well documented. Schmidt [96], Suppes [109] and Tarski [110] define the domain (dom), range (rng), inverse (inv), negation (neg) and relative product (or relation composition) (;) union (\/), intersection (/\) and difference (\) operators used in Drusilla.

3.3.4 Formal Specification Languages

One intended application for Drusilla is as a tool for rapid prototyping and fast program development. Consequently one aim of Drusilla is to simplify the task of deriving programs from formal specifications. A number of relational operators are included in the basic mathematical toolkit of the formal specification language Z as described by Spivey [104] and Woodcock and Loomes [121]. Consequently these have all been included as Drusilla operators, with the exception of transitive closure and reflexive transitive closure both of which can be replaced by recursion.

These operators comprise domain (dom), range (rng), inverse (inv), relation composition (;), relational image (img), domain restriction (<<), range restriction (>>), domain anti-restriction (<-), range anti-restriction (->). Z also includes a function overriding operator. This is included in Drusilla as a relation overriding operator (@) as MacLennan [66, 67] has previously suggested.

3.4 An Overview of Drusilla

Drusilla is a prototype relational programming system designed primarily to develop the relational paradigm in which binary relations form both the basic data structure and the computation mechanism. Relational programming is viewed as a generalisation of functional programming that encompasses many aspects of logic programming. It is hoped that Drusilla will be used as a tool for rapid prototyping of software and as a language for artificial intelligence applications.

This section gives a detailed description of the main features of Drusilla. A number of example programs are presented, some of which are described in detail. By the end of the chapter the reader should be able to understand all the programs

without any further explanation. The design of Drusilla is very much influenced by Miranda and hence this section has a structure very similar to Turner's [114] overview of Miranda.

3.4.1 Basic Ideas

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 which are to be computed. The order in which relation definitions are given has no significance; for example, there is no need for a relation definition to precede its first use. Drusilla has little excess baggage — there are no reserved words and the layout of a program has no syntactic significance. However, definitions must be terminated by a full stop. Any line starting with "--" is considered a comment. A very simple Drusilla program is given in Figure 3.1. It is just an example collection of Drusilla definitions for defining mathematical relations.

```
-- example Drusilla definitions

z = (cube || cube) ; [/].
(n) cube  n * n * n.
x = a + b.
y = a - b.
a = 10.
b = 5.
```

Figure 3.1: A simple Drusilla program

The central notion underlying Drusilla is that *all the world is a relation.* There is no notion of functional application of user-defined relations because it is not generally known whether a relation is functional. All the primitive relational operators (except relation containership `[cont]`), are however known to be functional and may therefore be used either as relations or as functions. This is demonstrated in the definitions of `cube`, `x`, `y` in Figure 3.1 where the functional forms of the arithmetic operations multiply, plus and minus are used. The definition of `z` illustrates the relational form of use of the division operator, denoted by enclosure in square brackets.

In the definition of the `cube` relation `n` is a formal parameter — its scope is limited to the definition in which it occurs. The other names introduced above have the whole program for their scope.

3.4.2 Discussion of Example Programs

This subsection aims to help elucidate the structure of Drusilla programs by explaining the creation of a few examples.

Factorial Definition

It is known that the factorial of zero is one. This is the base case for the recursive definition and can be defined explicitly as a relation:

```
{(0,1)}
```

The relation between n and $factorial\ (n-1)$ can be defined:

```
[- 1] ; fact
```

This expression can be used to define the relation between $n > 0$ and its factorial, $n * factorial\ (n-1)$:

```
id # ([- 1] ; fact) ; [*]
```

To ensure this relation is only applicable for $n > 0$ we override it with the base case for $n = 0$:

```
id # ([- 1] ; fact) ; [*] @ {(0,1)}
```

Fibonacci Definition

The fibonacci relation relates a number to its value in the fibonacci series and has a similar structure to the factorial definition. The base case for the recursive definition is for argument values of zero and one:

```
{(0,0),(1,1)}
```

The fibonacci of a number $n > 1$ is given by the expression $fib(n-1) + fib(n-2)$. The relation between n and $fib(n-1)$ can be expressed as a relation:

```
[- 1] ; fib
```

Similarly the relation between n and $fib(n-2)$ can be expressed as a relation:

```
[- 2] ; fib
```

These relations can be combined together using the dual composition operator (#) to create a relation between n and the pair $(fib(n-1), fib(n-2))$:

```
[- 1] ; fib # ([- 2] ; fib)
```

To get the some of these two fibonacci values we simply compose the relation with the addition relation:

```
[- 1] ; fib # ([- 2] ; fib) ; [+]
```

To make the relation applicable to all values $n > 1$ we override this relation with the base condition:

```
[- 1] ; fib # ([- 2] ; fib) ; [+] @ {(0,0),(1,1)}
```

Ackermann's Function as a Relation

Ackermann's function may defined in a modern functional language:

```
ack 0 y = y + 1
ack x 0 = ack (x-1) 1
ack x y = ack (x-1) (ack x (y-1))
```

As Drusilla does not use pattern matching we must define each condition as a relation and glue those relations together. The first case is defined as a partially parameterised relation:

```
(_,y) ackC  {(0,y+1)}.
```

The defining relation is between x and the result, but is only applicable for $x = 0$. The second case is similarly defined:

```
(x,_) ackB  {(0,1)} ; (x-1,_)ack.
```

The defining relation is between y and the result, but is only applicable for $y = 0$. The relation `{(0,1)}`, which relates y to 1 if and only if $y = 0$, is composed with `(x-1,_)ack` to form a relation between y and $ack(x-1,1)$ that holds when $y = 0$. The final case is defined as the relation:

```
(x,_) ackA  [-1] ; (x,_)ack ; (x-1,_)ack.
```

The first part of the expression, `[-1] ; (x,_)ack`, defines the relation between y and $ack\ x\ (y-1)$. This relation is composed with `(x-1,_)ack` to yield the relation between $ack\ x\ (y-1)$ and $ack\ (x-1)\ (ack\ x\ (y-1))$.

Finally the Ackermann relation must be constructed from these auxiliary relations. The pattern matching in the functional definition imposes an ordering on the definitions. If this ordering is not preserved then the meaning of the function will change. This ordering is preserved by gluing the relations together with the relation override operator (@):

```
ack = ackA @ ackB @ ackC.
```

Solving Quadratic Equations

Quadratic equations of the form

$$ax^2 + bx + c = 0$$

may be solved using the standard formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$$

The relation `quadSolve` solves the given quadratic equation, relating a triple of coefficients, `(a,b,c)`, to the roots of the equation:

```
(a,b,_) quadSolve   (a,b,_)quadB ; [- b] ; [/ (2*a)].
```

The expression $\sqrt{b^2 - 4 * a * c}$ is evaluated using the auxiliary definition `quadRoot`:

```
(a,_,c) quadRoot  sq ; [- (4*a*c)] ; [sqrt].
```

This is a non-functional relation — it relates the coefficients to both the positive and the negative roots of $b^2 - 4 * a * c$.

When `quadSolve` is used in forward mode it returns as result the set containing the two roots of the given equation.

3.4.3 Mathematical Relations

The numerical relations primitive to Drusilla are defined in Table 3.3. All relations can be used as elementary designations. The addition (+), subtraction (-) and multiplication (*) relations can be used to form compound designations because they are total functions. The division (/) and modulo (`mod`) relations although functional cannot be used for compound designations because they are undefined when the divisor is zero — they are partial functions. If they were used the result may be undefined and this would destroy our relational handling of exceptions which will be described in chapter 7. Square root (`sqrt`) cannot be used functionally because it is a relation, not a function — it relates any positive number to its positive and negative square roots; negative numbers are outside its domain.

The mathematical relations can be easily extended, for example a number of equality and inequality relations are shown in Figure 3.2. The identity relation `id` relates any value to itself. The equality relation `eq` relates a pair of values to `Unit` if they relate under the identity relation — it defines the set of all things that are equal. A value is related to any value except itself by `notId`. The `notEq` relation defines the set of values that are not equal. The relations 'less than or equal', `lessEq`, and 'greater than or equal', `greatEq`, are self explanatory.

3.4.4 Expressions

There are four expression forms:

Basic values: the simplest form of expression is a basic value such as a number, character string or the `Unit` element.

Tuples: a tuple is a sequence of elements of mixed type separated by commas and enclosed in parentheses. An example is `("Jones",Unit,39)`. Tuples are one of the two data structure forms in Drusilla.

Elementary relations designations: a program definition, formal parameter, a built-in operator enclosed in square brackets, or a relation defined in extension.

A relation is defined in extension by explicitly listing its elements in the form:

Functional Operators
Addition [+] :: (num × num) ↔ num (x,y) [+] $z \Leftrightarrow z = x + y$
Subtraction [-] :: (num × num) ↔ num (x,y) [-] $z \Leftrightarrow z = x - y$
Multiplication [*] :: (num × num) ↔ num (x,y) [*] $z \Leftrightarrow z = x * y$

Relational Operators
Division [/] :: (num × num) ↔ num (x,y) [/] $z \Leftrightarrow z = x / y$
Modulo [mod] :: (num × num) ↔ num (x,y) [mod] $z \Leftrightarrow z = x$ *modulo* y
Square root [sqrt] :: num ↔ num x [sqrt] $y \Leftrightarrow y$ *'square root of'* x
Less than relation [<] :: num ↔ num x [<] $y \Leftrightarrow x$ *'less than'* y
Greater than relation [>] :: num ↔ num x [>] $y \Leftrightarrow x$ *'greater than'* y

Table 3.3: Primitive mathematical operators

```
-- identity relation
(x) id (x).
-- less than or equal relation
lessEq = [<] ⋁ id.
-- greater than or equal relation
greatEq = [>] ⋁ id.
-- not identity relation
notId = neg id.
-- equality relation
eq = set id.
-- not equal relation
notEq = neg eq.
```

Figure 3.2: Equality and inequality definitions

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

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 replaced by relations as in the calculus. For example, the set of weekend days is denoted by the expression:

```
{("Sat",Unit),("Sun",Unit)}
```

This set could also be denoted by the relation

```
{(Unit,"Sat"),(Unit,"Sun")}
```

but the first form of set is the only one that can be used to restrict the domain (<<) or range (>>) of a relation. Relations defined in this way may be used as data structures or as program code relating inputs to outputs.

The use of relations as data structures allows relational database operations to be naturally expressed. An example of a relation projection operations are given in Figure 3.3 where projections of relation `workers` are taken. Such database operations are expressed most concisely when the relations are viewed as sets i.e. relations between elements and `Unit`. In modern functional languages the data structures are created using constructor functions. This is *not* the case in Drusilla — data structures are just values.

Compound relation designations: are constructed as in the calculus. Drusilla, like Ruby, is a constructive language — compound designations are built piecewise from smaller expressions, by applying the built-in, relational and mathematical operators as functions. The functional form of operator use makes Drusilla an applicative language. Equivalent expressions may be substituted for existing ones because the language has referential transparency. The syntax of a Drusilla expression is defined in Figure 3.4. Any expression that evaluates to a relation is called a *relational expression*.

3.4.5 Definitions

A Drusilla program consists of a collection of definitions, the grammar for which is given in Figure 3.5. There are three forms of definition: *value, fully parameterised* and *partially parameterised.*

Value Definitions

A *value definition* binds an identifier to an expression that can denote any form of Drusilla value including of course a relation. The definitions of `z`, `x`, `y`, `a` and `b` in Figure 3.1, `factRec` and `fib` in Figure 3.6 are all examples of value definitions. (A calculs definition of `factRec` is given in Figure 3.7 as an aid to clarity). The problem with value definitions is that they become difficult to formulate when the flow of information between relations does not have a natural pipeline structure. Experience with FP [3] has highlighted this problem. This is alleviated by introducing definition forms that allow formal parameters to be used.

```
-- 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,salary) getCode code.
(code,name,place,salary) getName name.
(code,name,place,salary) getPlace place.
(code,name,place,salary) getSalary salary.
(code,name,place,salary) getNameSalary (name,salary).
(code,name,place,salary) getCodePlace (code,place).

-- projection relations over workers
projectCode = getCode img workers.
projectName = getName img workers.
projectPlace = getPlace img workers.
projectSalary = getSalary img workers.
projectNameSalary = getNameSalary img workers.
projectCodePlace = getCodePlace img workers.
```

Figure 3.3: Example relation projections

compExp → *simpleExp* {*binaryOp simpleExp*}*
simpleExp → [*unaryOp*] *argExp*
argExp → *opSection* | *optCurryRel* | *domain*
domain → *number* | *string* | *unit* | *extensRel* | *tuple*
opSection → [(*leftSec* | *rightSec*)]
leftSec → *argExp primOp*
rightSec → *primOp argExp*
optCurryRel → [*curryTuple*] *curryObj* [*curryTuple*]
curryObj → *identifier* | *brackExp* | *elementOp*
elementOp → [*primOp*]
brackExp → (*compExp*)
tuple → (*compExp* {, *compExp*}*)
curryTuple → (*curryItem* {, *curryItem*}*)
curryItem → *compExp* | _
extensRel → { *domainPair* {, *domainPair* }* }
domainPair → (*domain* , *domain*)

Figure 3.4: Grammar for a Drusilla expression

program → {*relDefn*} *
relDefn → *nameBind* | *parDefn* .
nameBind → *identifier* = *compExp*
parDefn → *defnPars identifier compExp*
defnPars → (*parameter* {, *parameter*}*)
parameter → *identifier* | *defnPars* | _

Figure 3.5: Grammar for a Drusilla program

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

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

-- Euclid's greatest common divisor
(x,_) gcd    [> 0] << (id # [x mod] ; gcd) @ {(0,x)}.
```

Figure 3.6: Recursively defined mathematical functions

$$m\ \text{factRec}\ n \Leftrightarrow \neg\, m\ \text{id}\ 0 \wedge (m\text{-}1)\ \text{factRec}\ 1 \wedge (m,l)\ [*]\ n \vee m\ \text{id}\ 0 \wedge n\ \text{id}\ 1$$

Figure 3.7: Calculus definition of factorial

Fully Parameterised Definitions

A *fully parameterised definition* has the syntactic form:

domainTuple relationName definingExpression

It binds a name, *relationName*, 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. The formal parameters in *domainTuple* name the domain value so that formulation of the defining expression may be simplified. The definition of `cube` in Figure 3.1 is an example of such a definition. This form of definition can be used to extract tuple elements or to alter their nesting as demonstrated by the definitions of `fst`, `snd`, `lsh`, and `rsh` in Figure 3.8. A relation defined in this way must be functional because a given domain value relates to exactly one range value — the value of the defining expression. However, such functional relations cannot be applied to arguments since the aim of Drusilla is to explore the expressive power of relations rather than functions.

```
(x,y) fst x.
(x,y) snd y.
(x,y) swap (y,x).
(x) id (x).
((a,b),c) lsh (a,(b,c))
(a,(b,c)) rsh ((a,b),c)
```

Figure 3.8: General tuple operations

Partially Parameterised Definitions

A *partially parameterised definition* has the syntactic form:

domainTuple relationName definingExpression

It binds an identifier, *relationName*, to a relation defined by an expression, *defining-Expression*. Values in the domain of *relationName* are syntactically described by a tuple, *domainTuple*. This tuple contains *formal* and *anonymous* parameters. Part of any given domain value is named by the formal parameters and the rest is left unnamed by the anonymous parameters, denoted by underscores. If the anonymous parameters are extracted from *domainTuple* then they form a tuple (the *anonymous tuple*) that must syntactically match values in the domain of the defining relation, *definingExpression*. For example, if the anonymous tuple is a triple, then the domain values of *definingExpression* must also be triples. The definitions of the auxiliary relations `ackA`, `ackB` and `ackC` in Figure 3.9 are examples of partially parameterised definitions.

The pattern formed by the domain tuple does *not* constitute pattern matching in the functional language sense since it *cannot* be used to define a relation via several separate equations with different domain patterns. There is however a close alternative to this as demonstrated by the relational definition of Ackermann's function, `ack`, shown in Figure 3.9.

```
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 3.9: Definition of Ackermann's function as a relation

3.4.6 Higher-order Relations

Drusilla is a higher-order language — relations are first-class citizens and may be values in the domains and ranges of other relations. The relational values that occur in the domain and range elements of other relations are called *value-relations*. The relations `while` and `repeat` defined in Figure 3.10 are examples of higher-order relations. (The calculus definitions of both relations are given in Figure 3.11 as an aid to clarity). Both definitions contain subrelations, `p` and `f` in their domain values.

```
-- simulation of a while loop
(p,f,_) while  id @ (p << f ; (p,f,_)while).

-- simulation of a repeat loop
(p,f,_) repeat   f ; ((p,f,_)repeat @ (p << id)).

-- example use of while
createMantissa = ([> 1],[/ 10],_)while.
```

Figure 3.10: Example higher-order relations

Higher-order relations are very common because the relation is the main form of data structure.

3.4.7 Specialised Relations

Although application of user-defined relations to arguments is not permitted, it is possible to specialise a relation by partial instantiation of its domain and/or

$$
\begin{array}{|ll|}
\hline
(p,f,x)\ \text{while}\ y \Leftrightarrow & \neg\, x\ p\ \text{Unit} \wedge x\ \text{id}\ y \vee \\
& x\ p\ \text{Unit} \wedge x\ f\ z \wedge (p,f,z)\ \text{while}\ y \\
 & \\
(p,f,x)\ \text{repeat}\ y \Leftrightarrow & x\ f\ z \wedge (z\ p\ \text{Unit} \wedge z\ \text{id}\ y \vee \\
& \neg\, z\ p\ \text{Unit} \wedge (p,f,z)\ \text{repeat}\ y) \\
\hline
\end{array}
$$

Figure 3.11: Calculus definitions of `while` and `repeat`

range. This 'freezes' values into the domain/range of the relation. For example the expression `[- 1]` used above in the definition of `ackA` in Figure 3.9 denotes the partial instantiation of the domain of arithmetic plus relation; the value being 'frozen in' is `1`. This is actually a syntactic sugar for the normal Drusilla relation specialisation syntax `(_,1)[-]`. It denotes the predecessor relation which holds between any number and that number minus one. Placement of the tuple before the relation indicates that the domain is being specialised. Placement of the tuple after the relation indicates that the range is being specialised. The specialising tuple is a syntactic description of values in the domain of the relation. The presence of the anonymous value (_) indicates that no value is being instantiated in that part of the domain. As with partially parameterised definitions, the anonymous values can be extracted to form an *anonymous tuple*. The anonymous tuple forms a syntactic pattern that describes values in the domain of the specialised relation. The meaning of the specialised relation is not changed from the original — it is just a special case of it. For example:

```
x (_,1)[+] z ⇔ (x,1) [+] z
```

Specialisation of user-defined relations is demonstrated in the definitions of `ackB` and `ackA` in Figure 3.9. The specialising values may themselves be expressions as demonstrated in the definitions of `quadSolve` (expression `[/ (2*a)]`) and `quadRoot` (expression `[- (4*a*c)]`) in Figure 3.12. (The calculus definition of `quadSolve` is given in Figure 3.13 as an aid to clarity).

```
(a,b,_) quadSolve  (a,b,_)quadRoot ; [- b] ; [/ (2*a)].

(a,_,c) quadRoot  sq ; [- (4*a*c)] ; [sqrt].

(n) sq  n * n.
```

Figure 3.12: Program to solve quadratic equations

Relation specialisation is also used by the the definitions of `combI`, `curryK` and `cur` in Figure 3.14 where the S, K and I combinators are defined. Calculus types, although not part of the original source code, are included to aid clarity, as are the calculus definitions shown in Figure 3.15. The ability to define and

(a,b,c) quadSolve x	⇔	(a,b,c) quadRoot r ∧ (r-b,2 *a) [/] x
(a,b,c) quadRoot r	⇔	b sq squareB ∧ (squareB,4*a*c) [-] x ∧ x [sqrt] r
(n) sq y	⇔	(n,n) [*] y

Figure 3.13: Calculus definition of quadratic equation solver

use these proves the computational completeness of Drusilla. The combinator I is defined as I = S K K by specialising the S combinator. In order for the definition of `combI` to be calculus type correct, a curried form of the K combinator, `curryK`, has to be used. The curried form is obtained from the relation `cur` which demonstrates how relation specialisation can emulate currying as used in functional languages. The calculus type of `curryK` is different to that of `combK` but its meaning as a combinator is preserved.

Although there is no relation application by specialising the range of the containership operator (`cont`), a similar effect can be achieved. For example the expression `[cont](_,Unit)` is a relation between a set and its elements. Similarly `[cont](1,_)` is a relation between a relation `r` and any value `x` such that `1 r x`.

The primitive unary operators can also be specialised. If $[\oplus]$ is a unary operator for which x and y are, respectively, domain and range values then $[x \oplus]$ denotes a domain sepecialisation and $[\oplus\ y]$ denotes a range specialisation, defined:

$$\begin{array}{llll} y & [x\ \oplus] & Unit & \Leftrightarrow\ x\ \oplus\ y \\ x & [\oplus\ y] & Unit & \Leftrightarrow\ x\ \oplus\ y \end{array}$$

These specialisations are, respectively, syntactic sugar for the expressions:

$$\begin{array}{lll} [x\ \oplus] & \equiv & (x,\_)(set\ [\oplus]) \\ [\oplus\ y] & \equiv & (\_,y)(set\ [\oplus]) \end{array}$$

Example specialisations are `[> 0]`, `[< 0]` and `[mod y]` used in the definitions shown in Figure 3.16.

3.4.8 Lazy Reduction Semantics

Drusilla's semantics are reduction as normally associated with a functional language and parameters are passed by simple substitution of actual for formal parameters as defined by the beta rule of the $\lambda - calculus$. The evaluation mechanism, called relational laziness, is lazy in the sense that no subexpression is evaluated until its value is known to be required. More explanation of this mechanism is given in chapter 7. This permits non-strict relations (relations which hold even if part of their domain is undefined) such as *fst* and *snd*. The other main consequence of laziness is that it permits definition and handling of infinite relational data structures. For example the definitions of `nats`, `odds` and `squares` in Figure 3.17.

```
-- cancellator combinator K
-- K x y = x
combK :: (A x B) <-> A
(x,y) combK (x).

-- meta-application combinator S
-- S f g x = (f x) (g x)
combS :: ((P x N) <-> Q x P <-> N x P) <-> Q
(f,g,_) combS (id # g ; f).

-- identity combinator I
-- I x = x
-- I = S K K
combI :: A <-> A
combI = (combK,curryK,_)combS.

-- curried form of combinator K
curryK :: A <-> (B <-> A)
curryK = (combK,_)cur.

-- relation currying used in definition of combI
cur :: ((S x W) <-> V x S) <-> (W <-> V)
(r,x) cur ( (x,_)r ).

-- identity used in definition of combS
id :: A <-> A
(x) id (x).
```

Figure 3.14: Definitions of the S, K and I combinators

$$(x,y)\ \text{combK}\ x$$

$$(f,g,x)\ \text{combS}\ y \Leftrightarrow x\ g\ z \wedge (x,z)\ f\ y$$

$$x\ \text{combI}\ x$$

Figure 3.15: Calculus definitions of S, K and I combinators

```
-- signum
sgn = (0-1,_)fst @ ([> 0] << (1,_)fst) @ {(0,0)}.

-- absolute negate
absNegate = absB ; negate.

-- unary minus
(n) negate  (0 - n).

-- absolute value
abs = id @ ([< 0] << [* (0-1)]).

-- set of even numbers
even = (_,2)divisableBy.

-- set of odd numbers
odd = neg even.

-- (x,y) divisableBy Unit
-- if and only if x mod y = 0
(_,y) divisableBy  [mod y] ; {(0,Unit)}.

-- positive square root
posSqRoot = [sqrt] >> [> 0].
```

Figure 3.16: Standard mathematical functions

```
nats = {(Unit,0)} \/ (nats ; [+ 1]).
odds = {(Unit,1)} \/ (odds ; [+ 2]).
squares = inv nats << sq.
(n) sq  n * n.
```

Figure 3.17: Examples of infinite relational data structures

3.4.9 The Programming Environment

The current environment is very much a prototype. The basic action is to evaluate expressions supplied by the user at the terminal in the environment established by a program. A Drusilla program is written as a normal UNIX file. The interpreter is written in Miranda and is invoked by applying the interpreting function to the file name of a Drusilla program. No interactive editing of a Drusilla program within the environment is possible.

Expression evaluation takes place in one of three modes: show, forward, or test. The mode can be determined syntactically from the expression.

Show mode is evaluation of any Drusilla expression. If the result is a relation then its element pairs should be explicitly available.Some example expressions are:

example expression	result
`{(0,1),(1,2)} \/ {(3,23)}`	`{(0,1),(1,2),(3,23)}`
`2 + 3`	`5`
`nats`	`{(Unit,0),(Unit,1),...}`

Forward mode evaluation involves applying a relational expression to an expression denoting a domain value to yield a set of related range values. Some example expressions are:

example expression	result
`(id) 2`	`{ 2 }`
`(fact) 4`	`{ 24 }`
`(ack) (2,2)`	`{ 7 }`
`(quad) (1,0-7,12)`	`{ 3.0, 4.0 }`
`(createMantissa) 456`	`{ 0.456 }`

Test mode is a test of whether given domain and range expressions are related under a given relational expression. Some example expressions are:

example expression	result
`(2) (id) (2)`	`"TRUE"`
`(4) (fact) (12)`	`"TRUE"`
`((2,2)) (ack) (11)`	`"FALSE"`
`((1,0-7,12)) (quad) (23)`	`"FALSE"`

3.4.10 Polymorphic Strong Typing

Drusilla is statically and hence strongly typed. That is every expression and every subexpression has a type that can be deduced at compile time and any inconsistency in the type structure of a program results in a compile time error message. Drusilla actually possesses two related type systems:

Calculus types reflect the mathematical structure of the program.

Moded types are built on top of calculus types and convey the operational structure of the program — the modes of use possible for different relations.

If the definitions in a program are viewed as pieces in a jigsaw puzzle then the type systems dictate which pieces may be joined together. Calculus types ensure that two pieces can only be put together if meaningful to the overall picture and moded types are a further restriction that ensure pieces can only be put together if their edges match.

Calculus types (and hence moded types) are polymorphic in the sense of Milner [75]. Polymorphism is indicated by use of upper case letters as generic type variables. For example calculus types for some of the relations defined so far:

```
fst ::  (A x B) ↔ A
snd ::  (A x B) ↔ B
sq ::  num ↔ num
squares ::  num ↔ num
nats ::  un ↔ num
```

and moded types for the same relations:

```
fst ??  fo[(A x B) ↔ A]]
fst ??  te[(eA x B) ↔ eA]
snd ??  fo[(A x B) ↔ B]]
snd ??  te[(A x eB) ↔ eB]
sq ??  fo[num ↔ num]
sq ??  te[num ↔ num]
squares ??  sh[num ↔ num]
squares ??  fo[num ↔ num]
squares ??  te[num ↔ num]
nats ??  sh[un ↔ num]
nats ??  fo[un ↔ num]
nats ??  te[un ↔ num]
```

The need for moded types is discussed in chapter 4.

3.5 Conclusion

This chapter has introduced a new relational language — Drusilla. The underlying mathematical model of this language has been defined and used to formally define the built-in operators. Examples of operator use have been given in several example programs.

The implementation of this language must avoid MacLennan's constraint of fixed relation representations. The next three chapters consider techniques for implementing Drusilla. Chapter 7 explores more fully the programming styles possible in Drusilla.

Chapter 4

The Representation Bottleneck

4.1 Introduction

In his Turing award lecture Backus [3] identifies the barrier to expressive power in imperative languages as the 'von Neumann bottleneck'. He says:

> *'The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.'*

He observes that the assignment statement splits programming into two worlds: a world of expressions and a world of statements. His conclusion being that the world of expressions is desirable as it has useful mathematical properties and most computation takes place there. The world of statements, however, is undesirable as it only exists for computations centered around the assignment statement.

In a similar fashion an explicit constraint on expressive power in RPL can be identified as 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 directly inhibits freedom of expression because it separates the relational operators into two classes: one applicable solely to intensional relations and one applicable solely to extensional relations. This representation bottleneck keeps the programmer thinking in terms of computable functions and data structures and as such fails to preserve the relational abstraction. As a consequence RPL arguably offers little more, in terms of economy of expression, than the functional language FP [3] on which it is based.

MacLennan [67] refers to his operator division as the 'elimination of polymorphism' — operators are only defined for one representation even if they are implementable and useful on others. Strachey [107] originally distinguished between two major forms of polymorphism:

Parametric polymorphism is obtained when a function works uniformly on a range of types; these types exhibit some common structure.

Ad-hoc polymorphism is obtained when a function works, or appears to work uniformly on a range of types (which may not exhibit a common structure)

and may behave in unrelated ways for each type.

Ad-hoc polymorphism is the kind MacLennan is referring to. If the representation bottleneck is to be removed then each relational operator must be defined for different relation representations and the same symbol used for each occurrence, regardless of the representation of the relation(s) it is applied to and the context used to decide the appropriate definition.

This particular form of ad-hoc polymorphism is called *overloading* and *overload resolution* is the process of selecting appropriate operator definitions. If relational abstraction is to be preserved at the programming level then the system must automatically select representations and resolve operator overloading to preserve relational abstraction.

Section 4.2 discusses different relation representations and their relative merits and demerits. Section 4.3 examines related work on automatic representation selection and overload resolution. Section 4.4 describes a mechanism, based on abstract interpretation, for automatic representation selection and overload resolution. Section 4.5 identifies the weaknesses and failings of this approach. Section 4.6 describes a more powerful approach for representation selection. This is a mechanism based on Milner type inference called *typed representations*. Section 4.7 compares the two approaches to representation selection. Section 4.8 explains how the information generated by the typed representation analysis can be used to resolve operator overloading in expressions. Even with automatic representation selection constraints still exist on programmer freedom. Section 4.9 discusses how these constraints might be reported to the programmer. Finally, section 4.10 offers conclusions as to the degree of success with which automatic representation selection alleviates the representation bottleneck.

4.2 Relation Representation Techniques

MacLennan [67] identifies several representations of relations suited to implementation in a functional language. Drusilla is implemented in Miranda and uses a variety of relation representation techniques.

4.2.1 Extensional Representations

An *extensional representation* of a relation stores the relation elements in a data structure. This method can only be used if all the relation elements, or some formula for their generation, is known. The programmer may give an *extensional definition* of a relation by explicitly stating this information. If lazy evaluation is used in the implementation then the formula can generate an extensional relation with an infinite number of elements. For example, `s` is explicitly defined and `natSet` is a formula for generating the infinite set of natural numbers:

```
s = [(1,"a"),(2,"b"),(3,"c"),(2,"d"),(1,"e")]
natSet = [(Unit,0)] \/ (natSet ; [+ 1])
```

Extensional definitions can be extensionally represented by placing their elements in some data structure. If the data structure used is to support the different modes of relation use described in chapter 3 then it must allow associative lookup between related domain and range elements.

For a given extensionally defined relation an association list where each element is a pair could be used. In each pair the first element is a domain value and the second element is a list of range values related to that domain value. For example, `s` above would be represented:

```
[(1,["a","e"]),(2,["b","d"]),(3,["c"])]
```

Relation `natSet` would be represented:

```
[(Unit,[0,1,2, ...])]
```

For each element to be represented the set of range values related to some domain value must be known. However this range set can be lazily evaluated as with `natSet`.

This form of association list may be detrimental to laziness, especially if infinite extensional relations are to be manipulated. Consider for example the union of identity over zero with identity over the positive natural numbers:

```
[(0,0)] \/ [(1,1),(2,2), ...]
```

This expression would be translated to the representation level as:

```
union [(0,[0])] [(1,[1]),(2,[2]), ...]
```

The result should be a lazily evaluated list:

```
[(0,[0]),(1,[1]),(2,[2]), ...]
```

The first element `(0,[0])` should pair domain value zero with the set of range values it is related to under the two relations. The implementation is unlikely to be aware that value·zero is outside the domain of the second relation and consequently lazily evaluates the infinite list of positive numbers searching for domain value zero. A non-terminating computation is entered and no value ever returned as result.

It would be better for laziness if an element could be represented as soon as any pair of related domain and range values is known. A better choice of association list would be one with an element for each pair of related domain and range values. Relations `s` and `natSet` would respectively be represented by:

```
[(1,"a"),(1,"e"),(2,"b"),(2,"d"),(3,"c")]
    [(Unit,0),(Unit,1),(Unit,2), ...]
```

The above union expression would be translated to the representation level:

```
union [(0,0)] [(1,1),(2,2), ...]
```

When evaluated this gives the required result:

```
[(0,0),(1,1),(2,2), ...]
```

4.2.2 Intensional Representations

An intensional representation of a relation uses a computable function to represent the relation elements.

Corresponding Computable Function

If the relation being represented is computable and deterministic then it can be represented by the corresponding function in a functional language. The basic arithmetic operations and the relational operators that are known to be functional may be represented by such functions. They can only be used to represent those user-defined relations for which functionality is known. If functional relation R is represented by function f then:

$$x\,R\,y \Leftrightarrow y = f\,x$$

This form of mapping a function into a relation is referred to by Meijer [74] as taking its *graph*. For example the union of two relations S and T represented respectively by the functions f and g:

$$x\,(S \cup T)\,y \Leftrightarrow y = f\,x \vee y = g\,x$$

Set-valued Functions

Set-valued functions can be used to represent relations that are computable but non-functional. 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 $\{y_1, \ldots, y_n\}$. If x is not in the domain of r then this set will be empty. If R is functional then the set will contain at most one result, otherwise it can contain any number of results. This relationship can be stated formally:

$$x\,R\,y \Leftrightarrow y \in f\,x$$

Meijer [74] refers to the mapping of a set-valued function into a relation, as taking the function's *choice*. He refers to the mapping of a relation into a set-valued function as taking the *breadth* of the relation. Breadth and choice, as he observes, establish a bijection between set-valued functions and relations. For example the union of two relations S and T represented respectively by the functions f and g:

$$x\,(S \cup T)\,y \Leftrightarrow y \in f\,x \vee y \in g\,x$$

Characteristic Functions

Boolean-valued characteristic functions 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`. More formally:

$$x\,R\,y \Leftrightarrow f\,(x,y)$$

For example the union of two relations S and T represented respectively by the functions f and g:

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

4.2.3 Coercing Between Representations

In practice no one representation is ideal for all purposes. For this reason it may be appropriate to make a number of representations available and permit coercions between representations. We describe various representation coercions and show how they may be implemented in Miranda.

Coercing an Association List

Extensional representation is essential for show mode where all element pairs of a relation are to be listed. It is also the most flexible representation in that it can also be used in the other two modes of relation use. An association list can be used in forward mode, if it is coerced by associative look up, into a set-valued function:

```
coerceALtoSF :: [(*,**)] -> * -> [**]
coerceALtoSF rel domVal =
    map snd (filter ((=) domVal . fst) rel)
```

This is less efficient than a natural set-valued function due to the computational cost of associative look up. It also requires equality to be defined over relation domain values.

An association list can also be used in test mode by coercing it into a characteristic function. This can be done using a membership function:

```
coerceALtoCF :: [(*,**)] -> (*,**) -> bool
coerceALtoCF rel (domVal,rngVal) = member rel (domVal,rngVal)
```

This will be less efficient than a natural characteristic function due to the cost of the look up operation. The coercion also requires equality to be defined over both domain and range values.

For both association list coercions every pair in the representing data structure must be examined until the desired element(s) are found. If this data structure is infinite and the desired element is not present, a non-terminating computation is entered! Furthermore the need for equality to be defined for relation elements means that should the elements themselves be relations they cannot be represented by functions.

Coercing a Set-valued function

Point-to-point and set-valued functions are less flexible than association lists as they cannot be used in show mode. However, they are the most efficient representation for forward mode and can represent relations whose domain elements are represented by functions.

A relation represented by a set-valued function can be used in test mode by coercing its representation into a characteristic function:

```
coerceSFtoCF :: (* -> [**]) -> (*,**) -> bool
coerceSFtoCF f (x,y) = member (f x) y
```

This co-ercion requires equality to be defined for the relation's range elements.

Characteristic Functions Cannot be Coerced

Characteristic functions are less flexible than the other representations considered because they can only be used in test mode. However, they do not force equality to be defined on relation domain or range values. Some operations only use relations in test mode: for example, the restricting relation in domain and range restriction and anti-restriction operations. A characteristic function is the only possible representation for some operations: for example, if relation R is represented by characteristic function f then the inverse and negation of R can be represented respectively by the characteristic functions:

```
invCF :: ((*,**) -> bool) -> (**,*) -> bool
invCF f (x,y) = f(y,x)

negCF :: ((*,**) -> bool) -> (*,**) -> bool
negCF f (x,y) = ~(f (x,y))
```

If R is represented by a set-valued function then, by coercing it to a characteristic function, these definitions can be used.

Furthermore the characteristic function can represent relations whose domain and range elements contain relations represented by functions. The inverse of a function can be handled relationally by constraining the mode of use to test mode.

The relation representations discussed have their various merits and demerits. The best representation for any relation depends very much on its intended modes of use and the operators used to manipulate it.

4.3 Related Work

Section 4.3.1 reviews related work on automatic representation selection. Section 4.3.2 examines related work on overload resolution and discusses whether the operators would be easier to implement in the functional language Haskell which provides a mechanism for handling ad-hoc polymorphism.

4.3.1 Automatic Representation Selection

This subsection reviews related work in the area of representation selection and assesses how relevant this work is to selecting relation representations. One aspect common to the approaches discussed is that they are concerned solely with selecting representations for data structures and not with the more general

problem of selecting representations for those program components that perform computation.

Low's Overview of Issues

An overview of representation selection up to 1978 is given by Low [64], who observes that automatic data structure selection should allow the programmer to use abstract data types without having concern for their underlying representation. The data structures implementing them should be chosen automatically. This is the aim for Drusilla — the programmer should see relations but not their representations.

Low [64] identifies several points of importance in automatic representation selection:

Representation alternatives: given any abstract data type it is rarely the case that one representation is optimal for all programs.

Multiple representations: often there is no one representation that is most efficient or even applicable for all the primitive operations applied to the structure.

Information gathering: it is crucial to know how the abstract structures are used within the program.

These points are clearly of relevance to relation representation selection. Section 4.2 discussed alternative relation representations (association list, set-valued function and characteristic function) and concluded that no one representation is always possible or desirable. The effect of multiple representations can be achieved for a relation by coercing its representation. Information should be gathered concerning the operators used to manipulate relations.

Low identifies three major techniques for determining how abstract structures are used in a program:

1. Requiring the programmer to provide information.
2. Monitoring executions of the program.
3. Static analysis of the program.

Technique 1 can be performed through interactive conversations between the user and the system. This is not suitable — it should be possible for the Drusilla programmer to enter and execute a program without consulting the system. Alternatively, the user can be required to make special declarations in the program source. This could be used in the Drusilla system if the declarations are, in some way, abstracted away from actual representations, although it is better for the programmer if such declarations are not necessary.

Technique 2 can only be used if default representations for the abstract structures can be chosen. This is not possible for Drusilla programs since the problem

is to find those representations that permit program execution. Moreover, a programmer should be able to use a program without having to indulge in trial executions.

Technique 3 is used to generate information about the combinations of primitive operators used in a program. Ideally, this technique should be used for selecting representations in Drusilla programs as it requires no programmer involvement. The analysis can be based on information concerning the constraints relational operators impose between their argument and result representations.

Rule Based Approaches

Rosenchein and Katz [89] construct a model for the process of selecting representations for data structures. This model however is intended to serve as the basis for a knowledge-based interactive system for aiding programmers in the selection process rather than automating it. Kant [55] describes a similar system for high level language program specifications. A knowledge base of refinement rules details the implementation of high level language constructs.

The SETL Approach

SETL [100] is a 'very high level' language based on set theory. The main set-theoretic objects in SETL are finite sets and maps. A set is an unordered collections of objects with no element repetitions and a map is a set of pairs which can represent either a function or a relation. In the 1970's much work on automatic data structure selection was undertaken for implementation of the SETL language. Schwartz [99, 98, 35], Schonenberg [97] and Liu [63] all describe in varying detail the same compiler optimisations.

One of the basic ideas behind the SETL system is that the data representations used to realise an algorithm should depend on the program code and not vice-versa. Algorithms can be coded without specifying any non-set-theoretic data structures. Representations for the abstract data structures are automatically selected by an optimising compiler. The task of data structure selection is a complex one as each structure is efficient for some operations but inefficient for others.

The central notion in the SETL representation scheme is that of a *base*. Bases are auxiliary data structures that permit efficient access to related groups of variables. This is the only way bases are used; they are not explicitly manipulated by the program. Each auxiliary identifier declared to be a base constitutes a 'universal domain' for certain program objects.

A base is ordinarily represented as a linked hash table of element blocks. Each block holds, in addition to the value of the object it represents, fields that can be used to store related values. A set can be declared a subset of a base B and a map variable whose domain is a subset of B can be declared as a map from elements of B. For such variables there are three possible representations: *local, indexed* and *sparse*. In local representation a single bit element is reserved in each element block of B. Indexed representation uses a separate array which is indexed by the

element block indices of *B*. Sparse representation uses a linked hash table whose elements are pointers into *B*.

One of the main goals of introducing bases and based objects into a SETL program is to improve efficiency by reducing the number of hashing operations required during its execution. For example if a number of (possibly overlapping) sets are subsets of some universal set *B* then *B* can be introduced as a base and the other sets based on *B*. The number of hashing operations in the program will then, typically, be decreased since there is no need to hash into each set, only into the base *B* from which the elements of the other sets can be accessed.

It is the task of the automatic representation selection algorithm to identify such 'universal' sets. The algorithm was originally described in a report [63] and a paper [98]. Successive revisions of the algorithm are described in papers [97,35].

The algorithm initially generates provisional bases and based representations for variables involved in operations. Separate bases are created for each hashing operation and thus reflect only 'local' information. These bases are then propagated globally so that they may be integrated into an overall basing structure.

This algorithm uses information generated by subjecting the program to various analysis techniques. One of these is determination of type information at compile time by abstract interpretation as described by Tenenbaum [112]. Two instances of a variable can only be based on the same base if they are instantiated to values of the same type. Another important analysis statically determines inclusion relationships when an element is a member of a set or when one set is a subset of another. These two analysis techniques are significant to later work by Paige (discussed below) and to the representation selection work described in this thesis since they establish links between types and representations.

The SETL representation selection algorithm identifies subcollections of a program's data as independent 'universes' and describes the program objects in terms of their relationships to these universes. It decides only the major representation details; finer details are refined by choosing between the three different set/map representations discussed above: local, indexed and sparse. These finer decisions may further reduce the number of hashing operations involved in a program.

The SETL approach is concerned solely with efficiency. There is little variety in the data structures used and some degree of hashing is always retained. The approach is also largely heuristic and lacks firm theoretical basis. Paige's work, discussed below, follows in the same vein but introduces more theory. The SETL designers did not consider computable functions as representations. Functions are more general than extensional representations since they do not have to be finite and may be more efficient than hashing. 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 [98] describes nine different types of analysis used by the compiler.

After SETL... Work by Paige

Paige [18] 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.

An SQ2+ program is initially compiled into intermediate level imperative code in a language that is similar to SETL called SETM. This transformation is made using techniques described by Paige and Henglein [81]. 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 in the SETM intermediate code.

Paige [80] describes the representations used to implement set-theoretic operations in the SETM code that results from compilation of an SQ2+ query. He describes four basic kinds of data structure which extend those discussed by Paige and Henglein [81]:

1. The simplest data structure considered for implementation of a set is a doubly linked list with pointers to the first and last list cells. Each cell stores an element of the set. If the set is a map than each list cell stores a domain value and the range value(s) related to that domain value. This is similar to the association list relation representation described in section 4.2. Such a representation is called unbased and is capable of supporting all the basic set operations. It is however inefficient for operations which require associative access of elements. For example if an element is to be randomly taken from set S and searched for in set Q.

2. More efficient associative access is supported by based representations of sets. For example if sets S and Q are stored in the same place then access of an element from S simultaneously locates that element in Q. A universal set B is introduced as a base for S and Q. B and Q are represented as a collection of records, each record containing a Q and an S field. The elements of the B field are the values in set B and serve as a key. The Q field stores an 'undefined' value if the B field value does not belong to Q. Those records whose B field value does belong to Q are connected by a doubly linked list stored in the Q fields. The first and last cells of the Q field list are indicated by pointers. The elements of Q are said to be *strongly based* on B. The relationship between B, S and Q is:

$$S \cup Q \subset B$$

3. In the previous example set S is stored separately from B and Q as a doubly linked list of pointers to the records whose B fields hold a value which is in S. The elements of S said to be *weakly based* on B.

4. The fourth kind of data structure considered is an array. Arrays are applicable for any set of static size or of bounded dynamic size, and can be used instead of doubly linked lists in based representations.

The data structures used here improve the SETL based representations in several ways. SETL lacks a linked list representation for sets with unbased elements, does not permit iteration over sets with strongly based elements, and requires all sets with weakly based elements to be hashed making element addition expensive.

This work adds type theory to the SETL basing approach to representation selection by establishing a link between monomorphic subtypes and representations. This thesis identifies a similar link between (ad-hoc and parametric) polymorphic types and representation.

4.3.2 Overload Resolution

Overload resolution is a particular form of representation selection which selects a representation for a function when several candidate representations exist. Overload resolution in Drusilla is the process of selecting definitions for operators. The definition given to a particular operator must be applicable to the representations of the relation(s) it manipulates. The process of selecting definitions for operators is therefore dependent on the process of selecting representations for relations.

The Co-ordinating Operator Constructor in SQUIRAL

Smith and Chang [102] describe SQUIRAL (Smart QUery Interface for Relational Algebra) as a database interface that employs automatic programming techniques to analyse and refine high-level query specifications. The interface is regarded as a special purpose 'programmer'. The user formulates a query abstractly and then defines it in terms of the database primitives. The interface 'programmer' then takes over and progressively refines this query until it is expressed efficiently in machine primitives.

A query may be thought of as an operator tree where the leaf nodes are the given relations to be manipulated. Each node is an operator which operates on its descendant nodes. The root node represents the final operation which yields the result. Each operator (example operators are UNION, INTERSECT, PROJECT, JOIN) can be most efficiently implemented if it can assume its input tuples will be supplied in sort order on some domain. Each operator has a number of alternative implementation procedures which are applicable to relations sorted on different domains.

The co-ordinating operator constructor of SQUIRAL takes an operator tree and implements each operator from the set of alternative procedures in such a way that the sort orders of intermediate relations passed between operators are optimally co-ordinated. The scheme tries to maximize the efficiency of sort order decisions by performing two passes of the operator tree.

The first pass is an upward one and labels each branch in the tree with the set of sort orders that can be efficiently generated from lower operations. These are called the *preferred* sort orders. The second pass is a downward one which, for each operator node, selects the preferred sort order that most efficiently generates the sort order that node must pass up. With this selection the implementation of

the operator at that node is simultaneously constructed. Operator overloading is resolved by this downward pass.

The co-ordinating operator constructor of SQUIRAL is the main inspiration for the abstract interpretation based approach to relation representation selection described in section 4.4. For this a Drusilla expression is perceived as an operator tree. An upward pass of the operator tree labels each branch with the set of representations that can possibly be generated from lower operations. A downward pass of the tree then selects actual representations from the possible representations.

Overload Resolution in Ada

Operator overloading occurs in Ada where the same identifier can appear in different enumeration types whose scopes overlap and as the name of several functions. Runciman [92] summarises the Ada overload resolution requirements and reviews several published solutions.

The problem of resolving an overloaded expression is that of obtaining a single meaning for it. Overloading in the earliest implementation of Ada was resolved by a computationally expensive algorithm. Alternating top-down and bottom-up traversals of an expression tree propagate information in both directions until only one candidate definition remains for each operator node.

This algorithm was greatly simplified by an informal argument that resolution may be viewed as the communication of constraints between nodes in the expression tree. Messages between any pair of nodes can be exchanged by an upward collection to the root followed by a downward distribution. In other words one bottom-up followed by one top-down pass should be sufficient for resolution.

The same approach is taken in SQUIRAL and it further encourages the abstract interpretation approach to representation selection.

The Haskell Type Class System

Section 4.1 distinguished between *ad-hoc* and *parametric polymorphism*. An example of ad-hoc polymorphism is where multiplication is overloaded — the same symbol is used to denote multiplication of integers (e.g. 3 * 4) and multiplication of floating point values (e.g. 3.0 * 4.0). An example of parametric polymorphism is the length function which acts in the same way for a list of integers as it does for a list of floating point numbers. The Milner [75] type system is a widely accepted approach to parametric polymorphism.

Type classes [117] are an extension to the traditional Milner type system. They subsume many uses of ad-hoc polymorphism and enhance the expressive power of a polymorphic language. Type classes have been included in the language Haskell [44].

A *type class* is a family of types (whose members are called instances of the type) for which a collection of functions (the *member functions* of the class) are defined. For example, a type class can be used to overload multiplication as discussed above:

```
class  Num a where
       (*) :: a -> a -> a
```

A type a belongs to this class if has a function named `(*)` of the appropriate type. Instances of this class may be declared:

```
instance  Num Int where
          (*) =  multInt

instance  Num Float where
          (*) =  multFloat
```

These two definitions assert, respectively, that `Int` and `Float` belong to the class Num.

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. For example, the relation class with union and composition operators as member functions would be:

```
class  Relation a where
       union :: a -> a -> a
       comp :: a -> a -> a
```

A type a belongs to the class relation if it has `union` and `comp` operations defined for it.

An instance of this relation for association lists would be:

```
instance  Relation [(a,b)] where
          union = unionAl
          comp = compAl
```

Here the definitions of functions `unionAl` and `compAl` are not important but their type signatures are:

```
unionAl :: [(a,b)] -> [(a,b)] -> [(a,b)]
compAl :: [(a,b)] -> [(b,c)] -> [(a,c)]
```

The type of `unionAl` matches the `union` class type perfectly — it takes two relations of the same type and yields as result another relation of that type. The type of `compAl`, however, conflicts with the `comp` class type because its type is more general. The two relations it takes as argument need not necessarily have the same type. The operator is defined if the range type of the first relation matches the domain type of the second. Moreover, `compAL` returns as result a relation of another type. This definition therefore cannot be included in the class `relation`.

Mixing parametric polymorphism and ad-hoc polymorphism together is a problem. When an operator is defined for a particular relation representation it should work uniformly on that representation regardless of the type of the relation being represented. For example, composition should be defined for

two relations whenever the range type of one is the same as the domain type of the other. This is parametric polymorphism. The different definitions of a relational operator should also be grouped together as a single overloaded function that applies regardless of the argument representations. For example, the composition operator symbol, (;), should be applicable to relations irrespective of their representation. This is ad-hoc polymorphism.

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. Ad-hoc polymorphism is gained at the expense of losing parametric polymorphism.

4.4 Representation Selection by Abstract Interpretation

4.4.1 What is Abstract Interpretation?

Hughes [46] states that *abstract interpretation* (or *forwards analysis*) consists of running a program with partial information about its inputs to derive partial information about its result. A forwards analyser starts with information about the values of variables occurring in each subexpression and propagates it upward through the syntax tree to derive information about the expression as a whole. In contrast to this, *backwards analysis* derives contextual information about a program by running it backwards. A backwards analyser starts with information about the context of the entire expression and propagates it downwards through the syntax tree to the leaves to derive information about the contexts in which subexpressions occur. Forwards analysis uses synthesised attributes while backwards analysis uses inherited ones. Forwards and backwards analyses have been much used in optimising of compilers for declarative languages. This section describes how a combination of forwards and backwards analysis can be used to select representations for all definitions, expressions and subexpressions in a Drusilla program.

4.4.2 Preliminary Definitions

The Abstract Domain

Some form of denotation is required for the relation representation information being propagated around the syntax tree. It is usual to use elements of an *abstract domain*, which are called *abstract values* since they represent partial information

about the values of expressions. The use of a domain rather than a set guarantees that recursive definitions all have a solution.

The elements of the abstract domain are not single representations but *sets* of representations because, as section 4.2 concluded, no representation is always ideal. The relation representations — association list, set-valued function and Boolean-valued characteristic function — are denoted respectively by the abstract elements AL, SF and CF. Basic values (numbers, strings, unit element) are denoted by the abstract element, DV. For each expression the set of feasible representations is considered. The empty set of representations allows reasoning about expressions and subexpressions that have no possible representation. The abstract domain is the lattice presented Figure 4.1.

Abramsky and Hankin [1] state that domains are partially ordered sets $(D,\sqsubseteq)$ with least element $\perp_D$ such that

$$\perp_D \sqsubseteq d, \forall\, d \in D$$

In the representation domain the partial ordering is the superset ordering and the least element is the universal set of representations. More formally:

$$x \sqsubseteq y \Leftrightarrow y \subseteq x$$
$$\perp_D = \{AL,SF,CF,DV\}$$

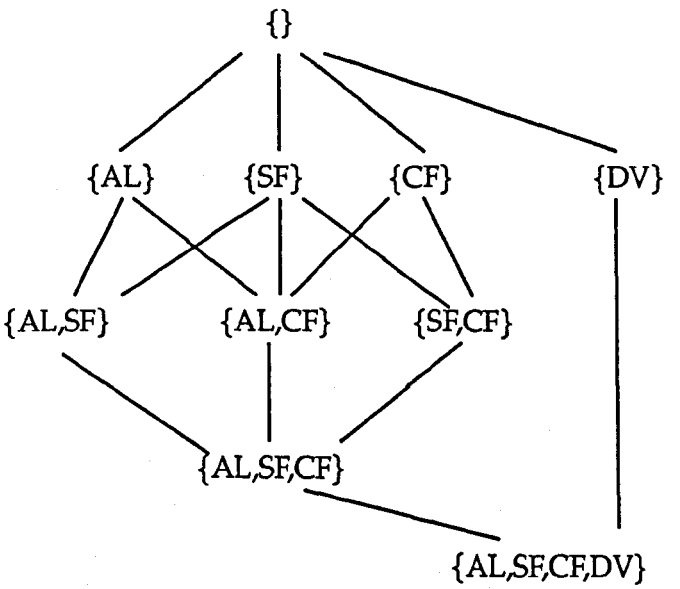

Figure 4.1: The abstract domain for representation analysis

The Abstract Syntax Tree

Figure 4.2 defines the abstract syntax of Drusilla expressions through which the abstract values will be propagated. For simplicity all compound relation designations are defined as an operator applied to a subexpression. If the operator

is binary then the two subexpressions it 'glues' together are tupled to form one subexpression. Relation specialisations are also represented as compound designations:

specialOp (rel × tuple)

Operator *specialOp* is a system operator that is opaque to the programmer, *rel* is the relation being specialised and *tuple* is the specialising tuple.

ast → *value* | *tuple* | *elementary* | *compound*
value → *number* | *string* | *unit*
tuple → (*ast* × ... ×*ast*)
elementary → *operator* | *userDefn* | *formPar* | *relExtension*
compund → *funcOperator ast*

Figure 4.2: Abstract syntax of Drusilla expressions

4.4.3 Analysis of an Expression

Representations for an expression and all its subexpressions are selected by using a combination of forwards and backwards analysis. Forwards analysis generates the possible representations for the expression by propagating abstract values of subexpressions up the abstract syntax tree. Backwards analysis selects actual representations from those possible propagating the context down the tree to subexpressions. For simplicity it is assumed that the expression is known to be calculus type correct.

Forwards Analysis of an Expression

The abstract interpretation function, `abstInt`, shown in Figure 4.3, returns the set of representations possible for a given expression. Values in the abstract domain are denoted by a hash symbol superscipt. The representations available for a compound designation `op arg` depends on the possible definitions of `op` and the representations possible for `arg`. The function `repMap` returns the representations possible for a given compound designation — it encapsulates the argument-result representation constraints for each operator. When applied to a given designation, the function `elementaryRep`, returns the set of representations possible for that designation.

The analysis depends on satisfaction of the *elementary representation condition*: the representations of all elementary designations — Drusilla operators, formal parameters and program definitions — must be known. Basic numerical operators are represented by set-valued and characteristic functions and the relational operators by set-valued functions. The relational operators *cannot* be represented by characteristic functions since this would require equality to be defined

```
abstInt value = {DV}
abstInt (ast_1 × ... × ast_n) =
  {(ast_1^# ×^# ... ×^# ast_n^#) | ast_i^# ∈ abstInt ast_i; 1 ≤ i ≤ n}
abstInt relExtension = {AL}
abstInt elementary = elementaryRep elementary
abstInt funcOp ast = {repMap funcOp ast^# | ast^# ∈ abstInt ast}
```

Figure 4.3: Abstract interpretation function

for their results, which may be represented by functions. Formal parameters are given synthetic representations. The program is analysed in such a way that each definition is analysed before the definitions that reference it.

Backwards Analysis of an Expression

Backwards analysis is applied to the expression for each possible representation generated by the forwards analysis. For a chosen expression representation the context propagated to the subexpressions are those representations that make the expression representation possible. The backwards analyser context function is defined:

$$context\ op\ expRep\ argReps = x, \quad x \in argReps \wedge repMap\ op\ x = expRep$$

The arguments given to the context function are: the operator forming the compound relation designation, `op`, the required expression representation, `expRep` and the set of possible subexpression representations, `argReps`. It extracts from `argReps` one representation that permits `expRep` to be generated by using function `repMap`, to calculate the representation context.

Once backwards analysis has been performed for each possible expression representation, the expression analysis is complete. The result is a set of syntactically identical expression versions; each of which has its own unique representation.

4.4.4 Analysis of a Definition

The expression analysis technique can also be applied to definitions. Chapter 3 identified three forms of Drusilla definition: binding and fully parameterised and partially parameterised. The representations possible for a binding definition are simply the representations possible for the bound expression.

The analysis of parameterised definitions is more complex — representations for the formal parameters must first be generated to satisfy the elementary representation condition. Hughes [46] observes that abstract interpretation often places constraints on variables or allows analysis by interpreting with given abstract values for the formal parameters. Therefore, all possible representation assumptions are made for each parameter, and forwards and backwards analysis

is performed for each assumption. For example, consider the definition of the cartesian product:

```
(r,s) cartProd (r ; inv s)
```

If `r` and `s` are two sets (relations between the set elements and the unit value) then this defines their cartesian product. The set of all possible assumptions is:

$$\{(rRep,sRep) \mid rRep \in \{AL,SF,CF,DV\};\ sRep \in \{AL,SF,CF,DV\}\}$$

The forwards analyser uses these assumptions as synthesised abstract values so that every elementary designation has a representation. The defining expression is then analysed to generate a set of possible representations, each of which depends on specific formal parameter representations. The backwards analyser again selects representations from those possible and, for each representation, places the necessary representation constraints on the formal parameters. For some of the parameter assumptions the expression may have no possible representation; this is denoted by the empty set of representations. This reflects constraints as to the representations parameters *cannot* have. The definition can then be assigned the set of representations possible for its defining expression. For example, the analysis of `cartProd` is shown in Table 4.1. In this example the backwards analysis has only one representation choice at each node because no elementary designation in the expression has more than one representation.

r	s	inv s	r ; inv s
{AL}	{AL}	{AL}	{AL}
{AL}	{SF}	{CF}	{CF}
{AL}	{CF}	{CF}	{CF}
{AL}	{DV}	{}	{}
{SF}	{AL}	{AL}	{SF}
{SF}	{SF}	{CF}	{CF}
{SF}	{CF}	{CF}	{CF}
{SF}	{DV}	{}	{}
{CF}	{AL}	{AL}	{CF}
{CF}	{SF}	{CF}	{}
{CF}	{CF}	{CF}	{}
{CF}	{DV}	{}	{}
{DV}	{AL}	{AL}	{}
{DV}	{SF}	{CF}	{}
{DV}	{CF}	{CF}	{}
{DV}	{DV}	{}	{}

Table 4.1: Abstract interpretation of `(r,s) cartProd r ; inv s`

4.4.5 Analysis of Recursive Definitions

Recursive and mutually recursive definitions are analysed using the standard abstract interpretation technique of fixpointing. This technique has theoretical justification in the fixed point theorem (see Abramsky and Hankin [1]) which states that every continuous function over an abstract domain has a least fixed point. In other words a set of mutually recursive definitions can be expressed as the fixpoint of a higher-order operator:

$$< f^{\#}, \ldots, f^{\#} > = \mathrm{F} < f^{\#}, \ldots, f^{\#} >$$

for some F so

$$< f^{\#}, \ldots, f^{\#} > = \mathrm{fix}\, \mathrm{F}$$

and the fixed point is the limit of an ascending chain:

$$< f^{\#}, \ldots, f^{\#} > = \bigsqcup_{i=0}^{\infty} F^i \perp$$

If this chain is finite then its limit can be used to calculate the fixed point. For representation selection this chain must be finite because the abstract domain is finite.

The representation fixed point is calculated iteratively for each group of mutually recursive definitions. Initially each definition is assigned the 'bottom' representation value — the universal set of representations. Forwards analysis is then applied to each definition, as described in section 4.4.4, to produce a new set of possible representations. The new representation set for each definition is an approximation to its 'real' set of possible representations and is used as the assumption for the next iteration of the analysis. This iteration continues until the least fixed point limit is reached, when, the representation set assumed for each definition is the same as the representation set possible under those assumptions. When this fixed point is reached the sets of representations *possible* for each definition are known. Actual representations are then selected by applying backwards analysis to each definition as described in section 4.4.4.

4.4.6 Analysis of a Drusilla Program

If the elementary representation condition is to be satisfied then analysis of a program must ensure that:

- No definition is analysed until all the definitions it refers to have been analysed.
- Groups of mutually recursive definitions are analysed as an entity as described in section 4.4.5.

To preserve these conditions the program is partitioned into groups of maximally strong components with respect to the program reference graph. This is a standard technique described in Peyton-Jones' book [82]. When this partitioning is complete each maximally strong component contains a group of mutually recursive definitions. The reduced program graph is acyclic; i.e. there exist no mutually recursive definitions outside these components.

The first condition is maintained by analysing the strong components in the reverse of their depth first search ordering (with respect to the reduced call graph). The second condition is maintained by analysing the definitions in each strong component together using the technique described in section 4.4.5.

4.5 Failings of the Abstract Interpretation

This approach has failings in terms of both efficiency and quality of the data generated. There are three main causes of inefficiency:

1. The need to analyse each expression tree twice — an upward pass followed by a downward pass — seems excessive; analysis based on one pass would be more desirable.

2. For recursive definitions iteration must be used to find fixpoints. This is a standard abstract interpretation technique but is expensive in terms of computation.

3. Representations for formal parameters must be synthesised. The cartesian product of all possible representations must be taken and forwards analysis applied for each representation.

The criticism concerning the quality of data produced is far more fundamental. The interpretation scheme generates possible representations for relation definitions within a program that is known to be calculus type correct. This information alone is insufficient: analysis of higher-order definitions reveals nothing about the representations possible for value-relations.

This has horrific consequences for operational behaviour of programs. For example, consider an expression which is the composition of two higher-order relations `r ; s` where the range of `r` is a relation and the domain of `s` is a relation. For the expression to be calculus type correct the two value-relations must possess the same type. For example, suppose the types of `r` and `s` are:

$$\mathtt{r} :: A \leftrightarrow (B \leftrightarrow C)$$
$$\mathtt{s} :: (E \leftrightarrow F) \leftrightarrow D$$

The composition expression will have a type if and only if the types of the value-relations can be unified. If they can be then the expression type will be:

$$\mathtt{r;s} :: A \leftrightarrow D$$

For this composition to be implementable calculus type consistency of the two relations is not sufficient — they must also possess the same representation. The implementation of composition requires either `r` to generate range values for `s` to consume in its domain or `s` to generate domain values for `r` to consume in its range. The value-relations are communicated between `r` and `s`. Therefore, it is necessary to ensure consistency of value-relation representation. For example, if `s` requires the relation in its domain to have an extensional representation then the relation in the range of `r` must also have an extensional representation. Such consistency is not enforced by the abstract interpretation and without it many programs found to be calculus type correct and successfully analysed can 'go wrong' at run-time. This is exactly the problem a Milner [75] type system is intended to prevent.

4.6 A Type System for Representation Selection

4.6.1 Why Use a Type System?

The static polymorphic typing of functional languages is adopted in Drusilla through the calculus types introduced in chapter 3.2.2. Such a type system should prevent a program from going wrong at run-time. However it was observed in section 4.5 that a program found calculus type correct could still 'go wrong' under the abstract interpretation approach to alleviating the representation bottleneck. This crucial failing suggests that some new form of type system is required — one that ensures that programs found type correct do not produce representation inconsistencies between relations on program execution.

The notion that a type system would be well suited to this task is reinforced by comments from Cardelli and Wegner [20]:

> *'A major purpose of type systems is to avoid embarrassing questions about representations and to forbid situations in which these questions might come up.*
>
> ...
>
> *A type may be viewed as a set of clothes (or a suit of armour) that protects an underlying representation from arbitrary or unintended use. It provides a protective covering that hides the underlying representation and constrains the way object may interact with other objects. In an untyped system untyped objects are* naked *in that the underlying representation is exposed for all to see. Violating the type system involves removing the protective set of clothing and operating directly on the naked representation.'*

In MacLennan's RPL the programmer needs to view the underlying representations of relations in order to know which operators are applicable. The aim of the Drusilla system is to cover naked relation representations with a suit of clothes that better preserves the relational abstraction. This would suggest the need for a type system.

A type system is further suggested by the example used in the criticism of the abstract interpretation approach. Suppose a program contains two definitions r and s whose calculus types can be inferred statically, using Milner polymorphism:

$$\mathtt{r} :: A \leftrightarrow (B \leftrightarrow C)$$
$$\mathtt{s} :: (E \leftrightarrow F) \leftrightarrow D$$

A type can be inferred for the composition of r and s by forcing the elements in the range of r and the elements in the domain of s to have the same type:

$$\mathtt{r};\mathtt{s} :: A \leftrightarrow D$$

A substitution for type variables *B*, *C*, *E*, and *F*, called the most general unifier, is generated by the type inference algorithm. This substitution ensures that *B* and *E* denote the same type as do *C* and *F*. This type information is generated using the rule:

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

The type system infers type constraints for relation domain and range values. The abstract interpretation should infer similar representation constraints but does not — representations are assigned to relations but not to domain and range values. A type system is a mechanism that can infer type constraints; there appears to be no reason why the same mechanism should not be used to infer representation constraints.

4.6.2 Overview of Typed Representation Analysis

The *typed representation system* of Drusilla is a generalisation of Milner's type inference system [75] that infers ad-hoc as well as parametric polymorphism. Milner's type system, as used in modern functional languages, normally infers a type for every expression and every subexpression within a program. Typed representation inference in Drusilla infers not only a calculus type but also a suitable representation for every expression and subexpression. The analysis not only ensures a program is type correct but also automatically selects representations. This section describes how the typed representation inference rules are derived from calculus type inference rules, and how expressions, definitions, and programs are analysed.

4.6.3 From Calculus Types to Representation Types

Objects in Drusilla must be mapped to the chosen implementation language, Miranda. Drusilla values of basic type (num, string, un, tuples) can be trivially mapped to the corresponding Miranda values. A number becomes that Miranda number, a character string becomes a list of Miranda characters and the `Unit` value a Miranda algebraic data type value `Unit`. Chapter 6 describes how a tuple can be represented as a list of Miranda values by defining the universe of Drusilla values as an algebraic data type. Values of basic type all have one,

natural Miranda representation. By contrast a relation (value of relational type) can map into any one of three possible representations: association list, set-valued function, or characteristic function. The representation selection problem is to decide a representation for any given value of relational type. A framework for this selection process is created by extending calculus types, adding information about representation when the type is relational. This extends the process of calculus type inference to that of *typed representation inference.*

Definition of the Typed Representation Universe

The typed representations of basic values are as for calculus types: *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* denotes a relation representation 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. The inclusion of typed representation information for relation domain and range values ensures that value-relations are given typed representations.

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.

Typed Representation Inference Rules

Each Drusilla primitive operator has one associated calculus type inference rule which forms the basis for its *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. For example, the calculus type inference rule for the relation composition operator

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

is combined with the representation constraints for relation composition, shown in Figure 4.4, to form the set of typed representation inference rules presented in Figure 4.5. Each typed representation rule reflects the natural computational constraints of type and representation for composition.

Representation Coercions

Representation coercions depend on equality being defined for certain values, as explained in section 4.2. Three of the typed representation inference rules for

r ; s	s: AL	s: SF	s: CF
r: AL	AL	AL	CF
r: SF	SF	SF	CF
r: CF	CF	⊥	⊥

AL for Association List
SF for Set-valued Function
CF for Characteristic Function

Figure 4.4: Representation constraints for relation composition

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

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

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

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

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

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

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

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

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

Figure 4.5: Typed representation inference rules for relation composition

relation composition correspond to definitions that coerce an association list into a set-valued function. Such coercions are necessary for relations to be used in forward mode. A set-valued function can only be coerced into an association list if equality is defined for its domain values. In typed representation inference rules *equality type variables* are used to incorporate the equality constraints necessary for representation coercions. Equality variables are described in section 4.6.5.

4.6.4 Representation Analysis of a Relation Extension

An expression denoting a relation defined in extension takes the form:

```
{("Plus",[+]),("Minus",[-]),("Times",[*]),("Divide",[/])}
```

Such an expression is naturally represented by the corresponding association list so analysis should produce a typed representation of the form:

$$\text{AL[domTR} \leftrightarrow \text{rngTR]}$$

To generate this the domain and range elements must both have one consistent typed representation. This means that if the relation is higher-order, for example if the range elements are relations, then the subrelations in each element must not only have the same type but also the same representation. The above expression, for example, has the valid typed representation:

$$\text{AL[string} \leftrightarrow \text{SF[(num} \times \text{num)} \leftrightarrow \text{num]]}$$

4.6.5 Typed Representation Analysis of an Expression

The typed representation inference algorithm is similar to standard polymorphic typecheckers, as described by Cardelli [19] and Reade [85]. When operators or user-defined relations are used as elementary designations they may be polymorphic and as such can have a different instance of that polymorphic type at each occurrence. The typed representations of such designations are called *type schemes* and the type variables associated with their typed representations called *generic*. Genericity is preserved by *refreshing* the polymorphic type scheme at each occurrence of the designation, substituting new type variables for old. Similarly the typed representations of operators must be refreshed when they form compound designations.

The algorithm begins with the leaves of the expression tree and proceeds left to right and bottom-up moving toward the root node. The non-leaf nodes of the tree are compound designation which consist of an operator applied to one subexpression. Whenever an operator node is reached each of its typed representation inference rules is, in turn, refreshed and applied to the subexpression. Each application of an inference rule involves unification which generates a new 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 tree and hence to the relations those variables are attached to.

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.

The algorithm, $\mathcal{TR}$, for typed representation analysis of an expression is presented in Miranda in Figure 4.9. The abstract syntax of expressions and typed representations processed are defined as algebraic datatypes in Figure 4.6 and Figure 4.7 respectively. The substitution operations for typed representation variables, defined in Figure 4.8, are simplified for presentation by removing the state variable used in the real implementation to store the typed representations of definitions and parameters and an infinite supply of type variables used when refreshing polymorphic types. The algorithm references several basic functions that are not defined here:

lookupRel takes as argument the name of a programmer-defined relation and returns as result the set of typed representations associated with that relation.

lookupParTR takes as argument the name of a formal parameter and returns as result the set of typed representations associated with that parameter.

polyAssList returns the typed representation of a completely polymorphic association list

```
RTR AL (TRvar (Alpha x)) (TRvar (Alpha y))
```

where x and y are unique type variables.

opRules takes as argument a Drusilla operator and returns as result the set of typed representations associated with that operator. Each typed representation can be used as an inference rule when that operator forms a compound designation. For example:

```
op arg
```

Each typed representation for op is of the form *RTR SF domTR rngTR*. Let *argTR* denote a typed representation of arg. If *argTR* and *domTR* unify with substitution σ then the expression op arg has a typed representation $\sigma rngTR$.

```
relExp ::=
    No num                                  number values
  | St [char]                               string values
  | Ut                                      unit values
  | Tu [relExp]                             tuple of expressions
  | Rel ident typedRep                      user defined relation
  | Fpar ident typedRep                     formal parameter
  | RO relOp typedRep                       elementary designation operator
  | Ext [(relExp,relExp)] typedRep          relation defined in extension
  | E relOp typedRep relExp                 compound relation designation
```

Figure 4.6: The abstract syntax of expressions

```
typedRep ::=
    Num                                     number typed representation
  | String                                  string typed representation
  | Un                                      unit typed representation
  | TRvar trVar                             variable typed representation
  | TuTR [typedRep]                         tuple typed representation
  | RTR relRep typedRep typedRep            relational typed representation
  | BotTR                                   undefined typed representation

trVar ::=  Alpha num                        normal typed representation variables
         | Beta num                         equality typed representation variables

relRep ::= AL | SF | CF                     relation representations
```

Figure 4.7: Algebraic datatype defining typed representation abstract syntax

```
subst ::= Sub (trVar -> typedRep)

abstype trSub
with
        idSub :: trSub
        newSub :: trVar -> typedRep -> trSub
        applySubToTR :: trSub -> typedRep -> typedRep
        applySubToExp :: trSub -> relExp -> relExp
        composeSubs :: trSub -> trSub -> trSub

trSub == subst

idSub = Sub TRvar

newSub i t = Sub (assocMl [(i,t)] TRvar)

applySubToTR (Sub f) (TRvar i) = f i
applySubToTR subs (RTR rep t1 t2) =
   RTR rep (applySubToTR subs t1) (applySubToTR subs t2)
applySubToTR subs (TuTR tup) =
   TuTR (map (applySubToTR subs) tup)
applySubToTR subs other = other

applySubToExp sub (Rel name typRep) =
   Rel name (applySubToTR sub typRep)
applySubToExp sub (Fpar name typRep) =
   Fpar name (applySubToTR sub typRep)
applySubToExp sub (E op typRep arg) =
   E op (applySubToTR sub typRep) (applySubToExp sub arg)
applySubToExp sub (RO op typRep) =
   RO op (applySubToTR sub typRep)
applySubToExp sub (Tu tup) =
   Tu (map (applySubToExp sub) tup)
applySubToExp sub (Ext rel typRep) =
   Ext (map (applyPair sub) rel) (applySubToTR sub typRep)
applySubToExp sub other = other

composeSubs sub1 (Sub f) = Sub (applySubToTR sub1 . f)

applyPair :: trSub -> (relExp, relExp) -> (relExp, relExp)
applyPair sub (x,y) =
   (applySubToExp sub x, applySubToExp sub y)

assocML :: [(*,**)] -> (* -> **) -> * -> **
assocML assList assFun z =
   hd [y | (x,y) <- (assList ++ [assFun z]); x = z]
```

Figure 4.8: Substitution operations for typed representations

```
fullTypeCheckExp :: trSub -> relExp -> [(trSub,relExp)])
fullTypeCheckExp subX expX =
 [(subY,applySubToExp subY expY) |
 (subY,expY) <- typeCheckExp subX expX]

typeCheckExp sub (Rel name typRep) =
 [(sub,Rel name relTR) | relTR <- lookupRelTR name]
typeCheckExp sub (Fpar name typRep) =
 [(sub,Fpar name parTR) | parTR <- lookupParTR name]
typeCheckExp sub (E op typRep arg) =
 [(newSub,newExp) | (argSub,newArg) <- fullTypeCheckExp sub arg;
 (RTR SF lhsTR rhsTR) <- opTypeRep op; argTR <- [getExpTR arg]
 (succ,opSub) <- [unify lhsTR argTR]; succ = True;
 newSub <- [composeSubs opSub argSub];
 newExp <- [applySubToExp newSub (E op rhsTR newArg)]]
typeCheckExp sub (RO op typRep) =
 [(sub,RO op opTR) | opTR <- opRules op]
typeCheckExp sub (Tu tup) =
 [(newSub,Tu newTup) |
 (newSub,newTup) <- foldr putExpInList [(sub,[])] tup]
typeCheckExp sub (Ext reln tr) =
 foldr analExtRel [(sub,Ext [] polyAssList)] reln
typeCheckExp sub other = [(sub,other)]

putExpInList exp subsExps =
 [(newSub,newExp : expList) | (listSub,expList) <- subsExps;
 (newSub,newExp) <- fullTypeCheckExp listSub exp]

analExtRel (dom,rng) subsExts =
 [(newSub,newExt) | (extSub,Ext reln extTR) <- subsExts;
 (elSub,[newDom,newRng]) <- tcExpList extSub [dom,rng];
 domTR <- [getExpTR dom]; rngTR <- [getExpTR rng]
 (succ,newExtSub) <- [unify (RTR AL domTR rngTR) extTR];
 succ=True; newSub <- [composeSubs newExtSub elSub];
 newExt = applySubToExp newSub (Ext ((dom,rng) : reln) extTR)]

getExpTR (No n) = Num
getExpTR (St b) = String
getExpTR Ut = Un
getExpTR (Tu tup) = TuTR (map getExpTR tup)
getExpTR (Rel name typRep) = typRep
getExpTR (Fpar name typRep) = typRep
getExpTR (Ext rel typRep) = typRep
getExpTR (RO op typRep) = typRep
getExpTR (E op typRep arg) = typRep
```

Figure 4.9: Algorithm for typed representation analysis of an expression

Unification For Typed Representation Inference

Unification for typed representation analysis is based on the normal unification algorithm [88] used in conventional typecheckers, but is complicated by the presence of equality type variables.

A typed representation is called an *equality type* if and only if equality is defined for all values belonging to that typed representation. The *basic typed representations, num, string* and *un,* are all equality types. A tuple has an equality type if and only if every element within that tuple has an equality type. A relational typed representation

$$\text{rep}\ [\text{domTR} \leftrightarrow \text{rngTR}]$$

is an equality type if and only if `rep` is an association list and `domTR` and `rngTR` are equality types. Every equality variable has equality type. An ordinary type variable is not of equality type but can be changed to an equality variable which does have equality type. Such a change is called a *type variable coercion.* Some typed representations can be made equality types by coercing the variables used in those structures.

For example, the empty relation, when represented by the empty association list,has the calculus type:

$$\text{AL}[X \leftrightarrow Y]$$

This typed representation is not an equality type because the domain and range typed representations are not of equality type, although it should be as equality is defined for any empty list. It can be made an equality type by coercing the type variables:

$$AL[X^{=} \leftrightarrow Y^{=}]$$

This method for making typed representations equality types is called *type restriction* and the algorithm is presented in Figure 4.10. The unification algorithm, $\mathcal{U}$, is presented in Figure 4.11. The rules for deciding whether two typed representations unify are:

- Two basic types unify if and only if they are the same type.
- Two tuples unify if and only if they have the same number of elements and all corresponding elements unify.
- Two relational typed representations may unify if and only if they have the same representation and their respective domain and range typed representations unify.
- An ordinary type variable may unify with a typed representation if and only if that variable does not occur within the typed representation. This is the normal 'occurs' check of unification. If a normal variable unifies with an equality variable then the substitution generated maps the normal variable into an equality variable.

- An equality variable may unify with a typed representation if and only if that typed representation is an equality type. If it is not then the unification algorithm attempts to make it an equality type by *type restriction.*

```
mkEquality :: typedRep -> (bool,trSub)
mkEquality Num = (True,idSub)
mkEquality String = (True,idSub)
mkEquality Un = (True,idSub)
mkEquality (TRvar trVar) = (True,mkVarEquality trVar)
mkEquality (TuTR tup) = mkListEquality tup
mkEquality (RTR AL dom rng) = mkListEquality [dom,rng]
mkEquality (RTR SF dom rng) = (False,idSub)
mkEquality (RTR CF dom rng) = (False,idSub)
mkEquality BotTR = (False,idSub)

mkVarEquality :: trVar -> trSub
mkVarEquality (Alpha var) =
   newSub (Alpha var) (TRvar (Beta var))
mkVarEquality (Beta var) = idSub

mkListEquality :: [typedRep] -> (bool,trSub)
mkListEquality = foldr mkListEqualityB (True,idSub)

mkListEqualityB :: typedRep -> (bool,trSub) -> (bool,trSub)
mkListEqualityB typRep (False,sub) = (False,idSub)
mkListEqualityB typRep (True,subX) =
   (succ,composeSubs subY subX)
   where
   (succ,subY) = mkEquality (applySubToTR subX typRep)
```

Figure 4.10: Substitution operation to enforce equality

4.6.6 Representation Analysis of a Definition

The typed representation inference algorithm can also be used to generate a set of typed representations for a Drusilla definition.

Analysis of a Binding Definition

The set of typed representation for a binding definition is simply the set of typed representations for its defining expression. For example, consider the definition of '2x + y':

```
doublePlus = ([2 *] || id) ; [+]
```

The expression and definition are both given the typed representation set:

doublePlus :: {SF[(num × num) ↔ num]}

```
unify :: typedRep -> typedRep -> (bool, trSub)
unify BotTR BotTR = (True,idSub)
unify BotTR other = (False,idSub)
unify other BotTR = (False,idSub)
unify (TRvar (Beta v)) (TRvar (Beta x))
   = (True, idSub),                                    v = x
   = (True, newSub (Beta v) (TRvar (Beta x))),         otherwise
unify (TRvar (Alpha v)) (TRvar (Alpha x))
   = (True, idSub),                                     v = x
   = (True, newSub (Alpha v) (TRvar (Alpha x))),       otherwise
unify (TRvar (Alpha v)) x =
   (~ (occurs (Alpha v) x), newSub (Alpha v) x)
unify x (TRvar (Alpha v)) =
   (~ (occurs (Alpha v) x), newSub (Alpha v) x)
unify (TRvar (Beta v)) x =
   (eqExists & noOccurs, composeSubs (newSub (Beta v) newX) sub)
   where
   (eqExists,sub) = mkEquality x
   noOccurs = ~ (occurs (Beta v) newX)
   newX = applySubToTR sub x
unify x (TRvar (Beta v)) =
   (eqExists & noOccurs, composeSubs (newSub (Beta v) newX) sub)
   where
   (eqExists,sub) = mkEquality x
   noOccurs = ~ (occurs (Beta v) newX)
   newX = applySubToTR sub x
unify (TuTR tupX) (TuTR tupY) =
   foldr2 tupUnify (True, idSub) tupX tupY
unify (RTR repX domX rngX) (RTR repY domY rngY) =
   (repX = repY & succ1 & succ2, composeSubs subY subX)
   where
   (succ1, subX) = unify domX domY
   (succ2, subY) = unify (applySubToTR subX rngX)
                         (applySubToTR subX rngY)
unify otherX otherY = (otherX=otherY, idSub)

tupUnify :: typedRep -> typedRep -> (bool,trSub) -> (bool,trSub)
tupUnify x y (succ1, subX) =
   (succ1 & succ2, composeSubs subY subX)
   where
   (succ2,subY) = unify (applySubToTR subX x)
                        (applySubToTR subX y)
```

Figure 4.11: Typed representation inference unification algorithm

Analysis of a Fully Parameterised Definition

The formal parameters of a (fully or partially) parameterised definition must initially be associated to unique type variables in the same way that λ-bound variables are in conventional functional language typecheckers. These type variables are *non-generic*, they must denote the same instance of the polymorphic type at each occurrence and, unlike generic type variables, must not be refreshed. Analysis of the defining expression then generates a set of type variable substitutions and typed representations for that expression. The substitution associated with each typed representation also determines the typed representation of each formal parameter. The typed representations for the definition are generated from the typed representations of the defining expression. For example the definition of 2x + y:

```
(x,y) doublePlus  2 * x + y
```

Such a definition may be given a set of typed representations of the form:

$$SF\ [paramsTR \leftrightarrow expTR]$$

Here `paramsTR` is the type of the parameter tuple and is obtained by applying the typed representation substitution to the type variables associated with the parameters. The typed representation for the expression is `expTR`. The representation is a set-valued function because the definition produces the range value `2 * x + y` from the domain value `(x,y)`. In our example the typed representations would be as follows:

`(x,y)`	:: (num × num)
`(2 * x + y)`	:: num
`doublePlus`	:: {SF [(num × num) ↔ num]}

Analysis of a Partially Parameterised Definition

Analysis of the defining expression is as for a fully parameterised definition — all parameters, named and anonymous, are associated to unique type variables. The defining expression must be relational and hence have a set of typed representations of the form:

$$rep\ [domTR \leftrightarrow rngTR]$$

The type variable substitution associated with each typed representation is extended by unifying the expression domain type, `domTR`, with the tuple of anonymous parameters. This new substitution is applied to the domain tuple to create a typed representation for the definition of the form:

$$newRep\ [newDomTR \leftrightarrow rngTR]$$

If the representation of the defining expression, `rep`, is a set-valued or characteristic function representation then the representation of the definitions, `newRep`, is the same. However, if `rep` is an association list and `domTR` is an equality type

then newRep is a set-valued function. The definition cannot be represented by an association list since it depends on formal parameters. When the definition is used the representation of the defining expression must be coerced to either a set-valued or characteristic function. Therefore, the definition representation, newRep, is made a set-valued function, which, if necessary, can be coerced into a characteristic function. If rep is an association list and domTR is not an equality type then the definition is unusable and has no typed representation. Here newDomTR is the typed representation of the domain tuple and rngTR is the same as before. For example, another definition of '2x + y':

```
(_,y) doublePlus  [2 *] ; [+ y]
```

The typed representations will be as follows:

```
([2 *] ; [+ y])   :: SF [num ↔ num]
       y          :: num
     (_,y)        :: (num × num)
 doublePlus       :: {SF [(num × num) ↔ num]}
```

4.6.7 Representation Analysis of Recursive Definitions

If recursive and mutually recursive definitions are not analysed separately from the rest of the program then some polymorphism may be lost and it might not be possible to typecheck the program at all. (For an explanation of this, see Mycroft [78].)

Typed representation analysis is applied to a group of recursive definitions in the same way a conventional functional language typechecker is applied to a group of recursive definitions or a 'let-rec' expression. The description of this intricate process is based on Hancock's description of typechecking 'let-recs' in Peyton-Jones' book [82]:

1. Associate new type variables with the definitions in the group. These variables are non-generic — all occurrences of a defined name on the right-hand side of a recursive definition should have the same type.

2. The definitions are analysed as described in section 4.6.6. If successful, this will yield a set of lists of typed representations. Within each list there is one typed representation for each definition. Each list is associated with a substitution for type variables that is applicable to all typed representations within that list. It is crucial that within a component each definition sees one consistent typed representation for all other definitions.

3. Within each list unify each definition's typed representation with the typed representation of the corresponding type variable. The right-hand side of each recursive definition must have the same type as its corresponding variable. Should the unification succeed, that constraint can be met.

4.6.8 Representation Analysis of Programs

Program analysis is simplified if two conditions are satisfied:

- No definition is analysed until all the definitions it refers to, outside its mutually recursive group, have been analysed.
- Groups of mutually recursive definitions are analysed as an entity as described above in section 4.6.7.

These conditions are the same as for the abstract interpretation approach. They are preserved by partitioning the program into groups of maximally strong components with respect to the program call graph. The first condition is preserved by analysing the components in the reverse of their depth first search ordering, with respect to the reduced call graph. The second condition is maintained by analysing the definitions in each component together as a single entity, as is customary in polymorphic type checkers, in the manner described in section 4.6.7.

4.6.9 Correctness and Completeness

The algorithm, $\mathcal{TR}$, shown in Figure 4.9, for inferring sets of typed representations for expressions, is based on an inference system (the sets of typed representation rules for operators) for inferring typed representations. This section has two aims:

1. To show $\mathcal{TR}$ is *correct* (*sound*) in the sense that the set of typed representations it yields is derivable in the inference system.
2. To show $\mathcal{TR}$ is *complete* in the sense that any typed representation derivable for an expression is an instance of that computed for $\mathcal{TR}$.

Algorithm $\mathcal{TR}$ is a generalisation of Milner's [75] in which each expression construct (operator) has a set of inference rules rather than just one. Therefore, the result of applying it to an expression E is a set:

$$\{(S_1, E_1), \ldots (S_n, E_n)\}$$

where each E_i, $1 \leq i \leq n$, is syntactically identical to expression E but has its own unique typed representation and substitution, S_i, for typed representation variables. Each (S_i, E_i) is produced by applying Milner's algorithm to E, selecting one applicable inference rule for each expression construct. The correctness and completeness proofs for Milner's algorithm given by Damas and Milner [25] therefore hold for algorithm $\mathcal{TR}$.

The soundness and completeness results have very practical ramifications for the Drusilla system. The set of typed representation inference rules for each operator should reflect what is computationally feasible for that operator. The completeness result states that if there exists some possible representation for an expression then the algorithm will find that representation. In this sense typed representation inference reflects the limit of computability for relational operators. However, this is not the same as full computational completeness as, for example, we disallowed enumeration of domain and range values for relations represented by characteristic functions.

4.6.10 Typed Representation Inference Examples

Figure 4.12 shows the typed representations inferred by the Drusilla system for the program in Figure 3.17. Definition `nats` can be given either a association list or a set-valued function representation since its defining expression may be viewed as a formula for generating the natural numbers or as a set-valued function relating `Unit` to the natural numbers. Definition `sq` is represented by a set-valued function because it is fully parameterised. Definition `squares` is represented by either an association list or a set-valued function depending on the representation of `nats`.

```
nats :: AL[un ↔ num]
nats :: SF[un ↔ num]

sq :: SF[num ↔ num]

squares :: AL[num ↔ num]
squares :: SF[num ↔ num]
```

Figure 4.12: Typed representations inferred for `nats` program

Figure 4.13 shows the typed representations inferred by the Drusilla system for the program in Figure 3.14. The definitions `combK` and `combI` combinators are represented by set-valued functions because they are fully parameterised. The S-combinator, `combS`, by definition is higher-order because it contains two relations in its domain. Consequently it has a larger set of typed representations — one for each possible value-relation representation. This representation set shows illustrates that this definition corresponds to several different definitions in RPL — one for each value-relation representation.

4.7 Comparison of Representation Selection Approaches

Section 4.5 described the fundamental flaws of the abstract interpretation approach to representation selection described in section 4.4. Section 4.6 detailed an alternative approach — a type system which not only ensures a Drusilla program is type correct, but also selects representations for definitions, expressions and relations.

This typed representation inference system certainly overcomes the failings of the abstract interpretation:

- Representations are generated efficiently. Each expression is only analysed once and fixpointing is not required within maximally strong components.
- Representations are selected for value-relations as well as relations.
- The equality constraints needed for representation coercions are generated.

CombK :: SF[(C × D) ↔ C]

CombI :: SF[F ↔ F]

combS :: SF[(AL[($G^=$ × $W^=$) ↔ V] × AL[$G^=$ ↔ $W^=$] × $G^=$) ↔ V]
combS :: SF[(AL[($G^=$ × $T^=$) ↔ V] × SF[$G^=$ ↔ $T^=$] × $G^=$) ↔ V]
combS :: SF[(SF[(G × W) ↔ S] × AL[G ↔ W] × G) ↔ S]
combS :: SF[(SF[(G × T) ↔ S] × SF[G ↔ T] × G) ↔ S]
combS :: CF[(CF[(G × W) ↔ P] × AL[G ↔ W] × G) ↔ P]
combS :: CF[(CF[(G × T) ↔ P] × SF[G ↔ T] × G) ↔ P]
combS :: CF[(AL[($P^=$ × Q) ↔ $M^=$] × CF[$P^=$ ↔ Q] × $P^=$) ↔ $M^=$]

Figure 4.13: Typed representations inferred for `S`, `K` and `I` combinators

Why should type inference be better than abstract interpretation for representation selection? The answer appears to lie with the use of substitutions and unification. For example, consider analysis of a parameterised definition. The abstract interpretation must synthesise abstract representation values for the parameters and analyse the definition once for each possible parameter representation. By contrast, in the type system the parameter representations are denoted by type variables and the substitution generated by the inference process maps those variables to the required typed representations. The substitution is created by unifying the type variables with the operator typed representation inference rules — the required representations are extracted directly from those built into the operator rules.

The problem could probably be solved by a richer abstract interpretation than the one described. For example, better abstract elements that include information about the representation of relation domain and range elements, could have been chosen. For example, higher-order abstract interpretation could be applied over a more complex domain. However it is unlikely that any solution based on abstract interpretation would be as simple or efficient as the type system since there is a natural link between type and representation.

4.8 Operator Overload Resolution

Typed representation analysis of an expression yields a set of syntactically identical expressions each of which possesses its own unique typed representation. Within each of these expressions any subexpression that is a compound relation designation is of the form:

```
op arg
```

Let *opTR* denote the typed representation of `op arg`, and *argTR* denote the typed representation of `arg`. The expression typed representation *opTR* is produced by

applying to *argTR* the inference rule for op that unifies with:

$$SF[argTR \leftrightarrow opTR]$$

Each of the inference rules for an operator reflects one, unique argument-result representation constraint and, therefore, corresponds to one particular definition of that operator. The above inference rule can therefore be used to identify the required definition of op.

The typed representation inference rules for an operator are also the typed representations that operator may have when used as an elementary relation designation. The particular typed representation assigned determines the required definition of that operator.

Thus typed representation analysis not only selects representations for relations but also resolves operator overloading. When expressions are interpreted or compiled the appropriate definition for each operator occurrence is selected using typed representation information.

4.9 Reporting Constraints to the Programmer

4.9.1 The Frontier of Expression

Any relational programming system must impose some constraint on expression construction because not all operators are definable for all argument representations. This constraint forms the frontier for what can be expressed in that system. In RPL the constraint is the fixed representation scheme — the representation bottleneck. However, this scheme outlaws many expressions that could be computed if each operator was allowed several definitions.

In Drusilla the constraint is imposed by the typed representation system. Each operator's constraints are determined by its typed representation inference rules which reflect the argument-result representation constraints in its implementation. For example, in RPL, the domain operator, dom, is *only* defined for an extensionally represented relation, for which the result is an extensionally represented set. By contrast, in Drusilla, the domain operator is defined for any extensionally represented relation, the result being another extensional relation, and for any relation represented intensionally by a set-valued function, the result being another set-valued function.

For example, the domain a relation represented by set-valued function `f`, can be represented by the set-valued function `domFunc`:

```
domFunc f x  = [],      f x = []
             = [Unit],  otherwise
```

Informally this states that a value is in the domain of a relation if it maps to a non-empty set of results under the representing function `f`.

The domain operator cannot be defined for any relation represented by a characteristic function because this representation can only be used to test whether given domain and range values are related, and not to test whether a given value

is a member of the relation's domain. The operator could be used if the range of the characteristic function is recursively enumerable, i.e. if the range is of type *un* or *string*. However, this is not used since, in practice, most range types are polymorphic.

The constraints for each operator are similar to this — unlike RPL they are not artificially imposed by a fixed representation scheme, but are naturally enforced by the limit of what is computable. This allows many more expressions to be used and pushes back the frontier of expression creating an implementation of relational programming which is closer to MacLennan's [65] original conception.

The computational constraints for operators curtail freedom of expression in Drusilla — it is possible to construct legal expressions that are not computable and hence useless. The Drusilla system is able to identify such expressions because they have no typed representation. The *modes* of relation use discussed in chapter 3 are used to inform the programmer of constraints on how relations may be used without reference to relation representations. They help the programmer to construct only representable expressions.

4.9.2 Homogeneity of Program and Data

The need for modes can be explained from another point of view.

One aim of the SETL system is to make the data representations used to realise an algorithm dependent on program code and not vice-versa. The programmer specifies only abstract data structures and need have no concern for their implementation.

The Drusilla system takes this idea to its logical conclusion by making program and code homogeneous — both are denoted by relations and are amenable to manipulation by the same set of relational operators. Unfortunately a price must be paid for this homogeneity. The representation resulting from application of an operator depends on the representation of its argument. Furthermore, some operators are only defined for extensional relations, for example relation range (`rng`). At the implementation level code and data can be distinguished by representation — relational data structures are extensionally represented and relations that are program code, intensionally represented. However, at the programming level this distinction blurs because a relational data structure may perform computation and intensional relations may be passed as argument to higher-order relations.

When constructing an expression the programmer may apply an operator to a relation, the representation of which it is not defined for, for example, applying the range operator to code, then it may not be possible to generate a typed representation for that expression. Alternatively, the typed representation generated may constrain use of that expression in a way unexpected by the programmer. The Drusilla system should, therefore, provide some mechanism to help the programmer construct expressions that have representations.

Two possible approaches to this problem are:

Automatic symbolic manipulation: this can be applied to any expression that cannot be represented in an attempt to produce a mathematically equivalent

expression that is representable. If this mechanism is capable of successfully transforming any unrepresentable expression then the programmer can program in the abstract mathematics of relational algebra and is free from all representation related concerns. Unfortunately perfection is unlikely because the representation constraints for operators and expressions reflect the limit of computability. It is a difficult task to make non-computable expressions computable. Chapter 5 presents a mechanism which, although not perfect, is capable of improving the representations of some expressions.

Automatic mode generation: section 4.9.3 describes how the possible modes of use for a definition can be automatically generated from its set of typed representations. This information aids the programmer in expression construction by explaining how relations may be used.

4.9.3 From Typed Representations to Moded Types

Chapter 3 introduced two type systems that assist the programmer:

Calculus types can be inferred statically to give the programmer information about the mathematical structure of a program.

Moded types for a relation can be derived by combining its calculus type with information about its possible modes of use (show,forward and test). Moded types not only convey mathematical structure but also operational structure.

There is a strong link between typed representations and moded types — moded types explain to the programmer how a relation may be used operationally given the constraints placed by its available typed representations. Section 4.2 described these constraints. Any expression that uses a relation in a mode not supported by the typed representations available to that relation is said to be *badly moded*. If an expression cannot be given a representation it must be badly moded. Conversely, if an expression is badly moded it cannot possibly have a representation. Therefore automatic generation of moded types for definitions helps the programmer to formulate well moded and hence representable expressions.

The modes that a relational typed representation *rep[domTR* $\leftrightarrow$ *rngTR]* may support are determined as follows:

- If *rep* is an association list and equality is not defined for either *domTR* or *rngTR* then **show** mode is supported.
- If *rep* is an association list and equality is defined for *domTR* but not *rngTR* then **show** and **forward** modes are supported.
- If *rep* is an association list and equality is defined for both *domTR* and *rngTR* then **show, forward** and **test** modes are supported.
- If *rep* is a set-valued function and equality is not defined for *rngTR* then **forward** mode is supported.

- If *rep* is a set-valued function and equality is defined for *rngTR* then **forward** and **test** modes are supported.
- If *rep* is a characteristic function then **test** mode is supported.

These rules are summarised in Table 4.9.3 which presents example typed representations together with the modes of use they support. Each mode is supported by a particular typed representation. For each typed representation new modes are permitted by introducing the equality constraints needed at the implementation level to support the appropriate representation coercions.

Typed representation	Moded types
AL[A $\leftrightarrow$ B]	sh[A $\leftrightarrow$ B]
AL[$A^=$ $\leftrightarrow$ B]	sh[$A^=$ $\leftrightarrow$ B], fo[$A^=$ $\leftrightarrow$ B]
AL[$A^=$ $\leftrightarrow$ $B^=$]	sh[$A^=$ $\leftrightarrow$ $B^=$], fo[$A^=$ $\leftrightarrow$ $B^=$], te[$A^=$ $\leftrightarrow$ $B^=$]
SF[A $\leftrightarrow$ B]	fo[A $\leftrightarrow$ B]
SF[A $\leftrightarrow$ $B^=$]	fo[A $\leftrightarrow$ $B^=$], te[A $\leftrightarrow$ $B^=$]
CF[A $\leftrightarrow$ B]	te[A $\leftrightarrow$ B]

In the Drusilla system each definition is given a set of typed representations. Moded types for a relation definition are generated from its typed representations by using knowledge of possible representation coercions. For example the definition of `fst`

```
(x,y) fst (x).
```

is given the typed representation:

$$\text{SF}\,[(\text{A} \times \text{B}) \leftrightarrow \text{B}]$$

This typed representation means that the relation can be used only in forward mode. However, by enforcing equality on the range elements, a restricted typed representation that supports test mode of use can be created:

$$SF[(A \times B^=) \leftrightarrow B^=]$$

Therefore, the moded types for `fst` are:

$$\text{fst}\ ??\ \text{fo}[(\text{A} \times \text{B}) \leftrightarrow \text{B}]$$
$$fst\ ??\ te[(A \times B^=) \leftrightarrow B^=]$$

where '??' means *'has the moded type'*.

Therefore, typed representations can be mapped into moded types by adding new coercion information to the operator typed representation inference rules. Moded types are defined in Miranda in Figure 4.14 and the algorithm for generating moded types from typed representations is presented in Figure 4.15.

From a different perspective moded types may be perceived as a relational form of the mode analysis used for Prolog programs. Mode analysis, as described by Reddy [86] and Debray [30], is used to see in what directions Prolog procedures may be run, i.e. which terms within a rule must be ground and which may be left uninstantiated.

```
modedType ::=   NumMT                                  number type
              | StringMT                               string type
              | UnMT                                   unit type
              | TuMT [modedType]                       tuple type
              | MTvar trVar                            moded type variable
              | RMT mode modedType modedType           relation moded type

mode ::=   Sh                                          show mode
         | Fo                                          forward mode
         | Te                                          test mode

repToMode :: relRep -> mode                            representation to mode
repToMode AL = Sh
repToMode SF = Fo
repToMode CF = Te
```

Figure 4.14: Moded types defined as an algebraic datatype

4.10 Summary and Conclusions

The introduction identified the *representation bottleneck* as the crucial weakness in the implementation of RPL and has discussed way to break down this barrier to freedom of expression in relational programming.

Representations for relations that are suited to the implementation of relational programming in a functional language were discussed. The characteristic function has been identified as a representation, which although not used in the RPL system, is of use. The advantages and disadvantages of each representation and possible coercions between representations were discussed.

Related work on representation selection and overload resolution was considered. The representation selection work is for a less general class of problem considering only representations for data structures and not for computing components. One other difference is that the problem discussed here is the initial, rather than efficient, execution of relational programs. The work on overload resolution proved to be of more relevance inspiring the mechanism based on abstract interpretation initially used to select representations for relations. The failings of this approach inspired a more powerful and efficient one based on type inference.

The typed representation system could be used more generally as a new mechanism for inferring both ad-hoc and parametric polymorphism. The advantage it has over a Haskell-like class system is that operators may be defined for arguments of differing type. For example, in Haskell classes, operations such as addition and multiplication can easily be defined for two floating point numbers or for two integers but cannot be so easily defined for one integer and one float. Furthermore, Haskell's class system can be used in the implementation of the ad-hoc polymorphism of the relational operators but only at the expense of losing

```
complexMode BotTR = []
complexMode (RTR rep domTR rngTR) =
   mkShow rep domTR rngTR ++ mkForw rep domTR rngTR ++
   mkTest rep domTR rngTR
complexMode other = [simpleMode other]

simpleMode Num = NumMT
simpleMode String = StringMT
simpleMode Un = UnMT
simpleMode (TRvar var) = MTvar var
simpleMode (TuTR tup) = TuMT (map simpleMode tup)
simpleMode (RTR rep domTR rngTR) =
   RMT (repToMode rep) (simpleMode domTR) (simpleMode rngTR)

mkShow AL domTR rngTR =
  [RMT Sh (simpleMode domTR) (simpleMode rngTR)]
mkShow rep domTR rngTR = []

mkForw AL domTR rngTR
   = [RMT Fo (simpleMode (applySubToTR sub domTR))
             (simpleMode (applySubToTR sub rngTR))], domEqSucc
   = [],                                             otherwise
     where
     (domEqSucc,sub) = mkEquality domTR
mkForw SF domTR rngTR =
   [RMT Fo (simpleMode domTR) (simpleMode rngTR)]
mkForw CF domTR rngTR = []

mkTest AL domTR rngTR
   = [RMT Te (simpleMode (applySubToTR bigSub domTR))
             (simpleMode (applySubToTR bigSub rngTR))],
                                  domEqSucc & rngEqSucc
   = [],                          otherwise
     where
     (domEqSucc,subA) = mkEquality domTR
     (rngEqSucc,subB) = mkEquality (applySubToTR subA rngTR)
     bigSub = composeSubs subB subA
mkTest SF domTR rngTR
   = [RMT Te (simpleMode (applySubToTR sub domTR))
             (simpleMode (applySubToTR sub rngTR))],  rngEqSucc
   = [],                                              otherwise
     where
     (rngEqSucc,sub) = mkEquality rngTR
mkTest CF domTR rngTR =
   [RMT Te (simpleMode domTR) (simpleMode rngTR)]
```

Figure 4.15: Algorithm to generate moded types from typed representations

parametric polymorphism of operators.

The information generated by this type system indicates the definition required for each relational operator instance thereby resolving operator overloading. The computational constraints imposed by typed representation analysis reflect the computational constraints of operator definitions. The representation bottleneck has, therefore, been widened to give relational programming more freedom of expression.

Representation constraints are reported to the programmer by means of moded types — a generalisation of calculus types that not only reflect the mathematical nature of a program but also its operational nature. Moded types for program definitions are automatically generated from their typed representations. In this way the constraints are reported to the programmer as a relational abstraction of definition representations.

In chapter 7 this system will be evaluated in order to assess whether this implementation effort gives the expected increase in programmer freedom.

Chapter 5

Symbolic Manipulation of Drusilla Programs

5.1 Introduction

As MacLennan [67] has noted a relational language is amenable to symbolic manipulation. Laws of equivalence between relational expressions are well documented in mathematics and laws specific to the operators used in Drusilla can easily be formulated. Such laws can be used as directed rewrite rules for the automatic transformation of relational expressions. The aim of manipulation is to transform a given expression into a form that is more desirable, with respect to some criteria, while retaining its meaning. This preserves the mathematical meaning of a relational program but changes its operational behaviour.

Chapter 4 identified two mechanisms that help ensure expressions have representations. One informed the user of the possible modes of use for relations; the other used symbolic (algebraic) manipulation to transform any unrepresentable expression to a form that can be represented. If an expression cannot be given a representation then it is unusable. Similarly, if a definition cannot be given a representation then it is unusable as is any definition that references it.

Symbolic manipulation can also be used to improve the execution characteristics of a relational program. The use of relations for programming permits flexible handling of non-determinism and search based computation as chapter 7 will demonstrate. However, a given state space may be so large that searching it becomes infeasible. This chapter describes how automatic manipulation can decrease the size of a program's search space to make it run faster and use less memory.

Section 5.2 introduces issues to be considered for the symbolic manipulation of Drusilla programs. The relevance of related work is assessed in section 5.3. Section 5.4 describes two possible manipulation strategies for improving representations in programs. Another manipulation strategy is introduced in section 5.5 and shown to be superior to the previous two. Section 5.6 considers how program search spaces might be manipulated. Section 5.7 discusses how representation and search space manipulation may be applied to a whole Drusilla program. Conclusions on the use of symbolic manipulation are drawn in section 5.8.

5.2 Preliminary Discussion

This section discusses issues that are fundamental to the symbolic manipulation of Drusilla programs.

5.2.1 Separating Manipulation Concerns

Symbolic manipulation can be applied to a program to

1. improve representation possibilities;
2. decrease the size of its search space.

The first of these two uses of manipulation is by far the most important. If a program is to be executable then all definitions must possess a defined typed representation. Representation oriented manipulation is essential for the execution of programs where one or more definitions cannot be represented. By contrast, search space oriented manipulation can only increase the execution speed of a program that is already executable. It is never essential for execution; it is purely an optimisation technique designed to improve the run-time performance of programs.

The aims of these two types of manipulation may sometimes conflict. For example there may be two possible transformations for a given expression: one improves the representation but increases search space size and the other decreases search space size but leaves the expression without a representation. To resolve such conflict the manipulation concerns are separated and representation considerations given priority over search space considerations. The manipulation scheme is split into two phases:

1. Manipulation attempts to transform the program until all definitions have at least one possible representation. This program state is referred to as *full representation*. During this manipulation no concern is given to the size of the program's search space. If full representation cannot be achieved then the program is unusable and rejected by the system: the programmer must reformulate the program definitions by considering the system generated mode information.

2. Search space oriented manipulation is applied to the program and care is taken to ensure that full representation is maintained. Transformations can only be applied if they decrease search space size while preserving representations.

The two types of manipulation used by these phases can be distinguished:

representation manipulation is algebraic manipulation designed to improve the representation of a given expression or definition.

search space manipulation is algebraic manipulation designed to decrease the size of a program's search space.

The emphasis of this chapter is on representation manipulation — possible approaches are explored and the strategy used in the Drusilla system is described in detail.

5.2.2 The Need for Calculus Type Correctness

The calculus type of an expression reflects its meaning in the abstract mathematical world of relational algebra. A program is therefore regarded as having valid mathematical meaning if and only if it is calculus type correct, i.e. if a calculus type can be generated for each defining expression in that program. It is essential that manipulation techniques preserve meaning. Therefore manipulation should never be able to change the calculus type of any expression or definition. A corollary of this is that manipulation should never be capable of correcting a program that is calculus type incorrect. This observation simplifies manipulation since it implies that any program considered for manipulation must be calculus type correct and hence have some recognizable structure.

In the Drusilla implementation calculus type inference is applied to the given program. If the program is type correct then typed representation analysis is applied and manipulation considered, otherwise it is rejected.

5.2.3 Types, Representations and Manipulation

Typed representation correctness is a stronger condition than calculus type correctness — the set of all expressions that have a defined typed representation is a proper subset of the expressions that have a calculus type. In an ideal world these two sets of expressions would be the same. This, however, would imply that the programmer could program in abstract mathematics without any regard for operational structure. The aim of representation manipulation is to generate typed representations for expressions that have a calculus type but no representation. This can be formally defined as a function:

```
representationManipulation :  (CT \ TR) → TR
```

where `CT` is the set of all expressions with a valid calculus type and `TR` is the set of all expressions with a defined typed representation.

It may also be necessary to apply typed representation inference to a transformed expression to generate its new representation. Therefore, typed representation inference and representation manipulation are inextricably linked and must interact.

Manipulation approaches can be grouped into two broad categories according to the way they interface with typed representation analysis. These are *eager* and *lazy* approaches described in section 5.2.5.

5.2.4 The Use of Laws for Manipulation

Symbolic manipulation in the Drusilla system is based on laws supplied by the implementor. This subsection explains:

- how laws can be used as term rewriting rules;
- how laws can be preprocessed to ensure rewriting improves representation and to obviate re-analysis;
- how the programmer can prove the correctness of laws before supplying them to the system.

Laws as Rewrite Rules

Term rewriting forms the basis of all the symbolic manipulation strategies considered. The rewrite rules are extracted from the given laws which are equations of the form:

```
expL = expR
```

Here `expL` and `expR` are two relational expressions and the programmer is asserting their equivalence. It is assumed that the programmer has proved their equivalence before making the assertion. An example law is:

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

A law typically consists of a number of free variables glued together by the relational operators. In the above example, `r`, `s` and `t` are the variables. Laws can also involve the empty relation (`{}`) and primitive operators in elementary relation designation form.

For a given law, `expL = expR`, let `L` and `R` be the sets of free variables occurring in `expL` and `expR` respectively.

If $L \subseteq R$ then it is sensible to consider the rewrite rule:

$$expR \rightarrow expL$$

If $R \subseteq L$ then it is sensible to consider the rewrite rule:

$$expL \rightarrow expR$$

These restrictions on the use of law equations as rewrite rules are in accordance with the description given by Huet and Oppen [45] of how a term rewriting system can be extracted from a set of equations.

Application of Rewrite Rules

The expression `A` on the left hand side of a rewrite rule, $A \rightarrow B$, pattern matches with an expression `C` if and only if there exists a substitution σ for the free variables in `A` such that:

$$\sigma A = C$$

If such a substitution can be created then expression `C` can be rewritten as σB.

Adding Representation Information to Laws

One representation is considered to be better than another if it supports more modes of use. For representation manipulation it would be desirable to know under which circumstances, if any, each rewrite rule improves a given expressions representation. For example, suppose a given expression pattern matches with the left hand sides of N different rewrite rules. Consider the case where N is large but only one rule will improve the representation. It would be computationally expensive to apply each rule and re-analyse each expression to determine if the representation has improved. If the 'good' rule could be identified in advance then this computational cost would be saved.

A relation definition may be constructed from any given 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. Typed representation analysis of such a definition yields a set of definitions each of which has its own typed representation. This process also produces typed representations for the law expressions `expL` and `expR` and all their subexpressions. The rewrite rules extracted from such analysed laws also contain this typed representation information.

For example, Table 5.1 shows the typed representations generated for the law:

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

Examination of this table reveals that the rewrite rule

```
inv s ; inv r → inv (r ; s)
```

improves representation for the following typed representations of `r` and `s`:

s :: SF[F ↔ G]	r :: AL[$H^{=}$ ↔ F]
s :: SF[W ↔ X]	r :: SF[V ↔ W]
s :: CF[T ↔ U]	r :: SF[S ↔ T]

However the reverse rule

```
inv (r ; s) → inv s ; inv r
```

never improves representation.

Table 5.2 shows the typed representations generated for the law:

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

Examination of this table reveals that the rewrite rule

```
(r >> s) ; t → r ; (s << t)
```

only improves representation for one combination of typed representations:

r :: CF[K ↔ $U^{=}$] s :: AL[$U^{=}$ ↔ un] t :: SF[$U^{=}$ ↔ $M^{=}$]

s	r	inv s ; inv r	inv (r ; s)
AL[$I^{=} \leftrightarrow$ J]	AL[H $\leftrightarrow$ $I^{=}$]	AL[J $\leftrightarrow$ H]	AL[J $\leftrightarrow$ H]
SF[F $\leftrightarrow$ G]	AL[$H^{=} \leftrightarrow$ F]	CF[G $\leftrightarrow$ $H^{=}$]	AL[G $\leftrightarrow$ $H^{=}$]
CF[C $\leftrightarrow$ D]	AL[$B^{=} \leftrightarrow$ C]	CF[D $\leftrightarrow$ $B^{=}$]	CF[D $\leftrightarrow$ $B^{=}$]
AL[$Z^{=} \leftrightarrow$ $P^{=}$]	SF[Y $\leftrightarrow$ $Z^{=}$]	CF[$P^{=} \leftrightarrow$ Y]	CF[$P^{=} \leftrightarrow$ Y]
SF[W $\leftrightarrow$ X]	SF[V $\leftrightarrow$ W]	$\perp_{TR}$	CF[X $\leftrightarrow$ V]
CF[T $\leftrightarrow$ U]	SF[S $\leftrightarrow$ T]	$\perp_{TR}$	CF[U $\leftrightarrow$ S]
AL[Q $\leftrightarrow$ $R^{=}$]	CF[P $\leftrightarrow$ Q]	CF[$R^{=} \leftrightarrow$ P]	CF[$R^{=} \leftrightarrow$ P]
SF[N $\leftrightarrow$ O]	CF[M $\leftrightarrow$ N]	$\perp_{TR}$	$\perp_{TR}$
CF[K $\leftrightarrow$ L]	CF[J $\leftrightarrow$ K]	$\perp_{TR}$	$\perp_{TR}$

Table 5.1: Typed representations for law `inv s ; inv r = inv (r ; s)`

Again the reverse rule

```
r ; (s << t) → (r >> s) ; t
```

never improves representation.

The procedure for extracting rewrite rules from laws is modified to ensure that only those rules that improve representation are used. Let ML and MR denote the set of modes supported by the representations of expL and expR respectively.

If L $\subseteq$ R and MR $\subset$ ML then allow the rule

$$expR \rightarrow expL$$

If R $\subseteq$ L and ML $\subset$ MR then allow the rule

$$expL \rightarrow expR$$

Rules extracted in this way are called *representation improving*. The reverse application of these rules must make representation worse and the reverse rules are therefore referred to as *representation spoiling*. A set of *representation maintaining* rules for which ML = MR can also be extracted.

Representation Improving Application of Rewrite Rules

The rewrite rule application procedure must be modified to ensure that the *representation improving* rules improve expression representation. This new procedure is only applicable to expressions that have been subjected to typed representation inference. For a representation improving rule $A \rightarrow B$ to improve the representation of an expression C two conditions must be satisfied:

1. Expression C must *syntactically* pattern match with expression A. There must exist a substitution σ for the free variables in A such that:

$$\sigma A = C$$

This is the condition for normal rule application.

r	s	t	(r >> s) ; t	r ; (s << t)
AL[C $\leftrightarrow V^{=}$]	AL[$V^{=} \leftrightarrow$ un]	AL[$V^{=} \leftrightarrow$ W]	AL[C $\leftrightarrow$ W]	AL[C $\leftrightarrow$ W]
SF[T $\leftrightarrow V^{=}$]	AL[$V^{=} \leftrightarrow$ un]	AL[$V^{=} \leftrightarrow$ W]	SF[T $\leftrightarrow$ W]	SF[T $\leftrightarrow$ W]
CF[K $\leftrightarrow V^{=}$]	AL[$V^{=} \leftrightarrow$ un]	AL[$V^{=} \leftrightarrow M^{=}$]	CF[K $\leftrightarrow M^{=}$]	CF[K $\leftrightarrow M^{=}$]
AL[C $\leftrightarrow D^{=}$]	AL[$D^{=} \leftrightarrow$ un]	SF[$D^{=} \leftrightarrow$ U]	AL[C $\leftrightarrow$ U]	AL[C $\leftrightarrow$ U]
SF[T $\leftrightarrow U^{=}$]	AL[$U^{=} \leftrightarrow$ un]	SF[$U^{=} \leftrightarrow$ U]	SF[T $\leftrightarrow$ U]	SF[T $\leftrightarrow$ U]
CF[K $\leftrightarrow U^{=}$]	AL[$U^{=} \leftrightarrow$ un]	SF[$U^{=} \leftrightarrow M^{=}$]	$\perp_{TR}$	CF[K $\leftrightarrow M^{=}$]
AL[$W^{=} \leftrightarrow Z^{=}$]	AL[$Z^{=} \leftrightarrow$ un]	CF[$Z^{=} \leftrightarrow$ S]	CF[$W^{=} \leftrightarrow$ S]	CF[$W^{=} \leftrightarrow$ S]
SF[N $\leftrightarrow M^{=}$]	AL[$M^{=} \leftrightarrow$ un]	CF[$M^{=} \leftrightarrow$ S]	CF[N $\leftrightarrow$ S]	CF[N $\leftrightarrow$ S]
CF[E $\leftrightarrow Z^{=}$]	AL[$Z^{=} \leftrightarrow$ un]	CF[$Z^{=} \leftrightarrow$ S]	$\perp_{TR}$	$\perp_{TR}$
AL[C $\leftrightarrow D^{=}$]	SF[$D^{=} \leftrightarrow$ un]	AL[$D^{=} \leftrightarrow$ Q]	AL[C $\leftrightarrow$ Q]	AL[C $\leftrightarrow$ Q]
SF[T $\leftrightarrow U^{=}$]	SF[$U^{=} \leftrightarrow$ un]	AL[$U^{=} \leftrightarrow$ Q]	SF[T $\leftrightarrow$ Q]	SF[T $\leftrightarrow$ Q]
CF[K $\leftrightarrow C^{=}$]	SF[$C^{=} \leftrightarrow$ un]	AL[$C^{=} \leftrightarrow M^{=}$]	CF[K $\leftrightarrow M^{=}$]	CF[K $\leftrightarrow M^{=}$]
AL[Z $\leftrightarrow H^{=}$]	SF[$H^{=} \leftrightarrow$ un]	SF[$H^{=} \leftrightarrow$ O]	AL[Z $\leftrightarrow$ O]	AL[Z $\leftrightarrow$ O]
SF[Q $\leftrightarrow U^{=}$]	SF[$U^{=} \leftrightarrow$ un]	SF[$U^{=} \leftrightarrow$ O]	SF[Q $\leftrightarrow$ O]	SF[Q $\leftrightarrow$ O]
CF[H $\leftrightarrow H^{=}$]	SF[$H^{=} \leftrightarrow$ un]	SF[$H^{=} \leftrightarrow$ O2]	$\perp_{TR}$	$\perp_{TR}$
AL[$W^{=} \leftrightarrow M^{=}$]	SF[$M^{=} \leftrightarrow$ un]	CF[$M^{=} \leftrightarrow$ M]	CF[$W^{=} \leftrightarrow$ M]	CF[$W^{=} \leftrightarrow$ M]
SF[N $\leftrightarrow Z^{=}$]	SF[$Z^{=} \leftrightarrow$ un]	CF[$Z^{=} \leftrightarrow$ M]	CF[N $\leftrightarrow$ M]	CF[N $\leftrightarrow$ M]
CF[E $\leftrightarrow M^{=}$]	SF[$M^{=} \leftrightarrow$ un]	CF[$M^{=} \leftrightarrow$ M]	$\perp_{TR}$	$\perp_{TR}$
AL[C $\leftrightarrow D^{=}$]	CF[$D^{=} \leftrightarrow$ un]	AL[$D^{=} \leftrightarrow$ K]	AL[C $\leftrightarrow$ K]	AL[C $\leftrightarrow$ K]
SF[T $\leftrightarrow U^{=}$]	CF[$U^{=} \leftrightarrow$ un]	AL[$U^{=} \leftrightarrow$ K]	SF[T $\leftrightarrow$ K]	SF[T $\leftrightarrow$ K]
CF[K $\leftrightarrow$ eP]	CF[eP $\leftrightarrow$ un]	AL[eP $\leftrightarrow M^{=}$]	CF[K $\leftrightarrow M^{=}$]	CF[K $\leftrightarrow M^{=}$]
AL[Z $\leftrightarrow$ H]	CF[H $\leftrightarrow$ un]	SF[H $\leftrightarrow$ I]	AL[Z $\leftrightarrow$ I]	AL[Z $\leftrightarrow$ I]
SF[Q $\leftrightarrow H^{=}$]	CF[$H^{=} \leftrightarrow$ un]	SF[$H^{=} \leftrightarrow$ I]	SF[Q $\leftrightarrow$ I]	SF[Q $\leftrightarrow$ I]
CF[H $\leftrightarrow U^{=}$]	CF[$U^{=} \leftrightarrow$ un]	SF[$U^{=} \leftrightarrow$ I]	$\perp_{TR}$	$\perp_{TR}$
AL[$W^{=} \leftrightarrow$ F]	CF[F $\leftrightarrow$ un]	CF[F $\leftrightarrow$ G]	CF[$W^{=} \leftrightarrow$ G]	CF[$W^{=} \leftrightarrow$ G]
SF[N $\leftrightarrow M^{=}$]	CF[$M^{=} \leftrightarrow$ un]	CF[$M^{=} \leftrightarrow$ G]	CF[N $\leftrightarrow$ G]	CF[N $\leftrightarrow$ G]
CF[E $\leftrightarrow Z^{=}$]	CF[$Z^{=} \leftrightarrow$ un]	CF[$Z^{=} \leftrightarrow$ G2]	$\perp_{TR}$	$\perp_{TR}$

Table 5.2: Typed representations for law `(r >> s) ; t = r ; (s << t)`

2. The typed representation of expression σA and all its subexpressions must pattern match with the corresponding subexpressions in C.There must exist a substitution δ for the typed representation variables in σA such that:

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

If these two conditions are satisfied then expression C can be rewritten as an expression $\delta(\sigma$B). Application of the substitution for free variables, σ, to expression B generates the new expression. Application of the substitution for typed representation variables, δ, to σB and all its subexpressions generates its typed representation obviating the need to re-apply typed representation inference. (However, if C is a subexpression, it will be necessary to re-analyse the enclosing expression). Although this lowers cost of computation the procedure will still be expensive if δ is created by matching the typed representation of every subexpression in A with the corresponding typed representation in C. The cost can be lowered further by exploiting the *leaf sufficiency* theorem.

Lemma 1 (fixed leaf representation) *Typed representation analysis of a compound relation designation, op arg, produces a set*

$$\{(s_1, op\ arg_1), \ldots, (s_n, op\ arg_n)\}$$

in which the representation of each arg_i, arg_i^{TR} $(1 \leq i \leq n)$, is fixed, i.e. cannot be completely polymorphic (a type variable).

Proof

Each of the typed representation inference rules for op applies to one fixed representation. Each arg_i^{TR}, $1 \leq i \leq n$, is unified with one of these rules.

□

Corollary 1 (inference rule uniqueness) *The typed representation of each expression op arg_i, $1 \leq i \leq n$, is produced by applying one of the typed representation inference rules for op which can be identified.*

Proof

Let exp_i^{TR} denote the typed representation of $op\ arg_i$. Each exp_i^{TR} is produced by applying to arg_i^{TR} the inference rule for op that unifies with:

$$SF[arg_i^{TR} \leftrightarrow exp_i^{TR}]$$

Uniqueness follows since each rule applies to one representation.

□

Theorem 1 (leaf sufficiency) *If condition 1 of the rule application procedure is satisfied then, to derive substitution δ, it is sufficient to pattern match the typed representations of the variables in* A *with those of the subexpressions in* C *being substituted for them.*

Proof

The substitution σ is a set of maplets:

$$\{V_1 \rightarrow exp_1, \ldots, V_n \rightarrow exp_n\}$$

where V_i, $1 \leq i \leq n$, are the free variables in A and exp_i are subexpressions of C. Let V_i^{TR} be the typed representation of V_i and let exp_i^{TR} be the typed representation of exp_i. If there exists a substitution, ξ, for typed representation variables such that

$$\xi V_i^{TR} = exp_i^{TR}, 1 \leq i \leq n$$

then by induction over expression construction:

$$\xi\,(\sigma\,A)^{TR} = C^{TR}$$

i.e. $\delta = \xi$. This follows from *inference rule uniqueness* because σA and C are syntactically identical. At each expression construct (operator) in both expressions, the same typed representation inference rule is applied.

□

The algorithm for expression pattern matching is given in Figure 5.1 and uses the typed representation pattern matching algorithm shown in Figure 5.2. The algorithm for expression rewriting is given in Figure 5.3. These algorithms process the expression and typed representation algebraic datatypes defined in Figure 4.6 and Figure 4.7 respectively.

```
varSub == ([char],relExp)

patternMatch :: relExp -> relExp -> (bool,trSub,[varSub])
patternMatch (E formOp formTR formArg) (E actOp actTR actArg)
  = (True,sub,varSubs),           actOp = formOp & succ
  = (False,idSub,[]),             otherwise
    where
    (succ,sub,varSubs) = patternMatch formArg actArg
patternMatch (Fpar name tr) actExp =
   (succ,sub,[(name,actExp)])
   where
   (succ,sub) = typRepMatch tr (getExpTR actExp)
patternMatch (Tu formTup) (Tu actTup) =
   patternMatchTups formTup actTup
patternMatch formOther actualOther =
   (formOther = actualOther,idSub,[])

patternMatchTups :: [relExp] -> [relExp] ->
                                (bool,trSub,[varSub])
patternMatchTups formTup actTup =
   foldr2 patternMatchTupEls (True,idSub,[]) formTup actTup

patternMatchTupEls :: relExp -> relExp -> (bool,trSub,[varSub])
                                       -> (bool,trSub,[varSub])
patternMatchTupEls formExp actExp (succ,oldSub,subsA) =
   (succ & newSucc,composeSubs newSub oldSub,subsB ++ subsA)
   where
   (newSucc,newSub,subsB) = patternMatch newFormExp actExp
   newFormExp = applySubToExp oldSub
                   (applyVarSubs subsA formExp)

applyVarSubs :: [varSub] -> relExp -> relExp
applyVarSubs subs exp = foldr makeSub exp subs

makeSub :: varSub -> relExp -> relExp
makeSub sub (E op tr arg) = E op tr (makeSub sub arg)
makeSub sub (Tu tup) = Tu (map (makeSub sub) tup)
makeSub (subParName,subExp) (Fpar parName tr)
   = subExp,                 parName = subParName
   = Fpar parName tr,        otherwise
makeSub sub other = other
```

Figure 5.1: Algorithm for pattern matching expressions

```
typRepMatch :: typedRep -> typedRep -> (bool,trSub)
typRepMatch BotTR other = (False,idSub)
typRepMatch other BotTR = (False,idSub)
typRepMatch (TRvar (Alpha v)) (TRvar (Alpha w))
   = (True,idSub),                                  v = w
   = (True,newSub (Alpha v) (TRvar (Alpha w))), otherwise
typRepMatch (TRvar (Beta v)) (TRvar (Beta w))
   = (True,idSub),                                  v = w
   = (True,newSub (Beta v) (TRvar (Beta w))),   otherwise
typRepMatch (TRvar (Alpha v)) x =
   (definedTR x & ~(occurs (Alpha v) x),newSub (Alpha v) x)
typRepMatch (TRvar (Beta v)) x =
   (showable x & ~(occurs (Beta v) x),newSub (Beta v) x)
typRepMatch (TuTR tupX) (TuTR tupY) = tupTRmatch tupX tupY
typRepMatch (RTR repX domX rngX) (RTR repY domY rngY) =
   (repX = repY & succ1 & succ2, composeSubs subY subX)
   where
   (succ1, subX) = typRepMatch domX domY
   (succ2, subY) = typRepMatch (applySubToTR subX rngX) rngY
typRepMatch otherX otherY = (otherX = otherY, idSub)

tupTRmatch :: [typedRep] -> [typedRep] -> (bool,trSub)
tupTRmatch tupX tupY =
   foldr2 tupElTRmatch (True, idSub) tupX tupY

tupElTRmatch :: typedRep -> typedRep -> (bool,trSub)
                                     -> (bool,trSub)
tupElTRmatch x y (succ1, subX) =
   (succ1 & succ2, composeSubs subY subX)
   where
   (succ2,subY) = typRepMatch (applySubToTR subX x) y
```

Figure 5.2: Algorithm for typed representation pattern matching

```
rule == (relExp,relExp)

algManip :: [rule] -> (typedRep -> bool) ->
                          relExp -> (bool,relExp)
algManip allRules goalFun exp
   = (False,exp),          possExps = []
   = (True, hd possExps),  otherwise
     where
     possExps = mapcat (applyRule exp) goodRules
     goodRules = filter (goalFun . rewrittenTR) allRules

rewrittenTR :: rule -> typedRep
rewrittenTR (lhs,rhs) = getExpTR rhs

applyRule :: relExp -> rule -> [relExp]
applyRule actExp (lhs,rhs) =
   createNewExp (patternMatch lhs actExp) rhs

createNewExp :: (bool,trSub,[varSub]) -> relExp -> [relExp]
createNewExp (True,sub,varSubs) newExp =
   [applySubToExp sub (applyVarSubs varSubs newExp)]
createNewExp (False,sub,varSubs) newExp = []
```

Figure 5.3: Algorithm for representation improving rewriting

The Correctness of Laws

The correctness of program transformations is dependent on the correctness of the rewrite rules used for those transformations, which, in turn, depend on the correctness of the laws from which they are extracted. Therefore each law, `expL = expR`, asserted by the programmer must be a true mathematical equivalence. It is essential for the programmer to prove the validity of all laws before submitting them since it is outside the scope of the Drusilla system to prove the correctness of laws.

Any Drusilla expression can be mapped into a calculus expression by replacing each operator used to form a compound relation designation by its calculus definition. Two Drusilla expressions are equivalent if, and only if, they can be mapped to the same expression in the Drusilla calculus. An example of a valid law is:

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

Proof

$$
\begin{aligned}
x\ (\mathrm{inv}\ (r\ ;s))\ y &\Leftrightarrow y\ (r\ ;s)\ x \\
&\Leftrightarrow \exists z\ .\ y\ r\ z \wedge z\ s\ x \\
&\Leftrightarrow \exists z\ .\ z\ (\mathrm{inv}\ r)\ y \wedge x\ (\mathrm{inv}\ s)\ z \\
&\Leftrightarrow \exists z\ .\ x\ (\mathrm{inv}\ s)\ z \wedge z\ (\mathrm{inv}\ r)\ y \\
&\Leftrightarrow x\ (\mathrm{inv}\ s\ ;\ \mathrm{inv}\ r)\ y
\end{aligned}
$$

□

As stated in section 5.2.2 all transformations and therefore all rewrite rules should be calculus type preserving. If this is to be the case then both sides of any given law should have the same calculus type. For example, consider the law:

```
(s << r) /\ (s <- r) = {}
```

This is a true mathematical equivalence. However, the expression on the left hand side has the same calculus type as relation `r` but the expression `{}` is a completely polymorphic relation. The domain and range types of the two expressions may be different and this law therefore cannot be used by the system.

5.2.5 Classifying Manipulation Strategies

This subsection describes classifications for representation manipulation strategies.

Eager and Lazy Manipulation

Manipulation strategies may be differentiated according to when they are invoked and the amount of rewriting they attempt.

Eager manipulation is based on the generate and test problem solving paradigm [120]. Manipulation is applied to each group of mutually recursive definitions before typed representation inference. Each defining expression is rewritten into as many different forms as possible. Typed representation inference is applied to each expression form generated. The one with the best representation is used, the others are discarded.

Eager approaches are perhaps the most intuitive. They are simple and guarantee to find the best form for each expression. Unfortunately they have two disadvantages:

- The cost of manipulation is high — many expressions are created only to be discarded.
- The cost of typed representation inference is high — each expression form generated must be analysed and typed representation inference is a complex and computationally expensive technique. The representation improving rule application procedure cannot be used because typed representations are not known before manipulation.

Lazy Manipulation is at the other extreme to eager manipulation. Typed representation inference is applied to the program in the usual manner. Each

expression and subexpression that cannot be represented is assigned the undefined representation $\perp_{TR}$. If a defining expression is assigned $\perp_{TR}$ then manipulation is invoked and its aim is to generate a representation for the whole expression by applying as few transformations as possible. Lazy approaches to manipulation have significant advantages over eager approaches:

- The cost of manipulation is lower — only expressions that have no representation are manipulated. The transformation is less costly because typed representation information can be used to direct manipulation. i.e. any subexpressions with $\perp_{TR}$ must be rewritten to a form whose representation permits the whole expression to be represented.
- The cost of typed representation inference is lower. Much of the cost of re-inference can be avoided by using the representation improving rule application procedure. Furthermore, the directed nature of lazy manipulation generates fewer expressions to be re-analysed.

Irregular, Bottom-up, and Top-down

Strategies may be classified according to the order in which they rewrite subexpressions. This corresponds to Kowalski's [59] concept of an algorithm being composed of separate logic and control elements. The logic of a manipulation algorithm is the rewrite rules it uses and its control the order in which it applies rules to subexpressions.

Irregular strategies select the subexpressions to rewrite in an irregular order.

Bottom-up strategies begin at the leaves of an expression tree and move upward toward the root applying rewrites at appropriate points.

Top-down strategies begin at the root and proceed towards the leaves applying transformations at appropriate points.

Locally and Globally Improving

If an expression is to be given a representation then each of its subexpressions must be given a representation. Moreover the representation of an expression is dependent on the representations of its subexpressions.

Locally improving strategies are concerned purely with giving each subexpression a representation. How that representation affects the whole expression is given no consideration.

Globally improving strategies aim to give each subexpression a defined typed representation that is beneficial to the whole expression. A strategy is called *angelic* if its subexpression transformations always improve the representation of the whole expression. If there is more than one possible transformation for the subexpression then the one most beneficial to the representation

of the whole expression is applied. This does not necessarily imply that each rewrite applied improves representation — it may be necessary for a subexpression's representation to be made worse temporarily in order for it to be improved later.

5.2.6 Problems Facing Manipulation

Two problems that any successful manipulation strategy must face are the need to search and the need to terminate.

The Need to Search

Symbolic manipulation is a search based computation. There may be many rewrite rules applicable and hence many different transformations possible for any given expression that has no representation. More than fifty laws have been identified for use in the Drusilla system. This means that a considerable search may be needed to find a representable expression, even with a lazy approach to manipulation. If a manipulation approach is to be effective then this search must be controlled:

Law analysis may be used to identify the representation improving laws. This obviously narrows the search; for example, of the 54 laws used in the Drusilla system, only the seven shown in Figure 5.4, improve representation if used alone.

Representation goals can be supplied to help rewriting to globally improve expression representation.

Rewriting Cycles and Non-termination

One pitfall is that of applying rewrite rules in a cyclical fashion. For example, an expression A may be transformed successively into expressions $B_1, B_2, \ldots, B_n$ and B_n then transformed back to A. If this is allowed to continue then the manipulation will be non-terminating.

The problem of non-termination is more general. A manipulation strategy will terminate if and only if there is no possibility for an infinite sequence of transformations.

If the strategy adopted in Drusilla is to be feasible then it must be accompanied by some proof of termination.

5.2.7 Properties of a Good Manipulation Strategy

In reality, the distinction between eager and lazy strategies blurs — thees categories are two ends of the spectrum of manipulation strategies. One strategy is regarded as being more lazy than another if it involves less rewriting and less typed representation analysis. The advantages of lazy manipulation over eager manipulation are, in practice, so substantial that a good strategy must be lazy.

```
inv (inv r) = r
rng (inv r) = dom r
inv s ; inv r = inv (r ; s)
r \ (r \ s) = r /\ s
inv r >> s = inv (s << r)
(r >> s) ; t = r ; (s << t)
(r ; s) ; t = r ; (s ; t)
```

Figure 5.4: Laws that improve representations

Globally improving rewrites are preferable to locally improving, but their application requires knowledge of the context of the whole expression. When a subexpression is to be rewritten its required typed representation must be propagated down from the root node. This also helps control the manipulation search by making it goal directed. Therefore, a good manipulation strategy must, to some extent, be top-down. However, when a subexpression is manipulated the enclosing expression must be re-analysed and may itself require manipulation — any strategy must therefore also be, at least partially, bottom-up.

The aim of this chapter is to identify the best strategy from those considered. This should be lazy, angelic, and be a combination of top-down and bottom-up to make rewriting goal directed.

5.3 Related Work

This section assesses the relevance of related symbolic manipulation work: optimisation of queries in relational databases, term rewriting systems and an artificial intelligence technique called meta-level inference.

5.3.1 Optimisation of Queries in Relational Databases

Efficient methods of processing unanticipated queries are a crucial prerequisite for the success of generalised database management systems. A wide variety of approaches to improve the performance of query evaluation algorithms have been proposed. The approach of interest here uses logic-based and semantic transformations.

A *query* is a language expression that describes data to be retrieved from a database. Query optimisation tries to minimise the response time for a given query language and mix of query types.

The main costs that optimisation attempts to minimise are those of secondary storage access and CPU usage. Several ideas underly manipulation techniques used to reduce these costs:

- avoid duplication of effort;

- avoid unnecessary operations by looking ahead;
- sequence operations in an optimal fashion.

Jarke and Koche [50] present a comprehensive survey of query optimisation techniques. They identify three main goals for query transformation: *standardisation, simplification* and *amelioration.*

Standardisation

This is the construction of a standardised starting point for query optimisation. This typically relies on a normalisation procedure to produce a normalised form for a given query. For example, queries expressed in relational calculus may be standardised by reduction to disjunctive prenex normal form.

Simplification

This is the elimination of redundancy. An expression that uses redundant operations can be transformed into an equivalent one without them. For example, the expression `A OR A` may be reduced to `A`.

Amelioration

This refers to the construction of expressions that are improved with respect to evaluation performance. Query simplification does not necessarily produce a unique expression. The evaluation of expressions that are equivalent may differ substantially with respect to performance parameters.

Many transformation heuristics, when applied to expressions, yield ameliorated expressions with respect to evaluation performance. Two example heuristics are:

- combination of a sequence of projections into a single projection;
- combination of a sequence of restrictions into a single restriction.

The goal of several ameliorating transformations is to minimise the size of intermediate results to be constructed, stored and retrieved. An important heuristic moves selective operations, such as restriction and projection, over constructive operations, such as join and Cartesian product, to perform the selective operations as early as possible. Smith and Chang [102] give examples of such heuristics.

Relevance of this Work

The three principles of standardisation, simplification and amelioration should apply to manipulation of Drusilla expressions. However, it is difficult to see how standardisation could be used. Expressions could be translated into the relational calculus by replacing operators with their calculus definitions but it

is difficult to see any advantage for manipulation of calculus expressions over Drusilla expressions.

Search space manipulation may be thought of as simplification because it aims to eliminate redundant operators in expressions. It may also be thought of as amelioration since it tries to improve evaluation performance. Representation manipulation may be thought of as an extreme form of amelioration. Without it an expression may give no performance at all!

The heuristics used for amelioration are, however, of little use for Drusilla expressions. They are designed to minimise the quantity of data processed i.e. the size of relations. Relations in Drusilla are not bulk repositories of data, they are computing components. The aim for manipulation of Drusilla expressions is computation oriented — representation manipulation makes expressions executable; search space manipulation aims to lower the cost of computation by reducing the number of operators used. However, pushing selection operators in front of construction operators may not improve representation possibilities and may even increase the computation cost. For example, it is considered desirable, for database queries, to push restriction operations in front of union operations. This corresponds to use of the rewrite rule:

$$\texttt{t << (r} \bigvee \texttt{s)} \rightarrow \texttt{(t << r)} \bigvee \texttt{(t << s)}$$

The right hand side of this rule uses one more operator than the left hand side. This rule is not representation improving and, generally, the more operators used in an expression, the poorer its representation is likely to be because each operator introduces representation constraints. Furthermore, if both expressions are represented by set-valued functions then the rewrite will be detrimental to efficiency — relation t is applied once on the left hand side and twice on the right hand side.

Database query optimisation also depends on selecting efficient storage methods that use structures such as indexes, B-trees or hash tables. Since these structures are not present in Drusilla storage optimisation techniques cannot be used.

The principles of transformation in databases are therefore of interest but manipulation of Drusilla expressions has different objectives.

5.3.2 Term Rewriting Systems

One paradigm of computing with equations uses them as rewrite rules over terms. Huet and Oppen [45] define a *term rewriting system* to be a set of directed equations:

$$\mathcal{P}_0 = \{\lambda_i \rightarrow \rho_i \mid i \in I\} \textit{ such that, for all } \lambda \rightarrow \rho \textit{ in } \mathcal{P}_0,\ \vartheta(\rho) \subseteq \vartheta(\lambda)$$

$\vartheta(M)$ denotes the set of variables occurring in term M.

The *reduction relation* $\rightarrow_{\mathcal{P}_0}$ associated with $\mathcal{P}_0$ is closed under substitution and replacement. That is:

$$\begin{aligned} \mathrm{M} \rightarrow_{\mathcal{P}_0} \mathrm{N} &\Rightarrow \sigma(\mathrm{M}) \rightarrow_{\mathcal{P}_0} \sigma(\mathrm{N}) \\ \mathrm{M} \rightarrow_{\mathcal{P}_0} \mathrm{N} &\Rightarrow \mathrm{P[u \leftarrow M]} \rightarrow_{\mathcal{P}_0} \mathrm{P[u \leftarrow N]} \end{aligned}$$

For discussion of term rewriting systems, the following notation will be used:

- $\rightarrow$ denotes the reduction relation $\rightarrow_{\mathcal{P}_0}$.
- $\rightarrow^+$ denotes the transitive closure of $\rightarrow$.
- $\rightarrow^*$ denotes the transitive-reflexive closure of $\rightarrow$.
- $=_{\mathcal{P}_0}$ denotes $\mathcal{P}_0$ equality when $\mathcal{P}_0$ is considered a set of equations.

The fundamental difference between equations and term rewriting rules is that equations denote equality whereas term rewriting systems treat equations directionally as one-way replacements. The only substitutions required for term rewriting are the ones found by pattern matching. Rewrite rules can be used in this way to make deductions equationally. However, a set of rules is only complete as a proof system if it is Church-Rosser (i.e. confluent). $\mathcal{P}_0$ is Church-Rosser if and only if

$$\forall M, N . M =_{\mathcal{P}_0} N \Leftrightarrow \exists P . M \rightarrow^* P \wedge N \rightarrow^* P$$

When $\mathcal{P}_0$ is Church-Rosser the normal form of a term is unique when it exists. A sufficient condition for the existence of such a canonical form is the termination of all rewritings. $\mathcal{P}_0$ is *noetherian* or *finitely terminating* if and only if, for no M, is there an infinite sequence of reductions issuing from M. When $\mathcal{P}_0$ is a finite set of equations that is confluent and noetherian the equational theory $=_{\mathcal{P}_0}$ is decidable, since now $M =_{\mathcal{P}_0} N$ if and only if $M\downarrow = N\downarrow$. The property of confluence is undecidable for arbitrary of term rewriting systems, but decidable for noetherian systems.

Relevance of this Work

The laws used in the Drusilla system are equations. Term rewriting can therefore be used as the basis for manipulation of Drusilla expressions. Generation of a representation for a Drusilla expression can be thought of as a term rewriting problem. A given expression, *A*, must be rewritten to an expression, *B*, that has a representation and can be found by equational reasoning.

The system will only be complete for such equational reasoning if it is noetherian and Church-Rosser. The analysed rewrite rules must be noetherian because they improve representation and the representation can only be improved a finite number of times.

However, there is no guarantee that the rules would be Church-Rosser. For example, suppose the system contains just two rules that improve representation:

```
inv s ; inv r → inv (r ; s)
inv (inv r) → r
```

If these rules are to be applied to an expression `inv (inv p) ; inv q` then there are two possible rewrites:

```
→ p ; inv q
→ inv (q ; inv p)
```

This system is not confluent without the law:

```
p ; inv q  =  inv (q ; inv p)
```

The lack of confluence introduces the element of search into manipulation. An attempt could be made to make a given system of rewrite rules confluent by applying the Knuth-Bendix completion procedure [56]. This algorithm attempts to make a system confluent through the addition of new rules but it may fail or enter an infinite loop. Moreover, there is no guarantee that the new rules it produces are representation improving or that the normal forms generated by the confluent system are those with the best representation.

Therefore, although equational laws can be used as directed term rewriting rules for the manipulation of Drusilla expressions equational reasoning cannot be exploited.

5.3.3 Algebraic Manipulation by Meta-level Inference

Meta-level inference [14, 15] is a technique for controlling inference that was developed for algebraic manipulation. It was tested in PRESS (PRolog Equation Solving System) [106] — a Prolog program for solving equations and performing algebraic manipulation on transcendental expressions. These are expressions involving polynomial, trigonometric, exponential and logarithmic functions. A commentary on this work is given in [34].

In PRESS inference is constructed at two levels simultaneously: the *object*-level and the *meta*-level. The object level encodes knowledge about the facts of the domain — in this case the rules of the algebra. The meta-level encodes control or strategic knowledge — in this case methods for applying algebraic manipulation.

Multiple Sets of Rewrite Rules

The object level of PRESS consists of rewrite rules organised into several sets. Each set performs a particular algebraic manipulation and is associated with a syntactic characterisation of the kind of rule it contains. The meta-level reasons about the task to be performed and the rules available to achieve it and on this basis selectively applies the rewrite rules to the current algebraic expression.

The use of multiple sets has several advantages over conventional term rewriting systems that exhaustively apply rules from a single set:

1. With multiple sets of rules, a particular axiom may be used in different directions in different sets. With selective application the axiom may even be used in different directions in the same set without any danger of looping.

2. The syntactic characterisation of rules allows the system to automatically decide, for a given rule, in which directions it should be used and in which sets.

3. A proof of method termination can be based on rule application being selective. i.e. selective application may terminate where exhaustive application might not.

   In PRESS the argument for termination is the same for each method. If no rule from the set applies then the method terminates trivially. If a rule does apply then some numerical property of the expression is reduced in magnitude. If this property is initially finite, then only a finite number of rule applications are possible before the method fails to apply and exits.

Meta and Object-levels

The object level in PRESS consists of algebraic axioms describing relationships between real numbers. Given an equation to be solved these define an object level search space. The methods used in the meta-level express relationships between the syntactic representations of algebraic axioms. The PRESS meta-level describes a meta-theory of algebra.

Inference at the meta-level in PRESS causes algebraic manipulation to be carried out at the object level. Many of the meta-level predicates express the relation:

`Answer` is the result of applying `Rule` to `Expression`

If this relation is set up as a goal to be satisfied with `Rule` and `Expression` bound to a particular rule and expression then PRESS answers the question by applying the rule to the expression. This constitutes a step in the object level search.

As meta-level inference continues it experiments with different object level steps. If successful it finds a proof or solution at the object level. By using this technique each object level decision can be based on an arbitrary amount of inference at the meta-level, i.e as much as desired. Making a wrong decision and hence taking direction on a fruitless search can become a rare event.

This technique still involves search but the search is at the meta-level, not the object-level. The object-level search space is determined by the ways a given expression can be rewritten. This is typically large because it has a high branching rate. By comparison, the meta-level search space is small because is has a low branching factor. Most choices lie between equally successful branches and bad choices rapidly lead to dead ends. This is because choices are usually between different methods of solution and each method uses a terminating rewrite rule set.

Relevance of this Work

This technique naturally applies to symbolic manipulation of relational expressions. Laws may be separated into classes using the analysis discussed in section 5.2.4. Methods for manipulation can be based on the approaches a human might take to transformation. These techniques form the basis for transformation in Drusilla, described in section 5.5.

5.4 Two Possible Representation Manipulation Strategies

This section describes and evaluates two manipulation strategies designed for use in the Drusilla system.

5.4.1 A Graph Theoretic Strategy

This strategy, based on graph theory, is *eager* and has *irregular* control of rule application.

The nodes of the graph are the given expression and all expressions mathematically equivalent to it. The arcs of the graph are directed and are formed by the available rewrite rules. An arc exists from an expression node A to an expression node B if there exists a rewrite rule which, when applied to A or one of its subexpressions yields B. The representation improving rule application procedure cannot be used since the strategy is eager.

The problem of transforming the expression into the required form can now be viewed in terms of searching the graph. Standard search algorithms such as depth first or breadth first can be used. The start node for the search is the given expression. The goal node, to be searched for, is any expression in the graph that can be given the required representation.

The search through the graph is paralleled by expression transformations. Whenever an arc is traversed, the corresponding rewrite rule is applied to the relevant part of the out node expression. Typed representation inference is applied to each expression node visited and the first one that can be given the required typed representation is taken as the goal. Rewriting cycles are avoided by these algorithms because they keep a record of all expression nodes visited. Furthermore, the manipulation will terminate when these algorithms terminate.

This strategy is complete — it will find the best representation for any expression given the rules available for manipulation. However, it is so inefficient that it is not practical — although guaranteed to terminate, it may take a long time.

Efficiency is a problem because the rules cannot be guaranteed to improve representation — a problem that applies to any eager strategy. Consequently there is little scope for intelligence and a complex search is required. Efficient search algorithms, such as best-first, branch-and-bound, or $A\star$, that try to find the shortest path from start node to goal node could be used, although these require knowledge of remaining distance to the goal node and this may be difficult to estimate. However, the limitations of eager manipulation would still make this strategy inefficient.

5.4.2 Integrating Manipulation With Representation Inference

This *lazy* strategy directly incorporates manipulation into the typed representation inference algorithm using locally representation improving rewrites. Typed representation inference proceeds as normal — each defining expression is analysed from the leaves to the root. Whenever a subexpression is given $\perp_{TR}$, an attempt is made to rewrite that subexpression to a form that can be represented.

Analysis of the expression then continues, even if the expression cannot be rewritten. If one subexpression has $\perp_{TR}$, then so must any expression it is part of and the rewriting attempts will therefore continue through the rest of the analysis. The strategy terminates when typed representation analysis terminates.

One advantage of this strategy over the graph theoretic is that it identifies the $\perp_{TR}$ expressions which must be rewritten. However, since the rewriting is locally improving the effect of each rewrite on the representation of the whole expression is not considered.

5.5 Representation Manipulation by Meta-level Inference

5.5.1 Overview of the Strategy

The manipulation strategy used in the Drusilla system is based on meta-level inference [14, 15, 106] as discussed in section 5.3.3. This approach uses several strategies, called methods, for rewriting expressions. Each method intelligently combines rewrite rules to transform some class of expression in a structured fashion.

Rule Classification

Bundy et al. used syntax to subdivide the rewrite rules into sets — each set containing rules particularly suited to certain transformations. In a similar fashion the Drusilla system uses law analysis to subdivide rewrite rules into two categories — those that improve representations and those that preserve representations. The first group is essential for representation manipulation and the second for search space manipulation.

Intelligence in Meta-level Inference

Meta-level inference is not one, but several goal directed strategies. Each method improves representation globally — given an expression, X, and a goal typed representation G, it must rewrite X to a form whose typed representation pattern matches with G. Manipulative power comes not just from each method individually, but also from the way methods interact manipulating different parts of a given expression. To transform an expression, `op arg`, a method may invoke other methods to manipulate the expression `arg` and its subexpressions. Representation goals for subexpression `arg` can be extracted directly from the typed representation inference rules for `op`.

For conventional manipulation strategies the size of the search space is dictated by the number of different ways a given expression can be rewritten. Such search spaces suffer a combinatorial increase in size for a linear increase in the number of rewrite rules. By contrast the search space for meta-level inference is the different ways in which the rules can be usefully combined.

Meta-level inference is more flexible than the approaches previously considered. Its transformation power can easily be increased by adding new methods. This power is maximised by ensuring that each method is generally applicable to a wide variety of expressions. It is also applicable to search space manipulation. The same transformation methods can be used to reduce search space size while preserving representations.

5.5.2 Methods for Representation Manipulation

This subsection describes the rewriting methods used for representation manipulation in the Drusilla system.

There are no rewrite rules applicable to basic values hence the method applicable to these involves no transformations and terminates trivially. The methods for manipulating tuples and compound relation designations use only representation improving rules and are described below. Methods are combined by the higher-order functions presented in Figure 5.5.

Rewriting a Compound Relation Designation

Every compound relation designation is parsed into abstract syntax of the form:

```
op arg
```

This expression will have no representation if there is no typed representation inference rule for `op` applicable to the representation of `arg`. There are two methods available for rewriting such an expression. In the order they are tried:

Compound Method A: rewrite the whole expression. An attempt is made to apply a single rule to the whole expression, the aim being to create a new expression with the goal representation. If such a rule cannot be found then the method fails. This method terminates because there is only a finite number of rules available for rewriting and at most rewrite is made.

Compound Method B: rewrite the subexpression then re-analyse. `arg` is rewritten to a form whose representation matches one of the typed representation inference rules for `op`. Each of these rules in turn can be used as a goal. For example, one rule for relation composition is:

$$\frac{(r \times s) :: (AL[A \leftrightarrow B] \times SF[B \leftrightarrow C])}{r\,;\,s :: AL[A \leftrightarrow C]}$$

An attempt is made to rewrite the tuple of relations `arg` to make its representation match with the top of the inference rule. If this rewrite succeeds then the inference rule is applied to the new typed representation for `arg`. This must yield a new typed representation for the whole expression.

```
method == (trSub,relExp,typedRep) -> (bool, trSub, relExp)

lMeth :: method -> method -> method
lMeth methodA methodB val
   = (True,subA,expA),      succA
   = methodB val,           otherwise
     where
     (succA,subA,expA) = methodA val

sOptMeth :: method -> method -> method
sOptMeth methodA methodB val
   = (True,subB,expB),      succA & succB
   = (succA,subA,expA),     otherwise
     where
     (succB,subB,expB) = methodB (subA,expA,TRvar (Alpha 0))
     (succA,subA,expA) = methodA val

tclMeth :: method -> method
tclMeth methA val
   = rtclMeth methA (subA,expA,TRvar (Alpha 0)),   succA
   = (False,subA,expA),                            otherwise
     where
    (succA,subA,expA) = methA val

rtclMeth :: method -> method
rtclMeth methA val
   = rtclMeth methA (subA,expA,TRvar (Alpha 0)),   succA
   = (True,subA,expA),                             otherwise
     where
     (succA,subA,expA) = methA val
```

Figure 5.5: Higher-order functions for combining manipulation methods

Compound Method C: rewrite the subexpression then the whole expression. This method applies if `arg` can be rewritten to `newArg` that has a defined typed representation but no inference rule for `op` applies to `newArg`.

An attempt is made to transform the whole expression `op newArg` to a form that does have a representation by applying a single rewrite rule.

The methods for manipulating a compound designation are presented in Figure 5.6.

Rewriting a Tuple

The form of the typed representation goal determines which method is invoked:

Tuple Method A: the goal is a tuple of representations. An attempt is made to rewrite each expression in the tuple. The goal for each expression transformation is the corresponding representation in the goal tuple of representations.

Tuple Method B: the goal is any defined typed representation. An attempt is made to transform each expression in the tuple. The goal for each expression transformation is simply to have a defined typed representation. A tuple of expressions has a representation if and only if every expression within that tuple has a representation.

The methods for manipulating expressions that are not compound designations are given in Figure 5.7.

5.5.3 Proof of Termination for Methods

Each method either improves the representation or reduces the size of the given expression to be manipulated. Termination is guaranteed because there is a bound on representation improvement and each expression is of finite size.

5.5.4 Comparison With Other Manipulation Strategies

The meta-level inference methods used here have significant advantages over the graph theoretic and integrated strategies.

This manipulation strategy, like the graph theoretic, is complete with respect to representation improving rules. However, it is more efficient than the graph based strategy because representation improving rules are exploited and the search is directed by typed representation information.

The integrated strategy is subsumed by meta-level inference — the subexpressions that have no representation are isolated in a similar manner but the rewrites applied improve the representation globally.

```
rewriteCompound (sub,E op tr arg,resGoal)
   = hd goodForms,                                  goodForms ~= []
   = (False,sub,E op tr arg),                       badForms = []
   = tclRep (argSub,E op tr newArg,resGoal),        otherwise
     where
     (goodTR,badTR) = goodBadTRrules resGoal op
     goodForms = mapcat (argRewrite sub op arg) goodTR
     badForms = mapcat (badTRarg sub arg) badTR
     (argSub,newArg) = hd badForms
rewriteCompound (sub,exp,goal) = (False,sub,exp)

goodBadTRrules :: typedRep -> relOp -> ([typedRep],[typedRep])
goodBadTRrules resGoal =
    split (rngTRmatch resGoal) . opTypeRep

split p xs = (filter p xs, filter ((~) . p) xs)

rngTRmatch :: typedRep -> typedRep -> bool
rngTRmatch resGoal (RTR SF dom rng) = trMatch resGoal rng

argRewrite :: trSub -> relOp -> relExp -> typedRep ->
                        [(bool,trSub,relExp)]
argRewrite subA op arg (RTR SF dom rng)
   = [(True,newSub,newExp)],          success
   = [],                              otherwise
     where
     (dummySucc,subB,newArg) = repOriented (subA,arg,dom)
     (success,subC) = typRepUnification dom (getExpTR newArg)
     newSub = composeSubs subC subB
     newExp = applySubToExp newSub (E op rng newArg)

badTRarg :: trSub -> relExp -> typedRep -> [(trSub,relExp)]
badTRarg sub arg (RTR SF dom rng)
   = [(newSub,newArg)],     succ
   = [],                    otherwise
     where
     (succ,newSub,newArg) = repOriented (sub,arg,dom)
```

Figure 5.6: Method for manipulation of a compound designation

```
twoPhaseMan = repOriented $sOptMeth searchOriented

repOriented = repManip $sOptMeth tclRep

repManip = noRewrite $lMeth repRewrite $lMeth rewriteCompound
              $lMeth tupRewrite

tclRep = tclMeth repRewrite

repRewrite = rewriteExp repImprove

noRewrite (sub,E op tr arg,goal) =
   (trMatch goal tr,sub,E op tr arg)
noRewrite (sub,Tu tup,goal) =
   (trMatch goal (getExpTR (Tu tup)),sub,Tu tup)
noRewrite (sub,simpleExp,goal) = (True,sub,simpleExp)

rewriteExp :: ((typedRep -> bool) -> relExp ->
                           (bool,relExp)) -> method
rewriteExp manFun (sub,E op tr arg,goal) =
   (succ,sub,newExp)
   where
   (succ,newExp) = manFun (trMatch goal) (E op tr arg)
rewriteExp manFun (sub,exp,goal) = (False,sub,exp)

tupRewrite (sub,Tu tup,TuTR tupReps)
  = (True,newSub,Tu newTup),       succ
  = (False,sub,Tu tup),            otherwise
    where
    (succ,newSub,newTup) =
             foldr2 tupElemManip (True,sub,[]) tup tupReps
tupRewrite (sub,Tu tup,TRvar v) =
   tupRewrite (sub,Tu tup,TuTR varTup)
   where
   varTup = map (const (TRvar v)) tup
tupRewrite (sub,other,goal) = (False,sub,other)

tupElemManip tupEl goal (True,sub,tup) =
   (succ,newSub,newTupEl : tup)
   where
   (succ,newSub,newTupEl) = repOriented (sub,tupElB,goal)
   tupElB = applySubToExp sub tupEl
tupElemManip tupEl goal (False,sub,tup) = (False,sub,tup)
```

Figure 5.7: Methods for representation manipulation

5.5.5 More General Methods

The power and flexibility of meta-level inference is directly dependent on the methods available — as new methods are added this power is increased. This subsection considers the implications of adding two new methods that would make the strategy applicable to a more general class of expressions.

Method 1: if representation manipulation of an expression is to be successful it may be necessary for its representation to be made worse in order for it later to be improved later. This method supports such manipulation by permitting *representation spoiling* rules to be applied before representation improving rules.

Method 2: this a restricted form of method 1. Representation spoiling rules are not permitted but *representation maintaining* rules may be applied before representation improving rules.

Problems with these Methods

Although these methods generalise the manipulation strategy and increase manipulative power they present two significant problems:

Non-termination: The proof of method termination given in section 5.5.3 does not hold for methods 1 and 2. Moreover, if they are introduced in their full generality then they may well be non-terminating. To guarantee termination, restrictions must be imposed upon them but this will reduce the manipulative power. For example, method 1 could be restricted to apply at most one representation spoiling rule to any given subexpression; method 2 could be restricted to apply a representation improving rule after each representation maintaining rule.

Fruitless search: The new methods permit many more rules to be used. Method 1 allows each law to be used in either direction. Method 2 allows each law to be used in at least one direction. Consider the 54 laws used in the Drusilla system — typed representation analysis isolates just seven that improve certain expressions with particular typed representations. Method 1 would allow 101 (i.e. $54 * 2 - 7$) and method 2 would allow 47 (i.e. $54 - 7$) rules, none of which improves representation. If an expression's representation is to be improved then application of a non-improving rule must eventually be followed by a representation improving rule. Therefore, one of the non-improving rules must generate an expression that matches in both syntax and typed representation with one of the improving rules, otherwise the search will be fruitless. It is precisely such fruitless avenues that meta-level inference aims to avoid. As more rules and more general methods are introduced the meta-level search space becomes closer in size to the object-level search space. This may mean a large increase in transformation time for comparatively little gain in manipulative power.

5.5.6 Examples of Representation Manipulation

Examples of the use of representation manipulation are given here. The transformations are produced by the Drusilla system using meta-level inference methods described in section 5.5.2.

Example One

The example program is:

```
r = neg [+ 1].
s = {(1,1),(2,4)}.
t = [* 4].
rst = (r ; s) ; t.
```

The typed representations of the definitions in this program are:

$$\begin{aligned} &\texttt{r} :: \text{CF}[\text{num} \leftrightarrow \text{num}] \\ &\texttt{s} :: \text{AL}[\text{num} \leftrightarrow \text{num}] \\ &\texttt{t} :: \text{SF}[\text{num} \leftrightarrow \text{num}] \\ &\texttt{rst} :: \perp_{TR} \end{aligned}$$

The definition of `rst` cannot be given a representation because of the nature of composition. The composition `r ; s` is given a characteristic function representation. The composition of this relation with `t` has no representation because a characteristic function can only be composed with an association list.

A new expression with a representation is generated by a single rewrite. The definition of `rst` is rewritten as:

```
rst = r ; (s ; t)
```

This new expression is generated using the law for associativity of composition.

```
(a ; b) ; c = a ; (b ; c)
```

The composition `s ; t` is given an association list representation. The composition of characteristic function `r` with this relation has a valid representation and the new typed representation is:

$$\texttt{rst} :: \text{CF}[\text{num} \leftrightarrow \text{num}]$$

Example Two

The example program is:

```
succ = [1 +].
pred = [- 1].
multId = [* 1].
numId = inv ((inv succ ; inv pred) ; inv multId).
```

Here `numId` is a contrived form of the identity relation over numbers. The typed representations given to the definitions are:

succ :: SF[num ↔ num]
pred :: SF[num ↔ num]
multId :: SF[num ↔ num]
numId :: $\perp_{TR}$

The two laws used for this transformation are:

```
inv (inv r) = r
inv q ; inv p = inv (p ; q)
```

The sequence of rewrites applied is:

```
    inv ((inv succ ; inv pred) ; inv multId).
→   inv (inv (pred ; succ) ; inv multId)
→   inv (inv (multId ; (pred ; succ)))
→   multId ; (pred ; succ)
```

This final expression has the typed representation:

`multId ; (pred ; succ)` :: SF[num ↔ num]

Note that the final rewrite is not actually necessary — the expression has a defined typed representation after the second rewrite — it is represented by a characteristic function. The system, however, always ends manipulation by attempting to improve the representation by applying a single rewrite to the whole expression. This is inexpensive, but may be useful.

5.5.7 Completeness for Representation Manipulation

This subsection proves the completeness of meta-level inference for representation manipulation with respect to the representation improving rules.

The proof is simplified by considering only representation improving rules. There is no need to consider the effect on representation of applying a rule since it is known the representation must improve. It is, however, necessary to prove that the representation manipulation algorithm $\mathcal{M}$ finds, if one exists, a sequence of rewrites that improves an expressions representation. To state this formally, notation is introduced:

$$\mathcal{R} \vdash (exp \rightarrow newExp)$$

Here $\mathcal{R}$ denotes the set of representation improving rules, *exp* is an expression without representation and *newExp* is an equivalent expression with representation. The turnstile states that `exp` can be transformed to `newExp` by applying rules from $\mathcal{R}$. Application of rule, $r \in \mathcal{R}$, to the whole expression `exp` (i.e. *not* to any of its subexpressions) is denoted by $r\ exp$. Completeness can now be stated formally:

$$\mathcal{R} \vdash (exp \rightarrow newExp) \Rightarrow (newExp \in \mathcal{M}\, exp)$$

The transformed expression newExp is created by applying some sequence of rewrite rules in $\mathcal{R}$, as a function, to exp. The meta-level inference algorithm is denoted by function $\mathcal{M}$. The proof is for the methods discussed in section 5.5.2 and presented in Figure 5.6 and Figure 5.7. In the following proof the typed representation of any expression, exp, is denoted by exp^{TR}.

Proof of completeness by induction over construction of expression

Base Case: There are no rewrite rules applicable to basic values at the leaves of an expression tree. This corresponds to the `noRewrite` method.

Hypothesis for Induction: assume the strategy is complete for expressions exp_1 and exp_2:

$$\mathcal{R} \vdash (exp_1 \rightarrow newExp_1) \Rightarrow (newExp_1 \in \mathcal{M}\, exp_1)$$
$$\mathcal{R} \vdash (exp_2 \rightarrow newExp_2) \Rightarrow (newExp_2 \in \mathcal{M}\, exp_2)$$

The strategy is now proved complete for any expression construct applied to exp_1 or exp_2.

Case tuple $Tu[exp_1, exp_2]$

The rules only apply directly to compound designations and not to tuples. A tuple can be manipulated by applying manipulation to each of its elements. **Tuple Method A** and **Tuple Method B** perform this task. The new tuple generated is $Tu[newExp_1, newExp_2]$

Case compound relation designation $op\ \ exp_1$

The rules only apply directly to compound designations and not to tuples. Suppose there exists a sequence of rewrites for $op\ \ exp_1$ which produces the required typed representation:

$$\mathcal{R} \vdash ((op\ \ exp_1) \rightarrow compExp) \Rightarrow (compExp \in \mathcal{M}\,(op\ \ exp_1))$$

Expression $compExp$ has the desired typed representation $compExp^{TR}$. By the hypothesis, algorithm $\mathcal{M}$ can generate $newExp_1$.

Subcase 1

The new expression $compExp$ can be generated applying a rewrite rule $r \in \mathcal{R}$:

$$r\ (op\ exp_1) = compExp$$
$$(r\ (op\ exp_1))^{TR} = compExp^{TR}$$

Compound Method A applies this rule.

Subcase 2

There exists a typed representation inference rule for op, SF[dom $\leftrightarrow$ rng], and a unifying substitution, σ, such that:

$$\sigma\, SF[dom \leftrightarrow rng] = \sigma\, SF[newExp_1^{TR} \leftrightarrow compExp^{TR}]$$
$$op\ \ newExp_1 = compExp$$
$$\sigma\ (op\ \ newExp_1)^{TR} = compExp^{TR}$$

Compound Method B produces the expression *compExp* using algorithm $\mathcal{U}$, shown in Figure 4.11, to compute σ.

Subcase 3

If subcase 1 and subcase 2 do not apply, there must be a rewrite rule, $r \in \mathcal{R}$, such that:

$$\begin{aligned} r\ (op\ newExp_1) &= compExp \\ (r\ (op\ newExp_1))^{TR} &= compExp^{TR} \end{aligned}$$

Compound Method C applies this rule.

□

5.6 Search Space Oriented Manipulation

A reasonable heuristic is that the size of the search space generated by an expression is related to the operators used, since each operator carries some computation overhead. For example, consider the expression:

```
(r ⋁ s) ⋀ (r ⋁ t)
```

If this expression is used in forward mode then both of the subexpressions `(r ⋁ s)` and `(r ⋁ t)` are applied to the given domain value and, hence, relation `r` is applied twice. In an expression, each compound designation produces some values necessary for the expression's result. Therefore, if the number of compound designations (i.e. operators) in an expression can be reduced while retaining its meaning then the overhead of computation may also be reduced. This does not decrease the set of values produced as result but does decrease the set of intermediate values needed to produce that result, i.e. the search space of the definition. However, the extent to which computation is saved is unpredictable because lazy evaluation is used. For example, the above expression can be reduced to:

```
r ⋁ (s ⋀ t)
```

When this expression is lazily evaluated in forward mode the two subexpressions applied are `r` and `(s ⋀ t)` — relation `r` is applied once. Application of `r` involves less computation than application of `(r ⋁ s)`.

5.6.1 Rules for Search Space Manipulation

The above expression reduction is described by the distribution law:

```
(r ⋁ s) ⋀ (r ⋁ t) = r ⋁ (s ⋀ t)
```

Any law in which the expression on the left hand side uses more operators than the expression on the right hand side and in which the right hand side is linear (i.e. contains no repeated law variables) can be used as a rewrite rule for attempting to reduce computation. Such rules are termed *search space improving*.

To ensure search space manipulation preserves the representation of any expression it processes, only those rules that are *representation maintaining* and reduce the number of operators are used.

5.6.2 Termination of Search Space Manipulation

Whenever an expression is rewritten by a search space rule, its size must be reduced since the expressions on the right hand sides of rules are linear. Therefore, since expressions are of finite size, termination is guaranteed.

5.6.3 Methods for Search Space Manipulation

Unlike representation manipulation, search space manipulation has no tangible goal to strive for — the aim is to reduce expression size as much as possible.

Basic values cannot be rewritten. A tuple is manipulated by applying manipulation to each of its constituent elements. The only expression form that can be directly rewritten is a compound relation designation `opA argA`; the method for this is constructed from two smaller methods, which are applied in sequence:

1. Repeatedly apply rewrite rules to the whole expression `opA argA` until no more rules are applicable. If this method succeeds then the expression is reduced either to a simple expression that contains no operators, in which case manipulation ceases, or to a simpler compound designation, `opB argB`, in which case the next method is applied.

2. Attempt to improve the search space of `argB` using the first method. If this method successfully applies rewrites to create a new, reduced, expression `argC` then the first method is re-invoked on the whole expression `opB argC`.

The algorithm for search space methods is given in Figure 5.8.

5.7 Manipulation of a Drusilla Program

Typed representation analysis of a Drusilla program proceeds in the usual fashion. When each maximally strong component is analysed, representation manipulation is invoked for any defining expression that cannot be given a representation. The goal for this manipulation is merely to generate a representation for the expression without care as to what that representation is.

In a component the representation available for each definition is dependent on the representations of the other definitions because of the mutual recursion between them. Ideally the goal for manipulation of a definition would be a representation that improves representation possibilities for the other definitions. Unfortunately, however, this representation cannot be known in advance. If one defining expression cannot be represented then neither can any definition within that component, so manipulation ceases, the program is rejected as unusable and it is left to the programmer to re-formulate definitions using mode information.

Section 5.2.1 states that representation and search space manipulation are separated into phases since the latter is of no practical value for a program unless it is fully represented. However, this is compromised slightly in the Drusilla

```
searchOriented = structSearch $1Meth searchTup $1Meth noSearch

structSearch = rtclSearch $sOptMeth searchCompound
                          $sOptMeth rtclSearch

rtclSearch = rtclMeth searchRewrite

searchCompound (sub,E op tr arg,goal)
   = rtclSearch (newSub,E op tr newArg,goal),      succ
   = (False,sub,E op tr arg),                      otherwise
     where
     (succ,newSub,newArg) = searchOriented (sub,arg,goal)
searchCompound (sub,exp,goal) = (False,sub,exp)

searchTup (sub,Tu tup,goal) =
   (True,sub,Tu newTup)
   where
   newTup = map (searchTupEl sub goal) tup
searchTup (sub,exp,goal) = (False,sub,exp)

noSearch (sub,exp,goal) = (True,sub,exp)

searchRewrite = rewriteExp searchImprove

searchTupEl :: trSub -> rewriteGoal -> relExp -> relExp
searchTupEl sub goal exp =
   newExp
   where
   (dummySucc,dummySub,newExp) = searchOriented (sub,exp,goal)
```

Figure 5.8: Methods for search space manipulation

system to simplify transformation. When a program is analysed representation manipulation is applied to each defining expression that has no representation. If a representation can be generated for the expression then search space manipulation is invoked. It is invoked regardless of whether representation manipulation is needed.

5.8 Conclusion

This chapter has described how laws of relational equivalence can be used as directed rewrite rules for the manipulation of Drusilla expressions to achieve two main goals:

1. most importantly, to improve representations, especially for expressions without representation;

2. to improve run-time performance by pruning the search space;

Typed representation inference is applied to the laws to ensure the rules extracted improve representations. The manipulation strategy uses meta-level inference to control search.

The results of this research are disappointing, particularly for representation manipulation for which it is difficult to find good examples since few rules improve representation.

An expression is given $\perp_{TR}$ by the application of a typed representation rule for some operator within that expression. The typed representation rules for an operator only yield $\perp_{TR}$ for those argument representations for which that operator is undefined. Table 5.3 shows the argument representations for which each operator is undefined. This table reflects the limit of what is computable with relational operators. Representation manipulation has only marginal success in transforming away such operators because comparatively few laws are applicable. Most laws apply to operators that are defined for all argument representations and few apply to those operators that are only defined for extensional relations, for example range (`rng`), cardinality (`card`) and image of a (extensionally represented) set under a relation (`img`). This poses a fundamental problem for representation manipulation.

Fortunately, in practice, few expressions are without representation since comparatively few operators are undefined for comparatively few argument representations. Moreover, all programs presented in chapters 3 and 7 are executable in the Drusilla system without any manipulation.

dom :: {	SF[CF[A $\leftrightarrow$ B] $\leftrightarrow$ $\perp_{TR}$] }
rng :: {	SF[SF[A $\leftrightarrow$ B] $\leftrightarrow$ $\perp_{TR}$], SF[CF[A $\leftrightarrow$ B] $\leftrightarrow$ $\perp_{TR}$] }
card :: {	SF[SF[A $\leftrightarrow$ B] $\leftrightarrow$ $\perp_{TR}$], SF[CF[A $\leftrightarrow$ B] $\leftrightarrow$ $\perp_{TR}$] }
; :: {	SF[(CF[A $\leftrightarrow$ B] $\times$ SF[B $\leftrightarrow$ C]) $\leftrightarrow$ $\perp_{TR}$], SF[(CF[A $\leftrightarrow$ B] $\times$ CF[B $\leftrightarrow$ C]) $\leftrightarrow$ $\perp_{TR}$] }
img :: {	SF[(SF[A $\leftrightarrow$ B] $\times$ CF[A $\leftrightarrow$ un]) $\leftrightarrow$ $\perp_{TR}$], SF[(CF[A $\leftrightarrow$ B] $\times$ SF[A $\leftrightarrow$ un]) $\leftrightarrow$ $\perp_{TR}$], SF[(CF[A $\leftrightarrow$ B] $\times$ CF[A $\leftrightarrow$ un]) $\leftrightarrow$ $\perp_{TR}$] }
@ :: {	SF[(AL[A $\leftrightarrow$ B] $\times$ CF[A $\leftrightarrow$ B]) $\leftrightarrow$ $\perp_{TR}$], SF[(SF[A $\leftrightarrow$ B] $\times$ CF[A $\leftrightarrow$ B]) $\leftrightarrow$ $\perp_{TR}$], SF[(CF[A $\leftrightarrow$ B] $\times$ CF[A $\leftrightarrow$ B]) $\leftrightarrow$ $\perp_{TR}$] }

Table 5.3: Undefined operator typed representations

Chapter 6

Architecture of the Drusilla System

6.1 Introduction

Chapters 4 and 5 described techniques designed to improve the implementation of relational programming by widening the *representation bottleneck* of RPL and GREL. The interaction of these techniques creates a far more sophisticated and complex implementation of relational programming. A data flow diagram describing the system is presented in Figure 6.1. This chapter describes the architecture of this system. Section 6.2 describes the preliminary processing of calculus type inference rules, typed representation inference rules and laws. Section 6.3 describes the parser for the Drusilla language. Section 6.4 discusses the interaction of the various semantic analysis techniques. Section 6.5 presents the framework used for the Miranda-level definition of the primitive operators. Section 6.6 explains the evaluation process for Drusilla expressions and run-time queries of relations. Section 6.7 explains how Drusilla programs can be compiled to Miranda programs. Section 6.8 draws conclusions about the architecture.

6.2 Preliminary Processing of Inference Rules and Laws

The inference rules for calculus type inference and typed representation inference must be defined before program analysis can commence. Similarly the laws for algebraic manipulation must be analysed before manipulation can commence.

6.2.1 Type Signatures and Inference Rules

Calculus type and typed representation rules are defined using the same mechanism but in separate text files. The set of of typed representation signatures for each operator is enclosed in braces with each rule on a separate line. For example:

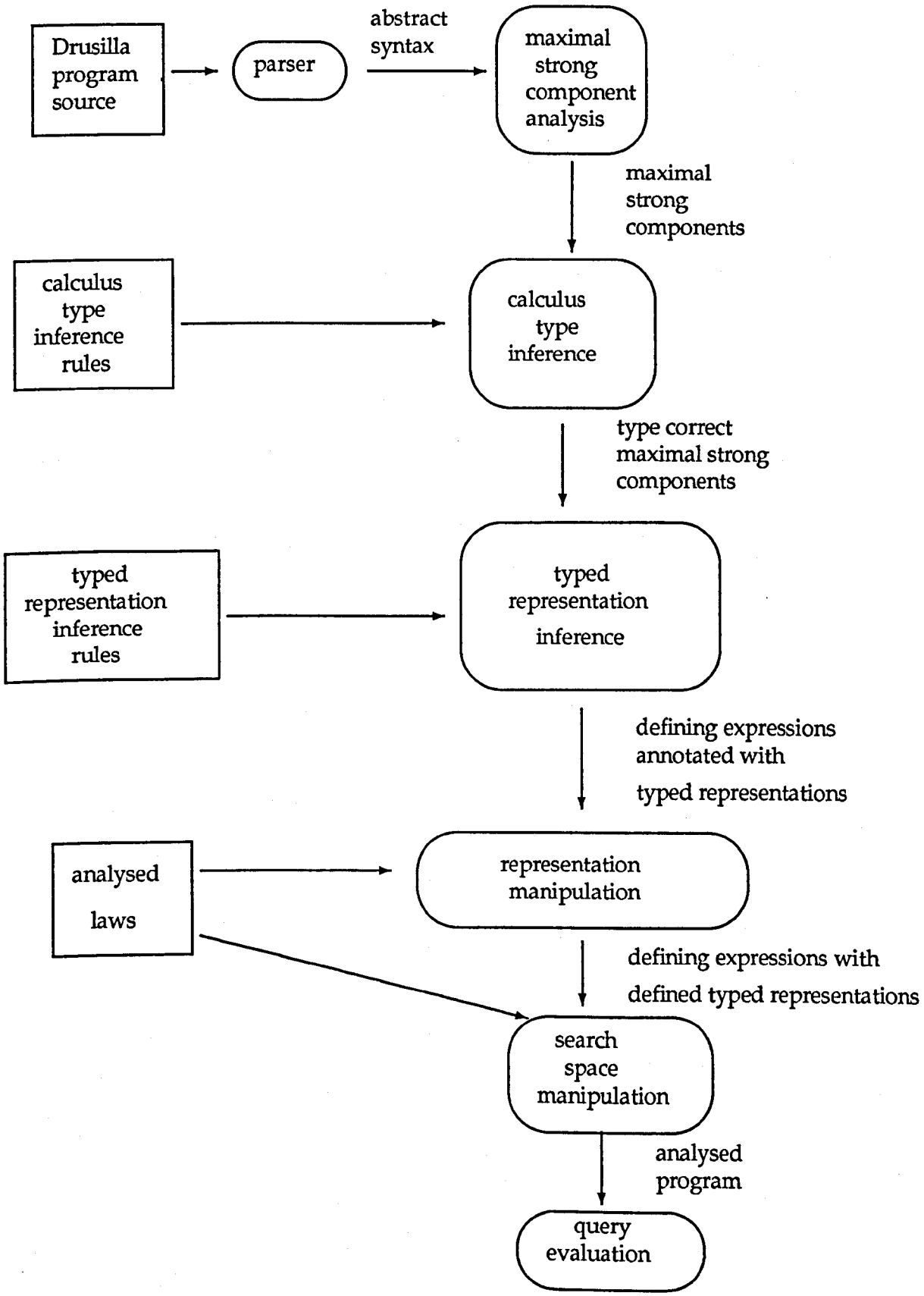

Figure 6.1: Data flow in the Drusilla interpreter

```
; :: { SF[(AL [A <-> B] x AL[B <-> C]) <-> (AL [A <-> C])],
     SF[(AL[A <-> B] x SF[B <-> C]) <-> AL[A <-> C]],
     SF[(AL[eA <-> B] x CF[B <-> C]) <-> CF[eA <-> C]],
     SF[(SF[A <-> eB] x AL[eB <-> C]) <-> SF[A <-> C]],
     SF[(SF[A <-> B] x SF[B <-> C]) <-> SF[A <-> C]],
     SF[(SF[A <-> B] x CF[B <-> C]) <-> CF[A <-> C]],
     SF[(CF[A <-> B] x AL[B <-> eC]) <-> CF[A <-> eC]],
     SF[(CF[A <-> B] x SF[B <-> C]) <-> bottom],
     SF[(CF[A <-> B] x CF[B <-> C]) <-> bottom] }
```

Equality variables are prefixed with an e.

The calculus rule for each operator is written on a separate line. For example:

```
; :: (A <-> B x B <-> C) <-> (A <-> C)
```

The processing mechanism is described here for calculus type signatures but also applies to typed representation signatures. The text file is parsed to generate a calculus type signature for each operator. Calculus types are represented by the algebraic datatype shown in Figure 6.2.

`calcType ::=`	calculus types datatype
`NumC`	number type
`| StringC`	string type
`| UnC`	unit type
`| VarC num`	type variable
`| TuC [calcType]`	tuple type
`| RelC calcType calcType`	relational type

Figure 6.2: Miranda algebraic datatype defining calculus types

For example, the above calculus type of relation composition is parsed into the abstract syntax:

```
RelC   (TuC [RelC (VarC 1) (VarC 2),RelC (VarC 2) (VarC 3)])
       (RelC (VarC 1) (VarC 3))
```

Such a signature can also be used as an inference rule that is applicable when that operator constructs a compound relation designation. For example, the relation composition `r ; s` will be parsed into the abstract syntax for a relational expression `Comp (Tu [r,s])`. Suppose the tuple `Tu [r,s]` is given the calculus type:

```
TuC [RelC StringC NumC,RelC NumC UnC]
```

The expression's calculus type is generated by the signature for composition:

```
RelC StringC UnC
```

This is generated by unifying the type for the tuple with the domain type for composition. i.e. the the two expressions:

```
TuC [RelC (VarC 1) (VarC 2), RelC (VarC 2) (VarC 3)]
  TuC [RelC StringC    NumC,    RelC    NumC      UnC]
```

The unification produces a substitution which is applied to the range type for composition to give the above composition expression type. In this way each operator's calculus type signature can be used as a calculus type inference rule.

6.2.2 Processing Laws

The laws for algebraic manipulation are also specified in text files. Each law is written as the equivalence of two expressions. For example:

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

From each law a relation definition is derived in which the defining expression is the pair of expressions that form the law and the formal parameters are the variables occurring in the law. Typed representation analysis is then applied to this definition as described in chapter 5.

6.3 The Parser

The parser is conventionally composed of a lexical analyser, which recognises the language tokens (basic values and operator symbols), and a syntactic analyser, which builds those tokens into the abstract syntax defined as algebraic datatype `relExp` in Figure 4.6.

Both analysers are constructed using the *'let form follow function'* idea of Fairbairn [33]. This style of parser is based on the notion of 'gluing' [47] smaller parsers together to form bigger parsers. The glue is provided by higher-order functions for the alternation (`$l`), sequencing (`$s`), transitive closure (`$tcl`) and reflexive transitive closure (`$rtcl`) of parsers.

The definition of the parser looks like the BNF that formally defines the grammar. Consequently it is easy to modify the parser as the language evolves. Each grammar rule may be associated to a build function, which constructs the abstract syntax. The Miranda definition of the parser (without the abstract syntax build functions) is shown in Figure 6.3.

6.4 Semantic Analysis

The semantic analysis breaks down into three sequential phases:

```
compExp      = (simpleExp $s rtcl (binaryOp $s simpleExp))
                $as buildCompExp
simpleExp    = (opt unaryOp $s argExp)
                $as buildSimpleExp
argExp       = opSection $l optCurryRel $l domain
domain       = number $l unit $l string $l extensRel $l tuple
opSection    = (lex SqOpen $s (leftSect $l rightSect) $s
                lex SqClose)  $as buildOpSection
leftSect     = (argExp $s primOp)
                $as buildLeftSection
rightSect    = (primOp $s argExp)
                $as buildRightSection
optCurryRel  = (opt curryTuple $s curryObj $s opt curryTuple)
                $as buildOptCurryRel
curryObj     = (identifier $l brackExp $l elementOp)
                $as buildCurryObj
elementOp    = (lex SqOpen $s primOp $s lex SqClose)
                $as buildElementOp
brackExp     = (lex ParOpen $s compExp $s lex ParClose)
                $as buildTuple
tuple        = (lex ParOpen $s relExpSeq $s lex ParClose)
                $as buildTuple
relExpSeq    = (compExp $s tcl (lex Comma $s compExp))
                $as buildRelExpSeq
curryTuple   = (lex ParOpen $s currySeq $s lex ParClose)
                $as buildTuple
currySeq     = (curryItem $s tcl (lex Comma $s curryItem))
                $as buildRelExpSeq
curryItem    = (lex Uscore $l compExp)
                $as buildCurryItem
extensRel    = (lex BrOpen $s opt pairSeq $s lex BrClose)
                $as buildExtensRel
pairSeq      = (domainPair $s rtcl (lex Comma $s domainPair))
                $as buildPairSeq
domainPair   = (lex ParOpen $s compExp $s lex Comma $s compExp
                $s lex ParClose)  $as buildDomainPair
```

Figure 6.3: Miranda code for the Drusilla expression parser

Maximal strong component isolation: once the program has been parsed and the abstract syntax constructed the maximal strong components of the program reference graph are isolated. A *reduced reference graph* is obtained by collapsing each component in the full graph to a single node. The reduced graph is used in the following analysis phases.

Calculus type inference: this ensures that each defining expression has a valid meaning in the world of relational algebra. If a program contains a definition that is calculus incorrect then the program is rejected. This ensures that later computationally expensive analysis techniques are never applied to incorrect programs.

The calculus type inference algorithm is the conventional Milner [75] type inference algorithm used in functional languages. The type inference rules used are dictated by the relational operators. These rules are given in Figure 6.4.

Typed representation inference and algebraic manipulation: typed representation analysis is applied to strong components in the reverse of their depth-first ordering. This ensures that each definition is analysed before the definitions that reference it non-recursively. Representation 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 program analysis ceases.

Search space manipulation is automatically applied to each defining expression that has a defined typed representation in an attempt to improve its search space characteristics.

6.5 Defining the Operators in Miranda

The definition of the relational operators in the Miranda implementation must preserve both ad-hoc and parametric polymorphism. Each operator must be defined for all possible argument representations. Ad-hoc polymorphism is implemented by overloading each operator giving it separate definitions for each particular argument representation. Parametric polymorphism is preserved by ensuring each definition is applicable to all relations of the given representation regardless of their domain and range types.

In order for Miranda to handle all definitions of an operator uniformly, the definitions must all possess the same Milner type. Therefore the universe of Drusilla values is defined as an algebraic datatype, `domain`, shown in Figure 6.5.

All built-in operators of Drusilla are defined over this domain of values. For example, the definition of arithmetic plus, +, as a point-to-point function:

```
plusF :: domain -> domain
plusF (T [N x,N y]) = N (x + y)
```

$$\frac{t :: A \leftrightarrow B}{\begin{array}{ll} \texttt{dom}\ t :: A \leftrightarrow un & \texttt{rng}\ t :: B \leftrightarrow un \\ \texttt{card}\ t :: num & \texttt{inv}\ t :: B \leftrightarrow A \\ \texttt{neg}\ t :: A \leftrightarrow B & \texttt{set}\ t :: (A \times B) \leftrightarrow un \end{array}}$$

$$\frac{r :: A \leftrightarrow B \qquad s :: A \leftrightarrow B}{\begin{array}{ll} r \setminus/ s :: A \leftrightarrow B & r /\!\setminus s :: A \leftrightarrow B \\ r \setminus s :: A \leftrightarrow B & r\ @\ s :: A \leftrightarrow B \end{array}}$$

$$\frac{s :: A \leftrightarrow un \qquad r :: A \leftrightarrow B}{\begin{array}{ll} s \ll r :: A \leftrightarrow B & s \ll\!- r :: A \leftrightarrow B \\ r\ \texttt{img}\ s :: B \leftrightarrow un & \end{array}}$$

$$\frac{p :: A \leftrightarrow B \qquad q :: B \leftrightarrow un}{\begin{array}{ll} p \gg q :: A \leftrightarrow B & p \mathrel{->} q :: A \leftrightarrow B \end{array}}$$

$$\frac{r :: A \leftrightarrow B \quad s :: B \leftrightarrow C}{r\ ;\ s :: A \leftrightarrow C}$$

$$\frac{r :: A \leftrightarrow B \quad s :: A \leftrightarrow C}{r\ \#\ s :: A \leftrightarrow (B \times C)}$$

$$\frac{r :: A \leftrightarrow B \qquad s :: C \leftrightarrow D}{r\ ||\ s :: (A \times C) \leftrightarrow (B \times D)}$$

Figure 6.4: Calculus type inference rules

```
domain ::=                                  universe of values datatype
     N num                                  number values
   | S [char]                               string values
   | U unitType                             unit value
   | T [domain]                             tuple of values
   | R rel                                  relational values
   | DUM                                    anonymous value for
                                            relation specialisation

unitType ::= Unit                           unit value datatype

rel ::=                                     relational value datatype
     Al [(domain,domain)]                   relation as association list
   | Sf (domain -> [domain])                relation as set-valued function
   | Cf ((domain,domain) -> bool)           relation as characteristic function
```

Figure 6.5: Miranda algebraic datatype defining universe of Drusilla values

This form of function implements compound relation designations. For example, the Drusilla compound designation `a + b` is represented by the expression:

```
plusF (T [N a, N b])
```

This can be treated as a new Drusilla value since it has type `domain`. When plus is used as an elementary designation, `[+]`, it must be coerced to be included in `domain`:

```
plusSf :: domain -> [domain]
plusSf = mklist . plusF
mklist x = [x]
```

This is included in `domain` as the value: `R (Sf plusSf))`.

The relational operators can be defined in the same manner. Consider, for example, relation composition (;). The seven argument representations for which composition can be defined are identified in Figure 4.5. It would be reasonable to expect seven different definitions of composition to be needed — one for each combination of argument representations. However, by using argument coercions, only five are needed: `compAl1`, `compAl2`, `compSf`, `compCf1` and `compCf2`, shown in Figure 6.6. These definitions use the library functions shown in Figure 6.7.

The typed representation of any expression `op arg` is created by applying one of `op`'s inference rules to the typed representation of `arg`. Each of these rules corresponds to a definition of `op`. For example, Figure 6.8, shows the typed representation inference rules and corresponding definitions for the composition operator. Each rule introduces the appropriate equality constraint (as an equality

```
compAl1 (T [relR,relS]) =
   R (Al [(a,d) // (a,b) <- r; (c,d) <- s; domainEquiv b c])
   where
   (R (Al r)) = relR
   (R (Al s)) = relS

compAl2 (T [relR, R (Sf s)]) =
   R (Al [(a,c) | (a,b) <- r; c <- s b])
   where
   (R (Al r)) = relR

compSf (T [R (Sf r), R s]) = R (Sf (mapcat (coerceToSf s) . r))

compCf1 (T [R r, R s]) =
   R (Cf f)
   where
   f (x,y) = exists (coerceToCf s . rpair y) (coerceToSf r x)

compCf2 (T [relR, relS]) =
   R (Cf f)
   where
   f (x,y) = exists (r . pair x) (domAlFilter (y =) s)
   (R (Cf r)) = relR
   (R (Al s)) = relS

|| general functions
mapcat f = concat . map f
pair x y = (x,y)
rpair x y = (y,x)
exists = or . map f
```

Figure 6.6: Miranda definitions of the composition operator

```
domainEquiv :: domain -> domain -> bool
domainEquiv (N x) (N y) = x = y
domainEquiv (S x) (S y) = x = y
domainEquiv (U x) (U y) = True
domainEquiv (T tuX) (T tuY) =
   # tuX = # tuY  &  and (map2 domainEquiv tuX tuY)
domainEquiv (R (Al r)) (R (Al s)) = setEqualH domPairEquiv r s

domPairEquiv :: (domain,domain) -> (domain,domain) -> bool
domPairEquiv (lftX, rhtX) (lftY, rhtY) =
   domainEquiv lftX lftY & domainEquiv rhtX rhtY

domAlFilter :: (domain -> bool) -> [(domain,domain)] -> [domain]
domAlFilter p = map fst . filter (p . snd)

coerceToSf :: domain -> domain -> [domain]
coerceToSf (Al a) = lookup a
coerceToSf (Sf f) = f

lookup :: [(domain,domain)] -> domain -> [domain]
lookup xs x = [b | (a,b) <- xs; a = x]

coerceToCf :: rel -> (domain,domain) -> bool
coerceToCf (Cf p) (x,y) = p (x,y)
coerceToCf (Al s) (x,y) = setMemH domPairEquiv (x,y) s
coerceToCf (Sf f) (x,y) = setMemH domainEquiv y (f x)

setMemH :: (* -> * -> bool) -> * -> [*] -> bool
setMemH equals item = exists (equals item)
```

Figure 6.7: Library of operations over `domain` values

variable) for a relation whenever it is coerced in the corresponding definition, as explained in chapter 4. The typed representation rules ensure that equality is defined for relation domain and range elements whenever necessary for the coercions applied by operator definitions.

For the Drusilla interpreter the typed representation inference rules were coded manually to match the definitions. It would be more satisfactory if they could be automatically generated from the operator definitions. The possibility for this is discussed in section 6.7.

$$\texttt{compAl1}\quad \frac{r :: AL[A \leftrightarrow B^{=}] \quad s :: AL[B^{=} \leftrightarrow C]}{r\,;s :: AL[A \leftrightarrow C]}$$

$$\texttt{compAl2}\quad \frac{r :: AL[A \leftrightarrow B] \quad s :: SF[B \leftrightarrow C]}{r\,;s :: AL[A \leftrightarrow C]}$$

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

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

$$\texttt{compCf1}\quad \frac{r :: SF[A \leftrightarrow B] \quad s :: CF[B \leftrightarrow C]}{r\,;s :: CF[A \leftrightarrow C]}$$

$$\texttt{compCf1}\quad \frac{r :: AL[A^{=} \leftrightarrow B] \quad s :: CF[B \leftrightarrow C]}{r\,;s :: CF[A^{=} \leftrightarrow C]}$$

$$\texttt{compCf2}\quad \frac{r :: CF[A \leftrightarrow B] \quad s :: AL[B \leftrightarrow C^{=}]}{r\,;s :: CF[A \leftrightarrow C^{=}]}$$

Figure 6.8: Definitions and corresponding rules for relation composition

6.6 Program Interpretation

If analysis of a program is successful then all of its definitions have a defined typed representation, and expression and query evaluation may commence.

6.6.1 Evaluation of a Drusilla Expression

A Drusilla expression is evaluated by mapping its abstract syntax to an equivalent Miranda expression, which is evaluated by the Miranda interpreter. The Miranda

expression generated uses only `domain` values and returns as result a `domain` value.

The functions that perform expression interpretation are defined in Figure 6.9. The main interpreting function is `expEval`. The evaluator must carry around a state variable `substEnv`, which binds the names of program definitions to their defining expressions and formal parameters to argument expressions.

Program defined relations are evaluated by `evaluateRel`, which uses two functions that are not shown, `relSfEval` and `relCfEval`. These evaluate relations that are represented by set-valued functions and characteristic functions respectively. They bind the relation formal parameters to values in the given domain tuple element and evaluate the defining expression using function `expEval`. The function `getExpTR` returns the typed representation assigned to a given expression and is defined in Figure 4.9. Relations defined in extension are evaluated by `extRelEval`.

Function `getOpDefn` selects the appropriate definition of each operator that forms a compound relation designation. Application of the function `getOpDefn` takes the form:

```
getOpDefn op argumentTR resultTR
```

where `op` is a a relational operator, `argumentTR` is a typed representation of an argument, `arg` and `resultTR` is the typed representation for the compound relation designation `op arg`. The result of this application is the required Miranda definition of `op`. The required definition for any elementary designation operator is generated by `getOpSF`.

This evaluation of operators and expressions forms the *interpretive overhead.* From one perspective this evaluation may be thought of as further representation selection — the representation selected for an operator is the appropriate definition and the representation selected for a Drusilla expression is the corresponding Miranda expression. It would be desirable for this extra representation selection to be performed at program analysis time and section 6.7 discusses this possibility.

6.6.2 Interpretation of Run-time Relation Queries

At execution time program definitions are queried in one of the three modes show, forward, or test. The Drusilla interpreter determines the modes of the query according to its syntax. The query evaluation function is defined in Figure 6.10.

Show mode: the query is a Drusilla expression. The expression is evaluated by the interpreting function shown in Figure 6.9.

Forward mode: these queries takes the form: `relExp domExp` where the result is the set of range elements to which `domExp` relates under `relExp`. The query is parsed into the abstract syntax:

```
Forward (Tu [relExp,domExp])
```

```
expEval :: substEnv -> relExp -> domain
expEval env (No n) = N n
expEval env (Ch c) = C c
expEval env (Bo b) = B b
expEval env Ut = U Unit
expEval env (Dum tr) = DUM
expEval env (Ext reln tr) = extRelEval env reln
expEval env (Tu tup) = T (map (expEval env) tup)
expEval env (RO op tr) = getOpSF op tr
expEval env (E op resultTR arg) =
   getOpFun op (getExpTR arg) resultTR (expEval env arg)
expEval env  (Rel name tr) =
   evaluateRel env (name,tr) (lookupRelExp (name, tr) env)
expEval env  (Fpar name tr) = lookupEnvPar name env

getOpSF :: relOp -> typedRep -> domain
getOpSF op (RTR SF domTR rngTR) =
   R (Sf (mklist . getOpFun op domTR rngTR))

extRelEval :: substEnv -> [(relExp, relExp)] -> domain
extRelEval env = R . Al . map (extPairEval env)

extPairEval :: substEnv -> (relExp, relExp) -> (domain,domain)
extPairEval env (dom, rng) = (expEval env dom, expEval env rng)

evaluateRel :: substEnv -> (ident,typedRep) ->
                  (relExp,relExp) -> domain
evaluateRel env (name,typRep) (Dum tr, defExp) =
   expEval env defExp
evaluateRel env (name, RTR SF domTR rngTR) (parsExp, defExp) =
   R (Sf (relSfEval env parsExp defExp))
evaluateRel env (name, RTR CF domTR rngTR) (parsExp, defExp) =
   R (Cf (relCfEval env parsExp defExp))
```

Figure 6.9: Evaluating function for Drusilla expressions

```
interExpEval :: substEnv -> relExp -> [char]
interExpEval env (E FORW tr application) =
  getForwFun (getExpTR application) (expEval env application)
interExpEval env (E TEST tr pred) =
  getTestFun (getExpTR pred) (expEval env pred)
interExpEval env expr = showDom (expEval env expr)
```

Figure 6.10: Evaluating function for relation queries

where `Forward` is a special Drusilla operator that the programmer is not permitted to see. In order for relation `relExp` to be used in forward mode, it must have a set-valued function or association list representation and its domain must have the same typed representation as `domExp`. Therefore, the abstract expression is given a typed representation by applying one of the two inference rules for `Forward`:

$$(\text{AL}[A^{=} \leftrightarrow \text{B}] \times A^{=}) \leftrightarrow \text{B}$$
$$(\text{SF}[\text{A} \leftrightarrow \text{B}] \times \text{A}) \leftrightarrow \text{B}$$

This query is evaluated by function `getForwFun` defined in Figure 6.11.

Test mode: these queries take the form: `domExp relExp rngExp`. If `relExp` relates `domExp` to `rngExp` then the answer to the query is `True` otherwise the answer is `False`. The query is parsed into the abstract syntax:

```
Test (Tu [relExp,Tu [domExp,rngExp]])
```

where `Test`, like `Forward`, is a Drusilla operator that the programmer is not permitted to see. In order for relation `relExp` to be used in test mode it must have a relational representation that supports test mode, its domain must have the same typed representation as `domExp` and its range must have the same typed representation as `rngExp`. The abstract expression is given a typed representation using one of the three inference rules for `Test`:

$$(AL[A^{=} \leftrightarrow B^{=}] \times (A^{=} \times B^{=})) \leftrightarrow string$$
$$(SF[A \leftrightarrow B^{=}] \times (A \times B^{=})) \leftrightarrow string$$
$$(CF[A \leftrightarrow B] \times (A \times B)) \leftrightarrow string$$

The query is evaluated by function `test` defined in Figure 6.12. This query is evaluated by function `getTestFun` defined in Figure 6.12.

```
getForwFun :: typedRep -> domain -> [char]
getForwFun (TuTR [RTR AL x y, z]) arg
   = "rel domain and arg types do not unify",   succ = False
   = showResSet arg,                            showable x
   = "mode error --- showable relation",        otherwise
     where
     (succ, subs) = typRepUnify x z
getForwFun (TuTR [RTR SF x y, z]) arg
   = showResSet arg,                            succ = True
   = "rel domain and arg types do not unify",   otherwise
     where
     (succ, subs) = typRepUnify x z
getForwFun (TuTR [RTR CF x y, z]) arg =
   "mode error --- test mode relation"
getForwFun typ arg =
   error ("getForwFun --- typRep is " ++ showExpTR typ)

showResSet :: domain -> [char]
showResSet = showFunction . forwF

forwF :: domain -> [domain]
forwF (T [R reln, val]) = coerceToSf reln val
```

Figure 6.11: Evaluation of a query in forward mode

```
getTestFun :: typedRep -> domain -> [char]
getTestFun (TuTR [RTR AL x y, TuTR [domTR,rngTR]]) arg
   = "domain/range and arg types don't unify",
                                    succ = False
   = testF arg,                     showable x & showable y
   = error "showable relation can't be used in test mode",
                                    otherwise
     where
     (succ, subs) = typRepUnify (TuTR [x,y])
                                (TuTR [domTR,rngTR])
getTestFun (TuTR [RTR SF x y, TuTR [domTR,rngTR]]) arg
   = "domain/range and arg types don't unify",
                                    succ = False
   = testF arg,                   showable x & showable y
   = error "forw rel can't be used in test mode",
                                    otherwise
     where
     (succ, subs) = typRepUnify (TuTR [x,y])
                                (TuTR [domTR,rngTR])
getTestFun (TuTR [RTR CF x y, TuTR [domTR,rngTR]]) arg
   = testF arg,                                    succ = True
   = "domain/range and arg types don't unify", otherwise
     where
     (succ, subs) = typRepUnify (TuTR [x,y])
                                (TuTR [domTR,rngTR])

testF :: domain -> [char]
testF (T [R reln, T [x,y]])
   = "TRUE",     succ
   = "FALSE",    otherwise
     where
     succ = coerceToCf reln (x,y)
```

Figure 6.12: Evaluation of a query in test mode

6.7 Compiling Drusilla Programs

Typed representation inference annotates each expression in a Drusilla program with a typed representation. The interpreter uses these annotations to map each Drusilla expression to a Miranda `domain` value at run-time. This *interpretive overhead* can be eliminated by replacing Drusilla expressions with Miranda expressions at program analysis time.

Typed representation inference of each definition produces a set of versions, each of which is associated with a typed representation. Each definition version is associated with a unique identifier in a symbol table. Each Drusilla expression can then be replaced by the text of the corresponding Miranda expression:

- Basic values are replaced by Miranda values: numbers remain numbers, lists of characters replace strings, `True` replaces `Unit`, tuples remain tuples.
- Each reference to a definition is replaced by the identifier appropriate to its typed representation, which is found from the symbol table.
- Each Drusilla operator is replaced by the textual name of the Miranda definition appropriate to its typed representation.

The result of applying this *representation substitution* to a Drusilla program is the text of a Miranda program, which may be written as a text file and compiled by the Miranda system. Therefore, it constitutes a compiler for Drusilla that compiles down to Miranda. The Miranda program produced is not defined over the `domain` of Drusilla values, but over the corresponding Miranda values. Each operator must be redefined for Miranda values with one definition for each combination of argument representations.

The typed representation inference rules for the Drusilla system were created manually to match the `domain` definitions of the operators. For several operators, one definition applies to several argument representations (by applying coercions), and remains type correct because of the `domain`. Consequently, several typed representation inference rules correspond to one definition.

However, under the new compilation scheme, each of an operator's definitions can apply to only one argument representation because Miranda values are used. Hence each definition for an operator corresponds to exactly one typed representation inference rule. Moreover this rule can be automatically generated by applying Milner type inference to that definition.

This has important ramifications for the Drusilla compiler. If the typed representation inference rules for each operator are obtained from the types inferred for its definitions then the type and representation constraints they capture are correct. When typed representation inference is applied to a Drusilla program using these rules, it generates a typed representation for each program expression. When *representation substitution* is applied, each Drusilla expression is mapped into a Miranda expression according to its associated typed representation.

Chapter 4 explained that typed representation inference algorithm is sound and complete because it is based on Milner's algorithm. The Miranda object program produced by the compiler is a representation of the Drusilla source program.

Soundness implies that the Miranda program generated by the compiler is derivable under the inference system formed by the typed representation inference rules. Since these rules are derived from the operator definitions that the Miranda program uses, type correctness of the Miranda program follows from typed representation correctness of the Drusilla program. Completeness implies that if a Miranda program is derivable from the Drusilla program under the inference system then the compiler will generate that program.

Drusilla programs run considerably faster when compiled than when interpreted. For example, chapter 7 will present a Drusilla solution to the eight queens problem. When this program is interpreted it runs overnight to produce all 92 solutions, but, when compiled all solutions can be produced in about five minutes. Figure 6.13 shows the algorithm for expression compilation. The result of compiling an expression is a triple:

```
(miraCode,localDefns,unusedLocalNames)
```

The expression is compiled to a list of characters denoting a Miranda expression, `miraCode`. Relation specialisations are compiled as λ-abstractions, which are represented in Miranda as `where` blocks. The local definitions needed to support the compiled Drusilla expression are represented by `localDefns`. Each local definition is represented by a list of characters, hence `localDefns` is a list of lists of characters. An infinite list of names for local definitions is passed to `compileExp` as argument `names`. Those that were not used are returned as `unusedLocalNames`.

Appendix A shows several of the Drusilla programs introduced in this thesis alongside the Miranda translations produced by the Drusilla compiler.

6.8 Conclusion

This chapter described the architecture of the Drusilla interpreter and explained how the techniques presented in chapter 4 and chapter 5 are combined to form a framework for a new, powerful implementation of relational programming. Chapter 7 evaluates this implementation in terms of the freedom of expression it permits relational programming to possess.

```
compileExp sym names (No numb) = (shownum numb,[],names)
compileExp sym names (St string) =
   ('\"' : string ++ "\"",[],names)
compileExp sym names Ut = ("True",[],names)
compileExp sym names (Ext reln tr) =
   (compileExtReln sym reln,[],names)
compileExp sym names (Tu tup) = compileTuple sym names tup
compileExp sym names (RO op tr) = (getOpSF op tr,[],names)
compileExp sym names (E op resultTR arg) =
   (opDefn ++ " (" ++ argCode ++ ")", locals,newNames)
   where
   opDefn = getOpFun op (getExpTR arg)
   (argCode,locals,newNames) = compileExp sym names arg
compileExp sym names (Rel name tr) =
   (lookupRelName sym (name,tr),[],names)
compileExp sym names (Fpar name tr) = (name,[],names)

compileTuple sym names expList =
   (listToTuple objList, locals,newNames)
   where
   (objList,locals,newNames) =
          compileExpList sym names expList
compileExpList sym names =
   foldr (compileListEl sym) ([],[],names)
compileListEl sym el (objCode,locals,names) =
   (newCode : objCode,newLocals ++ locals,newNames)
   where
   (newCode,newLocals,newNames) = compileExp sym names el

compileExtReln sym rel =
   '[' : (mkStringList (map (compileExtEl sym) rel)) ++ "]"
compileExtEl sym (d,r) =
   '(' : dObj ++ "," ++ rObj ++ ")"
   where
   dObj = fst3 (compileExp sym [] d)
   rObj = fst3 (compileExp sym [] r)

listToTuple [x] = x
listToTuple lis = '(' : (mkStringList lis ++ ")")
mkStringList [] = []
mkStringList lis = foldr1 addComma lis
addComma x y = x ++ "," ++ y
```

Figure 6.13: Algorithm for compiling analysed Drusilla expressions

Chapter 7

Evaluation of The Drusilla System

7.1 Introduction

Chapter 4 identified the main weakness in the implementation of RPL as the *representation bottleneck*. This fixed representation scheme creates an unnecessary compromise to relational abstraction and curtails freedom of expression. Chapter 4 and chapter 5 described new techniques for the implementation of relational programming, designed to widen this bottleneck: automatic representation inference and algebraic manipulation of relations. Chapter 6 described the use of these mechanisms in the Drusilla interpreter and the compilation of Drusilla programs into Miranda programs. Their use makes the implementation of Drusilla more complicated than that of RPL, but can be justified if it yields a significant increase in expressive power.

The aim of this thesis is to demonstrate that relational programs are not only expressive but also capture aspects of functional and logic programming. It should be possible to formulate higher-order polymorphic definitions as in a functional language while relying on a type system to ensure mathematical validity of programs. Furthermore the presence of lazy evaluation should permit definition of infinite relational data structures. The presence of logical aspects should permit reasoning about the relationships between entities at a high level. It should also permit concise solutions to problems that involve non-determinism and search based computation.

This chapter examines and evaluates Drusilla as a programming system with respect to the above criteria. Several Drusilla programs are presented and explained at both conceptual and implementation levels.

Section 7.2 describes the declarative and operational reading of Drusilla programs. A good understanding of this section is essential if the following sections and the programs presented in them are to be understood. Section 7.3 draws a comparison between RPL and Drusilla by translating MacLennan's four relational programs [68, 69] into Drusilla. Section 7.5 discusses relational aspects of Drusilla: the handling of sets, recursion over relations, non-determinism, control flow, exception handling, and formal derivation of Drusilla programs from specifications written in the calculus. Section 7.4 discusses functional aspects of Drusilla programs and presents several functional-style programs. Section 7.6

discusses logical aspects of Drusilla programs and example programs illustrate the handling of relationships and state space searching. Section 7.7 draws conclusions about the degree of success of the Drusilla system.

7.2 Declarative and Operational Reading

Bratko [11] observes that Prolog programs have two levels of meaning: *declarative* and *operational*. This is true of any program written in a declarative language. The *declarative meaning* is concerned only with *what* is defined by the program and determines what the output of a program will be. The *operational meaning* (or *procedural meaning*) also determines *how* the output is obtained.

Read declaratively, a program is perceived purely in mathematical terms. A functional program is a collection of function definitions where each definition assigns a name to an expression in λ – *calculus*. A Horn clause logic program is a collection of facts and rules written in first-order predicate calculus formulae. An equational program is a collection of equations — stated expression equivalences. A constraint program [28] comprises relations over some problem domain. The declarative reading of a program is similar to its denotational semantics in that it states program meaning without reference to the underlying computation process.

The operational (or procedural) reading of a declarative program considers not only its mathematical meaning but also its behaviour at execution time. Not only is the meaning of expressions understood, but also the order in which they are evaluated. In a functional language this is the order of evaluation of function applications within expressions, for example, eager or lazy evaluation [85]. For Horn clause logic programs the operational semantics is resolution based theorem proving with some in-built resolution strategy such as LUSH or SLD resolution. Equational languages have been given different operational semantics by different language designers. Hoffman and O'Donnell [41] use term rewriting, Dershowitz and Plaisted [32] use narrowing.

Similarly Drusilla programs may be read declaratively or operationally. The declarative reading of an expression is given by mapping it to the relational calculus replacing each operator occurrence as a compound designation by its calculus definition. Any relation has a separate operational reading, for each mode of use, when evaluated using relational laziness (described in section 7.4.3). Read in *show mode* a relation is viewed as a set of pairs. This is only applicable to definitions where the elements are explicitly listed or where some formula for element generation is given. Read in *test mode* a relation is perceived as a predicate that holds or does not hold between values. A relation read in *map mode* is viewed as a relator mapping domain elements to sets of range elements.

7.3 A Comparison of RPL and Drusilla

MacLennan [68, 69] developed four relational programs to demonstrate the expressive power of his language RPL. This section compares these four programs

with Drusilla translations based on the same algorithms. These programs cannot be understood without definitions of the operators used. The Drusilla operators are shown in chapter 3 in Table 3.2. Tables 7.1 and 7.2 respectively contain MacLennan's definitions of the RPL extensional and intensional relational operators. His definitions of the non-primitives are cyclic in places making exact meaning somewhat unclear. Furthermore, he does not define the operators (`x`) or (`id`) but one assumes that these refer to cartesian product and identity respectively for which the standard definitions apply. Not all the system operators are given — only those relevant to the programs. MacLennan actually defines 70 extensional operators (16 primitive and 54 non-primitive) and 23 intensional operators (13 primitive, 10 non-primitive).

In each example MacLennan's original RPL program is followed by the Drusilla translation. Only a brief description of each algorithm is given; more detailed descriptions are given by MacLennan [68, 69].

7.3.1 Program 1: Word Frequency Table

The programs shown in Figure 7.1 compute frequencies of word occurrences in a word table. Relation `s` is an example word table and definition `freq` computes the frequencies. The algorithm first inverts the word table relation to create a new relation between words and their positions. The unit image (`unimg`) of this relation generates a relation between each word and the set of positions in which that word occurs. The frequencies are derived by taking the cardinality of each occurrence set.

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 RPL and Drusilla programs contain the same number of definitions. In RPL two domain restriction operators are needed — one for extensional relations (`->`), the other for intensional relations (`restrict`) — however, in Drusilla only one (`<<`) is required.

```
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]
```

```
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 7.1: Word frequency programs in RPL (upper) and Drusilla (lower)

Primitive Extensional	
Operator	**Meaning**
t↓x	application
t | u	relative product
t # u	construction
x : y	pair formation
cur t	currying
unc t	uncurrying
ϑ x	unique element selection
size x	cardinality
str t	structure of a relation
t^+	transitive closure
filter p s	{x | x ∈ s ∧ p(x)}
or{true} = true or{false} = false or{true,false} = true	

Non-primitive Extensional	
Operator	**Definition**
α	ϑ . init
ω	ϑ . term
init t	dom t \ rng t
term t	rng t \ dom t
(x,y)	un (x:y)
(x,)	un . (x:)
(,y)	un . (:y)
Δ x	(x,x)
s \ t	dom [∉ t → s × un0]
p → t	filter (p . Hd) t
dom t	img Hd t
rng t	(dom . inv) t
Ω	∉ init t → t
Hd	α . un
Tl	ω . un
x ∈ t	or . img[x =] t
inv t	img [: . (Tl,Hd)!] t
t ! x	@ x $ t

Table 7.1: RPL extensional operators

Primitive Intensional Operators	
Operator	Meaning
f @ x	f x
img f s	$\{f\,x \mid x \in s\}$
(f . g) x	f (g x)
(f ‖ g) (x,y)	(f x, g y)
f $ t	img [f ‖ f] t
p → f ; g	if (p x) then (f x) else (g x)

Non-primitive Intensional Operators	
Operator	**Definition**
while [p,f]	p → iter [p → f] ; id
f §i	while [$\emptyset \neq \omega$, $(f\,.\,[id \parallel \alpha] \parallel \Omega\,.\,\omega)\,.\,\Delta$] . (i,)
f^n	while [n ≠ α, 1 + ‖ f] . (0,)
restrict(s,f)	img ((id ‖ f) . Δ) s

Table 7.2: RPL intensional operators

7.3.2 Program 2: Employee File Processing

The programs shown in Figure 7.2 process files of employee records. In the two respective programs the employee files are denoted by relations `F` and `f` while the 'hours worked' files are denoted by relations `U` and `u`. The object of the program is to generate a new employee file (`newF`) in which the hours worked (`"H"`) field has been updated for each employee. Relation `sumhrs` calculates the new hours worked and is used to create an updated file (`upd`). The relation override operator (`[;]` in RPL and `[@]` in Drusilla) is used to replace values in the old employee file with values in the updated version to create the new employee file. Once again the programs have the same number of definitions but MacLennan needs two operators to express relation composition — one for extensional relations (`|`) and one for intensional relations (`.`) — whereas Drusilla needs only one (`;`).

7.3.3 Program 3: Gaussian Elimination

The programs shown in Figure 7.3 perform gaussian elimination on a collection of simultaneous equations whose co-efficients are stored in a matrix. The matrix is represented by a vector of vectors. MacLennan's notion of a vector may be viewed as modelling either the mathematical concept of a sequence or an array from an imperative language. The domain of a vector is formed by the element position numbers. In a vector each domain position number relates to the element in that position. In Drusilla, MacLennan's vectors are called sequences. Drusilla sequences are described in section 7.4 and relations over them such as `foldl` are defined.

The gaussian elimination algorithm uses n successive steps where n is the

```
newF = (upd # F) | [;]
upd = (F # U) | (as . ["H",] . sumhrs)
sumhrs = [+] . <[↓"H"] || id>
as <a,b> = {a:b}
F = {124:{"N":"John","R":10,"H":100},
     118:{"N":"Bill","R":15,"H":120},
     207:{"N":"Sally", "R":14, "H":115}}
U = {118:6, 124:40, 207:40}
```

```
newF = f # upd ; [@].
upd = f # u ; sumhrs ; addH.
(x) addH  {("H",x)}.
(_,x) sumhrs  [cont]("H",_) ; [+ x].
f = {(124,{("R",10),("H",100)}),
     (118,{("R",15),("H",120)}),
     (207,{("R",14),("H",115)})}.
u = {(118,6), (124,40), (207,40)}.
```

Figure 7.2: Employee file programs in RPL (upper) and Drusilla (lower)

number of rows in the matrix. Each step performs the transformation

$$(M,k) \mapsto (M',k+1)$$

where M' is obtained from M by performing the elimination process on the kth column:

$$M' = \text{elim (M,k)}$$

The following subsection compares the RPL and Drusilla programs.

7.3.4 Program 4: Finite State Machine Minimisation

The programs shown in Figure 7.4 minimise finite state automata. The algorithm computes state equivalence classes by first deriving the inequivalent states through an iterative method. Initially it is assumed that the final and non-final states are inequivalent (definitions R_0 and `rZero`). Any pair of states which, under the same input, lead to inequivalent states are themselves inequivalent. This is the basis for the iteration, which converges when each state has been compared with every other state. The iteration (performed by *psi* and `iter`) uses polymorphic image (relations $\perp$ and `polyImg`) to find pairs of inequivalent states. The set of equivalent states ($R_=$ and `rEqB`) is the set difference of the set of all states and the set of inequivalent states. The minimised machine ($T_=$ and `tEq`) is derived from the original by replacing each state by its equivalence class.

The gaussian elimination and finite state automata programs are more substantial than the frequency table and employee file programs and are therefore

```
Gauss M = (elim for <1, ... , size M>) M
V = scaprod .  (([1.0 /] .  diag) ₇ (vecdif .  (column ₇ unit)))
elim = matdif . ([↓ 1] ₇ (outerprod . (V ₇ [↓])))
transmap f = [| f] . [#]
vecdif = transmap [-]
scaprod <k,v> = v | [k *]
outerprod <u,v> = u | (scaprod . [,v])
matdif = transmap vecdif
column <M,k> = M | [↓ k]
unit <M,k> = <1,...,size M> | [[= k] -> con 1; con 0]
diag <M,k> = M↓k↓k
con k = λx k
f for S = [@ S] . [f §]
```

```
gauss = id # initSeq ; (elim,_,_)foldl.
v = diag ; [1 /] # (column # unit ; vecdif) ; scaprod.
(_,k) elim  id # ((_,k)v # [cont](k,_) ; outerprod) ; matdif.
(r,s,f) transmap  r # s ; f.
vecdif = (_,_,[-])transmap.
(k,v) scaprod  v ; [* k] ).
(u,v) outerprod  u ; (_,v)scaprod.
matdif = (_,_,vecdif)transmap.
(m,k) column  m ; [cont](k,_).
(m,k) unit  dom m ; {(Unit,0)} @ {(k,1)}).
(_,k) diag  [cont](k,_) ; [cont](k,_).
(m) initSeq  dom m << id.
```

Figure 7.3: Gaussian elimination programs in RPL (upper) and Drusilla (lower)

```
T = {1:{10:10,20:20},2:{10:30,20:30}}
Q = {10,20,30}
F = {30}
Q² = Q x Q
n = size Q²
R₀ = F x (Q \ F)
ψ R = R ∪ ⋃ (rng (T | [⊥ R]))
R∞ = ψⁿ R₀
R= = Q² \ (R∞ ∪ R∞⁻¹)
eclass = [R= elimg]
equiv = [eclass img]
Q= = equiv Q
T= = T | [eclass $]
F= = equiv F
R ⊥ S = R | S | R⁻¹
σ f = (f o (1st ,(ε o 2nd)) , ([\] o (id ,(un o ε)) o 2nd))
f ρ i = 1st o (σ f while ([≠ ∅] o 2nd)) o [i,]
⋃ = [∪] ρ ∅
1st = [↓ 1]
2nd = [↓ 2]
```

```
bigT = {(1,{(10,10), (20,20)}), (2,{(10,30),(20,30)})}.
bigQ = {(10, Unit), (20, Unit), (30, Unit)}.
bigF = {(30, Unit)}.
qSq = bigQ ; inv bigQ.
rZero = bigF ; inv (bigQ \ bigF).
psi = id # (rngPoly ; distUnion) ; [⋁].
(r) rngPoly  bigT ; (_,r)polyImg.
rEq = (psi,rZero,_)iter ; rEqB.
(r) rEqB  qSq \ (r ⋁ inv r).
eclass = (card qSq,_)eclass2.
(_,x) eclass2  rEq ; (_,x)elimg.
equivQ = bigQ << eclass.
tEq = bigT ; (eclass,_)doll.
equivF = bigF << eclass.
(r,s) polyImg  r ; s ; inv r.
(f,s) doll  inv (inv s ; f) ; f.
(f,x,_) iter  [- 1] ; (f,x,_) iter ; f  @ {(0, x)}.
distUnion = ([⋁],{},_)foldr.
(r,x) elimg  r img {(x,Unit)}.
```

Figure 7.4: Automata minimisation programs in RPL (upper) and Drusilla (lower)

more interesting for comparison. Perhaps the most notable difference between the RPL and Drusilla programs is the number of different operators used. The RPL gaussian elimination program uses 11 different operators compared to 6 in the Drusilla program and the RPL finite state automata program uses 16 different operators to Drusilla's 9. In both cases RPL needs almost twice as many operators as Drusilla, even though the same algorithms are used and programs are of a similar size. MacLennan is often forced to introduce two definitions (two symbols) for what is conceptually, at the level of relational abstraction, one operator. One definition is applicable to extensional relations and the other is applicable to intensional relations. Table 7.3 shows examples of such operator duplication. Whenever such an operator is to be used it is left to the RPL programmer to decide which version is required. By contrast Drusilla requires only one symbol for an operator and the system decides which definition is appropriate using typed representation analysis.

Operation	Extensional Symbol	Intensional Symbol
relation application	↓	@
relation composition	|	.
dual composition (construction)	#	;
domain restriction	→	restrict

Table 7.3: Relational Equivalence of RPL operators

7.4 Functional Aspects of Drusilla

Drusilla is certainly not a functional language since it is not based on the $\lambda - calculus$. However, mathematically functions are just a subclass of relations — every function is a relation. In theory, therefore, relational programming should subsume functional programming. It should be possible to express anything in a relational language that can be expressed in a functional language. Similarly it should be possible to evaluate functional language expressions in a relational language. This section explains how functional programming concepts can be expressed in Drusilla.

7.4.1 Function Application

Functional language expressions are built from function applications. Drusilla expressions are also constructed from function applications where the functions are the built-in operators. The operators form a fixed set of combining forms, as advocated by Backus [3] in his functional language FP, except they combine relations not functions.

Application of user-defined relations to arguments is possible at program execution time where relation expressions can be evaluated in the form of Sanderson's [94] *relators*. This is the *map mode* of relation use — given a domain element, a

set of related range elements may be generated in the manner of a set-valued function.

7.4.2 Modelling Lists with Relations

Modern functional languages use tuples, lists and algebraic data types as data structures. Tuples are present in Drusilla but lists and algebraic types are not. Algebraic datatypes are arguably not so important — earlier functional languages such as FP did not incorporate them, but lists are central to all functional languages. In Drusilla relations are the main form of data structure as 'all the world is a relation'. Therefore, if relational programming is to encapsulate functional programming then some form of relational data structure must model lists.

Sequences in RPL

MacLennan uses a form of relation, which he calls a sequence, to emulate lists. He defines a sequence to be a relation with the structure:

$$a_1 \rightarrow a_2 \rightarrow \ldots a_{n-1} \rightarrow a_{n-2}$$

If `x` and `y` are two objects then the pair `(x,y)` can be an element of an extensional relation and used to relate x to y. MacLennan uses this to represent the relation of succession in a sequence:

$$x \rightarrow y$$

means y is the successor of x in the sequence. Thus a list of two or more elements:

$$< x_1, x_2, \ldots, x_{n-1}, x_n >$$

can be represented by the relation:

$$\{(x_1, x_2), (x_2, x_3), \ \ldots \ , (x_{n-1}, x_n)\}$$

For example, the list `[10,20,30,40]` would be represented by the relation:

```
{(10,20),(20,30),(30,40)}.
```

MacLennan defines several operators for constructing and decomposing such sequences and for accessing their elements. The α and ω operators, defined in Table 7.1, return, respectively, the first and last elements of a sequence.

MacLennan recognises that only sequences with two or more elements can be represented. This creates a problem for applications where it is necessary to manipulate an empty or singleton sequence. Another problem is that the same relation may be used to model several different sequences without it being clear exactly which one is intended. For example the two sequences:

$$2 \rightarrow 4 \rightarrow 2$$
$$4 \rightarrow 2 \rightarrow 4$$

will both be represented by the relation:

```
{(2,4),(4,2)}
```

This creates ambiguity about the exact sequence being represented. Similarly a sequence consisting of the same element repeated several times, for example:

$$1 \rightarrow 1 \rightarrow 1$$

would need to be represented by the relation:

```
{(1,1),(1,1)}
```

This is clearly not a set and so violates the concept of a relation as a set of pairs.

Sequences in Drusilla

The problems with MacLennan's sequences suggest that another form of relation should be used to model lists. Reade [85] observes that lists are derived from the mathematical concept of a *sequence*. Suppes [109] defines a sequence mathematically as a function on the set of natural numbers. If x is a sequence then x_n is the *n-th member* or *term* of x. Such a sequence may be modelled in Drusilla by a relation between the first n natural numbers and the elements of the sequence. 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)}
```

This form of relation is one method for emulating lists in Drusilla. Such sequences maintain the notion that 'all the world is a relation' and may be manipulated by the relational operators. The notion of an empty list is replaced by the empty relation and there is no ambiguity created by such sequence representations of lists.

Operations Over Sequences

The standard functional language list manipulating operations can all be defined for sequences in Drusilla. Example sequence processing operations are given in figure 7.5. 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 which results from appending them and a sequence is related to the number of elements it contains by `length`. Higher-order operations such as `filter`, `foldl`, `foldr` and `map` can also be defined. All definitions are polymorphic as in a functional language. These definitions combined with the `map` mode of relation use permit Drusilla to be used for evaluation of typical functional language expressions.

The operations exhibit similar structure, which we explain by deriving the definition of `map` from the standard functional language definition shown in Figure 7.6. In Drusilla `map` is a relation between a pair `(f,s)` and a relation `t`.

```
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) ; (p,_)condCons)
(p,_) condCons  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).
```

Figure 7.5: Standard list operations defined for Drusilla sequences

```
map f [] = []
map f (h:t) = f h : map f t
```

Figure 7.6: Definition of `map` in a functional language

Relations s and t are sequences and their corresponding elements are related under relation f.

The first equation defines the base case for the recursive definition. In Drusilla this should state that the empty sequence relates to the empty sequence:

```
{({},{})}
```

The second equation uses pattern matching to split the list into its head and tail. The function f is then applied to the head. This can be replaced in Drusilla by a relation between a sequence s and f (hd s):

```
hd ; f
```

The function f must be mapped over the tail of the list. This can be replaced in Drusilla by a relation between a sequence s and map f (tl s):

```
tl ; (f,_)map
```

These two relations are combined by the dual composition operator so that they relate sequence s to the pair (f (hd s), map f (tl s)):

```
hd ; f # (tl ; (f,_)map)
```

The final operation must construct the new sequence from the new head and tail. To do this we compose the relation with cons:

```
hd ; f # (tl ; (f,_)map) ; cons
```

Map must be defined for both non-empty and empty sequences though. Since these are mutually exclusive we can simply take the union (\/) of the two defining relations and complete the definition of map:

```
{({},{})} \/ (hd ; f # (tl ; (f,_)map) ; cons)
```

7.4.3 Relational Laziness

Lazy evaluation [85] ensures that expressions or components of structures are expanded in a demand driven way and are not evaluated more than is necessary to provide a value at the top level. Lazy evaluation delays evaluation of arguments to functions until they are needed by evaluating expressions at the outermost level possible at each step. For example, in evaluating f x, f is first evaluated until a function is obtained. The application of this function to argument x is then evaluated in such a way that x is evaluated only when it needs to be (and as little as possible) to determine a final answer.

Lazy evaluation centres around function application. All Drusilla expressions that are not basic values are constructed by applying the built-in relational operators to simpler expressions. As these operators are functional their manipulation of relations is a form of function application. Lazy evaluation in Drusilla is the same as in normal functional languages. An operator is evaluated before the relation(s) it manipulates and those relations are only evaluated as much as the

operator requires. This is called *relational laziness*. At the implementation level, Drusilla inherits laziness from Miranda.

Two benefits normally associated with laziness are manipulation of infinite data structures and non-strict definitions. Chapter 3 gave examples of such definitions. Non-strict definitions are as natural in the relational framework as in the functional framework. A relation is non-strict if it relates a given domain value to a range value even when (part of) the domain value is undefined.

The definitions of `nats`, `odds` and `squares` in Figure 3.17 all illustrate that infinite relational data structures can be lazily evaluated. However there is a problem with recursion over such structures. For example, the sieve of Eratosthenes program presented in Figure 7.7, can only be used for sieving the prime numbers from some initial subsequence of the natural numbers. If the infinite sequence is sieved, a non-terminating computation is entered and the outermost reduction never provides any value at the top level.

```
sieveInit = (2,_)seqRange ; sieve.
sieve = [({},{})] \/ (hd # tl2 ; sieveB).
(x,_) sieveB ((_,x)hasntFactor,_)filter ; sieve ; (x,_)cons.
hasntFactor = neg ([mod] ; {(0,Unit)}).
seqRange = (1,_,_)seqRangeB.
(i,_,y) seqRangeB (i,_)mkrel # ([+ 1] ; (i+1,_,y)seqRangeB)
                    ; [\/] @ {(y,{(i,y)})}.
(x,y) mkrel {(x,y)}.
```

Figure 7.7: The sieve of Eratosthenes written in Drusilla

Generally any relation composition

```
r ; s
```

represented by a set-valued function, denotes a Miranda expression in the implementation:

```
concat . map g . f
```

Set-valued function `f` represents `r` and set-valued function `g` represents `s`.

Laziness is lost because the system uses the conventional definition of `map`, shown in Figure 7.6, which is insufficiently lazy as it evaluates the argument list to weak head normal form through pattern matching. A lazier definition of `map` ensures recursion over infinite lists terminates by using explicit projection operators (`hd` and `tl`).

```
map f xs = f (hd xs) : map f (tl xs)
```

This definition does not evaluate the argument list to weak head normal form until the projection operations are applied. However it cannot be applied recursively to a finite list as it is only defined for non-empty lists. It could be used in the

implementation if accompanied by static analysis to detect infinite lists, although not all infinite lists can be detected since this is equivalent to the halting problem.

If the system knows r is functional, then a *point-to-point function* representation, f', can be used and the expression `r ; s` translated into:

$$g \cdot f'$$

This solves the problem that was being created by the insufficiently lazy definition of `map`. However, this technique is not used since functional relations cannot be detected by the Drusilla system.

7.5 Relational Aspects of Drusilla

7.5.1 Extensional Mathematical Operators

The basic mathematical relations of Drusilla may be defined extensionally as shown in Figure 7.8. These definitions, written in Miranda, are based on a diagonalised cartesian product of the natural numbers `natPairs` which exploits laziness.

```
nats = [0 ..]
natPairs = [(x,y) // x <- nats; y <- nats]
lessAl = [(x,y) | (x,y) <- natPairs; x < y]
greatAl = [(x,y) | (x,y) <- natPairs; x > y]
plusAl = [((x,y),x+y) | (x,y) <- natPairs]
minusAl = [((x,y),x-y) | (x,y) <- natPairs]
multAl = [((x,y),x*y) | (x,y) <- natPairs]
divAl = [((x,y),x/y) | (x,y) <- natPairs; y ~= 0]
modAl = [((x,y),x mod y) | (x,y) <- natPairs; y ~= 0]
sqrtAl = concat (map bothRoots [1 ..])

bothRoots x = [(x,y),(x,-y)]
              where
              y = sqrt x
```

Figure 7.8: Extensional representations of mathematical relations

The possibility of using extensional representations is of interest for relational programming as it would permit many more expressions to be used in show mode. For example, the set of positive integers could be explicitly enumerated by evaluating the expression `[> 0]`. While this is useful it does pose the problem of conflicting operator definitions. One fundamental premise of the Drusilla system is *representation independence* — each possible representation for a given relation should denote exactly the same conceptual relation and, therefore, relate the same values. For example, each representation of arithmetic plus should satisfy the calculus definition

$$(x, y)\,[+]\,z \Leftrightarrow z = x + y$$

for all values of x, y and z. The characteristic and set-valued function representations of plus relate real numbers but the extensional representation can only relate the natural numbers because the real numbers are not recursively enumerable. To maintain representation independence, extensional representations of operators are not permitted in Drusilla.

7.5.2 Describing Sets of Values

Drusilla is unconventional in its treatment of sets. In Prolog, sets can be represented by lists or by predicates. Functional languages typically use either lists or predicates (Boolean-valued functions) to describe sets of values. For example, if set S is denoted by predicate p then, for any element x:

$$\mathrm{p\ x} = \mathrm{True} \Leftrightarrow \mathrm{x} \in \mathrm{S}$$
$$\mathrm{p\ x} = \mathrm{False} \Leftrightarrow \mathrm{x} \notin \mathrm{S}$$

Such predicates are used in operations such as list filtering. Thus a set of integers, {5,6,7,8}, named p could be represented in a functional language by either of the definitions:

```
p = [5,6,7,8]
p x = x > 4 & x < 9
```

However, these are two separate types of object — the first is a list and the second a function — which therefore cannot be used interchangeably.

In contrast Drusilla denotes a set by a relation between the set elements and `Unit`. If relation `r` denotes set S then:

$$\mathrm{x\ r\ Unit} \Leftrightarrow \mathrm{x} \in \mathrm{s}$$
$$\neg\ \mathrm{x\ r\ Unit} \Leftrightarrow \mathrm{x} \notin \mathrm{s}$$

Consequently Boolean values are not needed in Drusilla. The set {5,6,7,8} could be denoted by either of the relations:

```
r = {(5,Unit),(6,Unit),(7,Unit),(8,Unit)}
r = [> 4] /\ [< 9]
```

These two relations both denote the same set of values and both could, for example, be used to filter a sequence. This lessens the gap between program and data. The only difference between them is in their possible modes of use — the first can be used in show, forward or test modes while the second can only be used in forward or test modes. Any relation that may be used in show mode may be perceived as data.

7.5.3 Recursive Decomposition of Relations

Functional and logic languages naturally admit a recursive style of programming since they mainly use recursively defined datatypes, which are created by constructor functions, and can be recursively decomposed naturally. By contrast, the relational data structures used in a relational language are *not* recursively defined and are *not* created by constructor functions, they are just values (sets of pairs). This makes recursion over them problematic as there is no natural way of recursively decomposing them.

This problem is partially circumvented in the definition of sequences because construction (`cons`) and destruction (`hd`, `tl`) operations are defined. However problems are still present. For example, the `hd` relation (`hd = [cont](1,_)`) relates a sequence `s` to any value `x` such that `1 s x` If there is more than one such element then the sequence has more than one head. Moreover, if the sequence is infinite and `hd` is used as a relator then evaluation never terminates since there is always the possibility of another head value further on in the sequence.

The decomposition problem can also be avoided using relation `decomp`, defined in Figure 7.9. Relation `decomp` holds between a relation `r` and a pair `((x,y),s)` where `(x,y)` is an element of `r` and `s` is the relation remaining after `(x,y)` is removed from `r`. Although `decomp` provides recursive decomposition of relational data structures it is not functional because element extraction is non-deterministic. Consequently it introduces non-determinism into any relation that references it. This non-determinism must be handled at the implementation level as a set of alternatives. For example, consider the quick sort program shown in Figure 7.10. At execution time, when `qsort` is applied as a relator to a relational data structure whose range type is *num* the result is several, identical, sorted sequences. This form of recursion incurs a high run-time overhead — `qsort` must be evaluated for each possible recursive ordering of elements. This is a good argument for introducing recursively constructed data structures into relational programming although this would compromise the notion that *'all the world is a relation'*. The problem of handling recursion in relational programming is an important topic for future research.

```
decomp = id # [cont] ; elRem.
(r,(x,y)) elRem ((x,y), r \ {(x,y)}).
```

Figure 7.9: Decomposition of a relational data structure

7.5.4 Using Relations to Handle Non-determinism

One main criticism of functional languages is their poor handling of non-determinism. This problem arises directly from the underlying mathematical model — functions can only describe deterministic (functional) relationships. In contrast logic

```
qsort = decomp ; qsortB @ {({},{})}.
((i,x),_) qsortB  (_,[< x])part ; (qsort ; (x,_)post || qsort)
                                ; join.
(s,p) part (s >> p,s -> p).
(s,t) join   s ⋁ inv (inv t ; [+ (card s)]).
(x,s) post  s ⋁ {(card s + 1,x)}.
```

Figure 7.10: Quick sort based on relation decomposition

languages are suited to programming non-deterministic solutions to problems because they are based on predicates (named n-ary relations) and backtracking — an implementation mechanism that allows those predicates to return different alternative results. Relations, by their nature, may denote non-functional relationships and so have potential to describe non-deterministic (search based) computations. This subsection demonstrates that Drusilla naturally permits non-deterministic solutions to problems. Examples of such solutions are given. Various constructs suggested as non-deterministic extensions to functional languages are defined in Drusilla without extensions.

Modelling Non-determinism with Lazy Lists

A non-deterministic computation is one that has alternative possible results. A functional program might represent this by lazily generating a list of results as described by Wadler [116]. For example, Figure 7.11 presents typical solutions to generating all the permutations of a list and non-deterministically inserting an item into an arbitrary position within a list. Both solutions generate a list of alternative results — `perms` returns a list of all possible permutations and `insert` a list of all possible insertions.

The Drusilla solution, shown in Figure 7.11, allows permutations to be viewed as a relation between two sequences. Similarly insert is a relation between a pair `(x,s)` and a relation `t` where `t` is a sequence generated by non-deterministically inserting element `x` into sequence `s`. This permits reasoning about the permutation and insert relationships and does not require 'mapping' over lists. At the implementation level the Drusilla interpreter exploits the functional language 'lazy list' approach advocated by Wadler [116] as the mechanism for handling non-determinism. The relational abstraction permits this to be hidden from the programmer.

Adding Non-determinism to Functional Languages

Many non-deterministic constructs have been suggested for functional languages but most interfere with reasoning, often to the extent of compromising referential transparency. Hughes and O'Donnell [48] give a good review of the various approaches for inclusion of non-determinism and suggest what is perhaps the

```
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 7.11: Permutations solutions in Miranda (upper) and Drusilla (lower)

best solution to date — addition of a set data type. Computations that may yield alternative results are regarded as set-valued functions. These functions when applied to arguments, yield a set of alternative possible results, which can be gathered using set-theoretic operations such as *union*. Their ultimate goal is to give a semantics to a language in which each set is represented by one of its values chosen non-deterministically. Hughes and O'Donnell demonstrate how non-deterministic primitives may be implemented using this data type in a referentially transparent way.

McCarthy [73] suggested the amb operator:

amb x y = either x or y

Intuitively amb evaluates its two arguments in parallel and returns the first to yield a value. This makes amb 'angelic' or 'bottom avoiding':

$$amb\ x\ \bot\ =\ x$$
$$amb\ \bot\ y\ =\ y$$

Hughes shows that by using sets this may be defined:

$$amb\ ::\ *\ \rightarrow\ *\ \rightarrow\ \{*\}$$
$$amb\ x\ y\ =\ \{x\}\ \cup\ \{y\}$$

The angelic property is preserved because this union operator is angelic — the representative element of the set amb x y is the first of x or y to return a result.

Henderson [40] advocates a non-deterministic merge, which interleaves two streams in the order in which their elements become available. This is defined by Hughes and O'Donnell as in Figure 7.12. They use an operator for mapping a function over a set (*) defined:

$$f\ *\ S\ =\ \{f\ x\ \mid\ x\ \in\ S\}$$

Non-determinism can be handled in Drusilla because relations are equivalent to set-valued functions — relations are combined directly using relation-theoretic operators. For example the amb function can be written as a relation constructed from the union of two relations — one to choose the first argument, the other the second.

```
amb = fst \/ snd
```

A Drusilla translation of Hughes solution to the non-deterministic merge is also shown in Figure 7.12.

```
merge xs ys = bias xs ys \/ bias ys xs
bias [] ys = {ys}
bias (x:xs) ys = (x :) * merge xs ys
```

```
merge = bias \/ (swap ; bias).
(_,ys) bias  {({},ys)} \/ (hd # (tl ; (_,ys)merge) ; cons).
(a,b) swap (b,a).
```

Figure 7.12: Non-deterministic merge by Hughes (upper) and in Drusilla (lower)

Hughes observes that with his approach referential transparency is only compromised if a choose function, which can non-deterministically choose any element from a set, is introduced. This function has the type:

$$choose :: \{*\} \rightarrow *$$

However, it is not choose itself which introduces the side effect but its application to an argument. If the function is replaced by a relation and never applied to an argument then it may be reasoned about and composed with other relations without any compromise to referential transparency. The choose relation may be defined in Drusilla:

$$choose :: (A \leftrightarrow un) \leftrightarrow A$$

```
choose = [cont](_,Unit).
```

Although they are mathematically the same relations have an advantage over set-valued functions — they may be combined directly. Hughes and O'Donnell need to define two extra operators for the composition of two functions that return sets of results. The first operator (*) defined above takes the image of a set under a function. The second (⋃) is a distributed union operator defined for 'flattening' a set of sets of results and has the type:

$$\bigcup :: \{\{*\}\} \rightarrow \{*\}$$

For example, the composition h of two set-valued functions f and g can be defined:

$$f,\ g,\ h :: int \rightarrow \{int\}$$
$$h\ x = \bigcup (f * g\ x)$$

Relations need no extra operations over sets and the composition can be defined directly:

```
h = g ; f.
```

The ability to define such relations directly and the facility for higher-order, polymorphic definitions should permit non-deterministic programs to be more concise than in a functional or logic language. For example the `perm` relation could be defined more concisely as the fold right of a non-deterministic insertion by exploiting the `foldr` relation over sequences defined in Figure 7.5:

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

The main disadvantage of Drusilla is that the relational operators do not preserve the angelic property, for example, `amb` is no longer bottom avoiding. This is a major aim of Hughes and O'Donnell's work — they are interested in generating one solution not all solutions. However, in the implementation of Drusilla set-valued functions are processed in exactly the same manner. Consequently, Hughes and O'Donnell's language with its set data type and set-valued functions could be used to implement Drusilla in which case its angelic properties would be inherited by Drusilla allowing 'bottom avoiding' relations to be defined. Kowlaski [60] refers to this form of non-determinism as 'Don't care' — the program executor 'doesn't care' which solution is generated. Kowalski also identifies another form of non-determinism called 'Don't know' — the program executor 'doesn't know' which solution is required until it is generated. 'Don't know' non-determinism encapsulates the notion of searching and state space handling. This form of non-determinism is important for logic programming and Drusilla's handling of it is discussed in section 7.6.

7.5.5 Relational Flow of Control

A programming language must include primitives or constructs for allowing the programmer to influence the flow of control in a program. Conventional imperative languages typically use an 'if ... then ... else' statement. The functional language equivalent of this is guards and the standard `if` function defined:

```
if True x y = x
if False x y = y
```

Application of this function is sometimes denoted by a special syntactic form:

```
if ...then ...else
```

Modern functional languages also exploit pattern matching in function definitions through alternative Kleene recursion equations. When such a function is applied to an argument the equation chosen is the first one (in the top down textual ordering) whose left hand side pattern matches with the argument. Control is influenced by patterns and the textual ordering of equations. Similarly in Prolog one procedure may be defined by several Horn clause rules. Prolog satisfies a goal by matching it against the rules within a procedure in a top-down manner until one rule is selected and then satisfies goals within that rule from left to right. Once again control can be influenced by careful textual ordering.

In such `if` constructs the result of a conditional test determines the flow of control. This seems inappropriate for the relational algebra style of programming advocated in Drusilla; conditions at the relational level would appear more desirable. Pattern matching is considered unnecessary for expressiveness and the textual ordering it brings with it is considered undesirable, since one aim of declarative programming is to remove the prescriptive element where textual ordering of program elements has semantic importance.

In Drusilla an alternative approach is taken to influence the flow of control. When a relation is defined it may be partitioned into several subsidiary relations defined separately. The main relation can be constructed by using the union or ordered union (relation override) operators to 'glue' the subsidiary relations together. The override operator is applicable when subsidiary relations with disjoint domains cannot easily be identified since it allows one relation to be given precedence over another. For example, a relation over sequences may be constructed from two relations — one whose domain is the empty sequence (relation), and one whose domain is any non-empty sequence of the domain type The insertion sort relation *insort* in Figure 7.13, for example, uses this technique, as do the sequence manipulation operators presented in Figure 7.5. This preserves the expressive power of textual ordering in a mathematically tractable way — conditions are introduced in the form of expressions.

```
insort = [({},{})] \/ (hd # (tl ; insort) ; insert).
(x,_) insert  (x,_)cons @ (hd # tl ; (x,_,_)insert2).
(_,h,ts) insert2  [h <] << (_,ts)insert ; (h,_)cons.
```

Figure 7.13: A Drusilla insertion sort program

7.5.6 Relational Exception Handling

Exception Handling in Functional Programming

The problem of exception handling arises through function application. An expression of the form `f x` denotes a new value formed from application of function `f` to domain value `x`. This generates an exception condition if `f` is a partial function and the value `x` is outside its domain. Spivey [105] describes a theory of how to handle exceptions in a functional language by making such partial functions total through use of an algebraic data type. Here an error case is defined for function f which returns some result denoting failure if $x \notin dom\ f$.

Wadler [116] defines an alternative mechanism for handling exceptions by introducing functions that return a list of result values, not one value. This handles non-determinism by allowing a function to return more than one value in the list. Also an exception can be raised by a function returning the empty list, which denotes the absence of any result.

While both of these mechanisms are effective, they do complicate code to some extent. For example, with Wadler's technique two functions cannot be composed directly, instead one must be mapped over the result of another:

```
f . g
```

is replaced by

```
concat . map f . g
```

Exception Handling in Relational Programming

Sanderson [94] observes that when a relator is applied to an operand there is no expectation as to the number of results. Any number or none at all may be produced, so the notation takes account of an application being undefined without having to make special provision for it.

The relational and mathematical operators can be applied to relations to construct expressions because they are known to be total functions. When applied to an argument they produce exactly one, defined result. Functions such as division (/) and modulo (mod) are known to be partial and cannot be used to form compound relation designations.

At execution time when a relation is applied to an argument value in forward mode it returns a set of results. If the set contains one result then the relation is acting as a function. If the set contains several results then non-determinism has been handled. If the set is empty then an exception has been handled. This corresponds to Wadler's work, and indeed his mechanism is used at the implementation level. However it seems more intuitive in the relational framework where the lists of results can be hidden behind relational abstraction.

7.5.7 Deriving Programs from the Calculus

Drusilla programs can be derived informally from specifications expressed in the relational calculus. This is desirable for several reasons:

- It is often easier to express the solution to a problem in the calculus and then derive a Drusilla program than to write the program directly.
- For the translation of Prolog programs to Drusilla it is easier to translate from predicate calculus to relational calculus and then to relational algebra than it is to translate directly to relational algebra.
- When deriving a program from a formal specification, it may be more natural to express the specification in the calculus than in Drusilla. If a program is derived formally from a formal specification then there is no need to prove its correctness.

The technique for deriving programs is, in principle, simple. The Drusilla relational operators are all defined in the relational calculus as shown in Table 3.2.

Therefore any calculus expression that matches the definition of an operator may be replaced by that operator. A Drusilla relation can be obtained by repeatedly substituting operators for subexpressions in a calculus specification

This work relates conceptually to the fold/unfold transformation system of Burstall and Darlington [17, 26]. This system is based mainly on two operations: *fold* and *unfold*. An unfold operation replaces a function call by the corresponding body instance. The fold operation is the reverse of unfold — it replaces a body instance by a corresponding function call. If an expression matches the body of some function definition then that expression may be replaced by a call to that function. This fold operation takes place when a Drusilla program is derived from a calculus expression. Operators are substituted for expressions that match their calculus definitions. For example, a sequence membership relation is derived in Figure 7.14. Calculus expressions appear in Roman font and Drusilla expressions appear in `courier`

$$
\begin{aligned}
&\text{xs member y} \\
&\quad \Leftrightarrow \text{xs hd y} \vee (\text{xs tl zs} \wedge \text{zs member y}) \\
&\quad \Leftrightarrow \text{xs hd y} \vee \texttt{xs (tl ; member) y} \\
&\quad \Leftrightarrow \texttt{xs (hd \textbackslash/ (tl ; member) y} \\
&\quad \Rightarrow \texttt{member = hd \textbackslash/ (tl ; member)}
\end{aligned}
$$

Figure 7.14: Derivation of a sequence membership relation

As a slightly more complex example relation `nfib` is derived from the definition of the `nfib` function shown in Figure 7.15.

```
nfib 0 = 1
nfib 1 = 1
nfib n = nfib (n-1) + nfib (n-2) + 1
```

Figure 7.15: Definition of `nfib` function

The base cases of the recursive definition for $n = 0$ and $n = 1$ can be mapped directly into a calculus relation:

```
{(0,1),(1,1)}
```

This is also a Drusilla expression and hence no derivation is needed.

The recursive part of the function can be expressed in the calculus and a Drusilla expression derived as shown in Figure 7.16. (Drusilla expressions appear `courier` font.)

Finally the relation denoting the base case overrides the derived expression to give the complete definition of `nfib`:

```
nfib = [-1] ; nfib # ([-
2] ; nfib) ; [+] ; [+ 1] @ {(0,1),(1,1)}
```

```
n nfib x  ⇔ (n − 1) nfib y ∧ (n − 2) nfib z ∧ (y,z) [+] w ∧ (w,1) [+] x
          ⇔ (n,1) [-] m ∧ m nfib y ∧ (n,2) [-] k ∧ k nfib z ∧
            (y,z) [+] w ∧ (w,1) [+] x
          ⇔ n ([-1] ; nfib) y ∧ n ([-2] ; nfib) z ∧ (y,z) ([+] ; [+1]) x
          ⇔ n ([-1] ; nfib # ([-2] ; nfib)) (y,z) ∧ (y,z) ([+] ; [+1]) x
          ⇔ n ([-1] ; nfib # ([-2] ; nfib) ; [+] ; [+1]) x


          ⇒ nfib = [-1] ; nfib # ([-2] ; nfib) ; [+] ; [+ 1]
```

Figure 7.16: Derivation of `nfib` relation

7.6 Logical aspects of Drusilla

This section considers the extent to which Drusilla is a logic programming language.

7.6.1 What is Logic Programming?

Logic programming is more difficult to define precisely than functional programming as a spectrum of definitions exists. At the most general extreme of the spectrum any programming language based on a formal logical system is included — a fact observed independently by Malachi [70] and Goguen [38]. This of course includes relational programming along with pure functional and equational programming. The most narrow extreme is confined to the procedural interpretation of the Horn Clause subset of first-order predicate calculus as advocated by Kowalski [58] and Van Emden [61]. This definition is perhaps the one most commonly adhered to although in reality it includes little other than Prolog.

Drusilla has been developed out of the desire to merge aspects of functional and logic programming into a single language. For this purpose these definitions are too extreme — the first so general it includes functional programming and the second so narrow it offers no scope for inclusion of functional features. A less extreme definition that better describes the essence of logic programming is required. Hogger [42] gives such a definition:

> *'A logic program consists of sentences expressing knowledge relevant to the problem that the program is intended to solve. The formulation of this knowledge makes use of two basic concepts: the existence of discrete objects, referred to here as individuals; and the existence of relations between them. ...*
>
> *... Reasoning about some problem posed on the domain can be achieved by manipulating these sentences using logical inference. In a typical logic programming environment the programmer invents the sentences forming his program and the computer then performs the necessary inference to solve the problem.'*

Predicate Formula	Drusilla Relation
P(x)	p
Q(x)	q
P(x) ∧ Q(x)	p /\ q
P(x) ∨ Q(x)	p \/ q
¬ P(x)	neg p
P(x) ⇒ Q(x)	neg p \/ q

Table 7.4: Predicate calculus formulae and corresponding Drusilla relations

7.6.2 Relational Logic Programming

The above definition of logic programming bears resemblance to relational programming. A relational program is a collection of relation definitions each of which expresses a relationship between two sets of objects — its domain and range. Each definition also forms a sentence expressing knowledge relevant to the problem the program is intended to solve. 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*, rather than a calculus, and new relationships are constructed from existing ones using *algebraic operations*. The presence of operations for direct construction of new relationships means that object names are used less in construction of expressions. Arguably this also clarifies the logic of relationships in a program.

Gray [39] observes the correspondence between predicates and the sets of values that form the extensions of those predicates. This relationship also exists between predicates and Drusilla relations as shown in Table 7.4.

No rule of inference is present in Drusilla but this may be regarded as an implementation mechanism, not a conceptual feature. The conceptual feature is the ability to express relationships between objects at a high level while relying on the system to assimilate them. The inference rule is just one assimilation technique for establishing whether relationships hold; reduction as used by Drusilla is another.

There appears to be a link between relational programming and intuitionistic logic. The essence of *intuitionistic logic* (or *constructive logic*) is that it is not truth functional but *proof functional* — a proposition is true *if and only if it has a proof.* To assert the existence of an object it is necessary to show how to find it. The meaning of a compound proposition is explicated by showing how a proof may be given as a function of the proofs of its constituents. A proposition in logic programming may be thought of as being the question of whether some relationship exists between entities. Prolog proves a proposition by using backwards inference to show the negation of the proposition is inconsistent with existing relations. In Drusilla, a similar process is entered at the implementation level, but at the conceptual level the programmer constructs the proposition from existing relations. The relational operators provide the functions for proof construction.

This is illustrated in Figure 7.17 where a Prolog program for inferring ancestral relationships in the Julio-Claudian (Roman Cæsar) family tree is given along with a Drusilla translation for comparison.

7.6.3 More Structured Control

Hogger and Kowalski [60, 59] both advocate the notion of a logic program algorithm being composed of separate logic and control elements. Kowalski expresses this as:

$$algorithm = logic + control$$

The declarative and operational properties of statements can be analysed separately as discussed in section 7.2. In a logic program the logic component is the set of statements making up the program. The control component is the execution strategy and is largely fixed in the interpreter. The strategy used by Prolog, and the one Hogger regards as standard, is top-down, left to right satisfaction of goals. The programmer can influence control of execution by choosing the textual ordering of calls and procedures and by using special devices such as the non-logical 'cut' operator.

The separation of logic and control in Drusilla is less explicit. The logic of an expression is given by the operators and relations it uses. The execution strategy applied to Drusilla programs, relational laziness, dictates that the order of evaluation for relations in an expression is determined by the operators used to construct that expression. Control can be influenced both by the programmer and by the implementation. The programmer controls flow logically through expression construction and choice of operators, as explained in section 7.5.5. Textual ordering is unimportant and no non-logical features are required. The implementation influences control through symbolic manipulation using known laws of relation equivalences to transform expressions. An expression when transformed has different operator structure and hence different flow of control.

Although Drusilla contains logic and control features, they are not so clearly separated as in a relational calculus logic language. Bellia and Levi [8] state that one reason for wanting to integrate functional and logic languages is to add the kind of control information present in functional programming to logic programming — the rigorous division between control and logic is not necessarily desirable. Hogger observes that the programmer needs to be aware of the control strategy for the sake of efficiency. However the textual ordering of procedures and calls does not just give control information — it simplifies program reading by giving it structure as well. Imagine reading a text if it was just a series of facts written down at random with no structure. It would be extremely hard to comprehend although its content, in terms of logic, would be unchanged. Structure helps convey meaning hence texts are broken down into sections, paragraphs and sentences. The use of operators to combine expressions introduces structure as well as control in a logical fashion.

It is interesting to relate the approach of Drusilla to a quote from Hogger concerning the future of control flow in logic programming.

```
parent(drusus, germanicus).
parent(drusus, claudius).
parent(antonia, germanicus).
parent(antonia, claudius).
parent(germanicus, caligula).
parent(germanicus, drusilla).
parent(agrippina, caligula).
parent(agrippina, drusilla).
parent(caligula, juliaDrusilla).
parent(caesonia, juliaDrusilla).
male(germanicus).
male(drusus).
male(claudius).
male(caligula).
female(X) :- male(X), ! , fail.
female(X).
father(X,Y) :- male(X), parent(X,Y).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Y) :- father(X,Z), parent(Z,Y).
parentCouple(X,Y) :- parent(X,Z), parent(Y,Z), X \== Y.
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \== Y.
brother(X,Y) :- sibling(X,Y), male(X).
sister(X,Y) :- sibling(X,Y), female(X).
uncle(X,Y) :- brother(X,Z), parent(Z,Y).
uncle(X,Y) :- parentCouple(X,Z), sister(Z,W), parent(W,Y).
```

```
parent = {("Drusus","Germanicus"), ("Antonia","Germanicus"),
  ("Drusus","Claudius"), ("Antonia","Claudius"),
  ("Germanicus","Caligula"), ("Agrippina","Caligula"),
  ("Germanicus","Drusilla"), ("Agrippina","Drusilla"),
  ("Caligula","Julia Drusilla"), ("Caesonia","Julia 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.
brother = male << sibling.
sister = female << sibling.
uncle = (brother ; parent) \/ (parentCouple ; sister ; parent).
```

Figure 7.17: Ancestors programs in Prolog (upper) and Drusilla (lower)

'Effective logic programming in the present state of the art, therefore relies mostly upon choosing appropriate logic components to suit the limited control available. One day it might be possible to rely upon one's implementation to devise the most effective control for whatever program is input, thus placing the burden of intelligence upon the machine rather than upon the programmer.'

7.6.4 Relation Level Negation is Logical

The unimportance of textual ordering in Drusilla is desirable for logic. For example the definition of mother in the Prolog program in Figure 7.17 could have been written

```
mother( X,Y ) :- female( X ), parent( X,Y ).
```

without changing its logical content or declarative reading. However its operational content would be drastically transformed. The original definition of `mother` uses `parent` to generate parent names and any parent who cannot be proved to be male (using the `male` unit clauses) is presumed female under the closed world assumption. This is the classic generate and test problem solving paradigm described by Winston [120]. In the new definition of `mother` the textual ordering of the generator `parent` and the test `female` is switched. The `female` predicate cannot generate names since it is defined using negation-as-failure. Consequently this rule attempts to prove a parent is female before that parent is generated and hence alway fails!

By contrast there is only one logical way to define the mother relation in Drusilla — a `mother` is the `parent` relation with its domain restricted to `female`. The `female` relation can be defined using the relation complement operator (`neg`) to obtain the negation of the `male` relation directly. Constraints on the use of `female` still exist: for example, it cannot be used to generate female names. However, these constraints are reported to the programmer as modes — `female` can only be used in test mode (yet `mother` can be used in show, forward or test modes). This is perfectly logical and requires no negation-as-failure or extra-logical 'cut' operator. Therefore, Drusilla succeeds as a logic language where Prolog fails!

7.6.5 Non-determinism and Search Based Computation

Another main feature of logic languages is handling of non-determinism and search based computation. Indeed, Bellia et al [7] regard the central features of logic programming to be 'don't know' non-determinism [60] (search based computation) and logical variables for unification.

Section 7.5.4 describes how Drusilla naturally handles non-determinism in a tractable manner. 'Don't know' non-determinism is not handled by a mechanism such as backtracking, but by lazily evaluating lists of alternative results [116]. To illustrate Drusilla handling a problem with a search space, a solution to the 8-queens problem is presented in Figure 7.18.

```
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 7.18: Drusilla solution to the eight queens problem

When this program is given to the Drusilla interpreter it can be used to enumerate all 8-queens solutions using a query of the form

```
(queens) 8
```

or to test whether a particular board position is an 8-queens solution using a query of the form

```
(8)(queens)({(1,4),(2,2),(3,5),(4,8),(5,6),(6,1),(7,3),(8,7)})
```

At the implementation level the Drusilla interpreter performs this test by lazily enumerating all the solutions in a list and testing whether the given board position is in this list. Although this is obviously inefficient it means that the Drusilla program is polymodal in the sense that a Prolog solution might be. However a Prolog solution can perform such a test in constant time by exploiting partially instantiated data structures. Such structures contain logical variables that are initially uninstantiated but bound to values by unification as the program executes.

Description of Eight Queens Program

The relation `queens` holds between a number n and any chess board position that contains n queens such that no queen threatens any other. Each solution s is a relation between rows and columns: each element $r\ s\ c$ indicates there is a queen in row r at column c. The base case for the recursion denotes the empty board solution for putting 0 queens on the board:

```
{(0,{})}
```

To place n queens on the board one must first position $(n-1)$ queens. The computation is simplified if they are placed in the first $(n-1)$ rows. This can be expressed recursively as a relation between n and $(n-1)$ *queens*:

```
[-1] ; queens
```

Let newQueens be a relation between n and a position pair (n, x), which places an nth queen on the board at row n in some column x, $1 \leq x \leq 8$. Relation newQueens is composed with `[-1] ; queens` so that n is related to a pair $(board, pos)$ where $board$ is a relation denoting positions of $(n-1)$ queens and pos is the position of the new queen:

```
[-1] ; queens # newQueen
```

This is referred to as the *positioning relation*. The positions available for the new queen must be restricted so that it does not threaten any existing queen. Relation attack, tests for such an attack, and restricts the range of the positioning relation so that attacking queens are excluded:

```
[-1] ; queens # newQueen -> attack
```

This expression still relates n to the pair $(board, pos)$ but the position of the new queen (pos) is now such that it does not threaten any of the $(n-1)$ queens. The new board is constructed by relAdd which adds the nth queen to the relation that stores the positions of the $(n-1)$ queens:

```
[-1] ; queens # newQueen -> attack ; relAdd
```

Finally the base case must override the recursion:

```
[-1] ; queens # newQueen -> attack ; relAdd @ {(0,{})}
```

The relation newQueen holds between n and each possible position $(n, column)$ for a new queen. It uses colPositions which places an nth queen in row n at each column. Containership (`[cont]`) is used to non-deterministically extract one position. At the implementation level this means the queen is placed in each column in turn and a set of possible new positions is created.

The relation attack relates a given board position $(board, pos)$ to Unit if the queen at pos threatens some queen in $board$. It uses relation check which relates a pair of board co-ordinates denoting the positions of two queens to Unit if they threaten each other.

7.6.6 From Prolog to Drusilla

Some Prolog and Drusilla programs are similar, as the ancestor programs shown in Figure 7.17 demonstrate. Generally any Prolog program may be mapped into the Drusilla relational calculus from which it may be possible to derive a Drusilla program as discussed in section 7.5.7. This mapping preserves the declarative content of a Prolog program because predicate calculus forms a specific viewpoint of relational calculus [94]. However, the derived Drusilla program may differ in operational behaviour from the original Prolog because of the different operational semantics — lazy reduction semantics compared to resolution based theorem proving.

The transformation converts pure, declarative Prolog terms to calculus values:

Atoms become either character strings or numbers.

Logical variables are replaced by mathematical variables.

Lists are replaced by sequences as described in section 7.4 and pattern matching on lists for their decomposition is replaced by explicit use of *hd* and *tl* relations.

Functors are exchanged for n-tuples — the functor name is dropped but its arguments retained as the tuple.

Predicates are converted to n-ary relations as follows:

- If a predicate is unary then it may be thought of as denoting a set of (non-tupled) values. This can be replaced by a binary relation between the values and `Unit`.
- An n-ary predicate (where $n > 1$) is formed by prefixing an n-tuple with an n-place predicate symbol. Such a predicate may be mapped to a binary relation by splitting the n-tuple into a pair of smaller tuples. One tuple is identified as forming an element of the relation domain, the other an element of the relation range. For example `p(x,y,z)` becomes `(x,y) p (z)`.

Goals within a clause body are combined with the conjunction operator ($\wedge$).

Rules that comprise a procedure denote alternatives within that procedure and are therefore combined with the disjunction operator ($\vee$).

As an example, the standard Prolog membership predicate is shown with its relational calculus translation in Figure 7.19. The corresponding Drusilla membership relation is derived in Figure 7.14.

```
member( X, [X|_] ).
member( X, [_|Xs] ) :- member( X,Xs ).
```

xs member y $\Leftrightarrow$ xs hd y $\vee$ (xs tl zs $\wedge$ zs member y)

Figure 7.19: List membership in Prolog (upper) and Drusilla calculus (lower)

This conversion loses Prolog's non-directionality of predicates because part of the predicate is fixed as the domain and part fixed as the range. However, the derived Drusilla relation has alternative modes of use. The degree of success for such transformations remains a question for future research.

7.7 Conclusions

This chapter has evaluated the success of Drusilla as a programming system in terms of both the implementation and the possible programming styles.

The implementation effort is justified by favourable comparison of RPL programs with their Drusilla translations. Furthermore, the example programs demonstrate that Drusilla provides certain freedom of expression that RPL lacks. Certain problems have concise solutions in Drusilla but not in RPL because many more operators are applicable to intensional relations. In particular, Drusilla demonstrates that relational programming can possess aspects of both functional and logic programming.

The main advantages are:

- Non-determinism and search-based computation are naturally handled.
- The programmer controls flow of execution in a structured fashion. The form of control present in functional programming is introduced to logic programming.
- Exceptions are handled in a natural manner.
- Programs can be formally derived from calculus specifications.

The main disadvantages are:

- Laziness is lost for recursion over infinite data structures.
- Recursive decomposition of relational data structures is inelegant and expensive.

Chapter 8

Conclusion

8.1 Introduction

This chapter draws together the theoretical and practical threads of the thesis. Section 8.2 draws conclusions about the design of the Drusilla language. Section 8.3 draws conclusions about the implementation of Drusilla. Section 8.4 suggests directions for future research on relational programming.

8.2 Conclusions on the Drusilla Language

8.2.1 The Underlying Mathematical Model

Chapter 3 defined the typed relational calculus that underlies Drusilla and used it to formally define the built-in relational operators and create a universe of discourse for the language. The calculus is typed to allow Milner type inference to statically check Drusilla programs.

MacLennan based his model of relational programming on sets from which relations can be derived. Our model is based on relations from which sets can be derived. This appears more appropriate for relational programming since it is difficult to identify the corresponding set of values for any relation that is polymorphic or higher-order.

8.2.2 Comparison with other Relational Languages

Chapter 2 discussed relational languages proposed by other researchers and concluded that a relational language that includes aspects of these languages is needed. The merits and demerits of Drusilla are compared with these languages.

Popplestone identifies relational programming with logic programming — his proposed language is similar to Prolog but uses forward inference. Chapter 7 discussed Drusilla as a logic programming language.

Möller's relational language was designed to exhibit certain algebraic properties that assist formal derivation of algorithms. A related aspect of Drusilla is the ability to derive relations from calculus expressions. Also, the laws used for

algebraic manipulation form a relational algebra but the emphasis is on making programs executable rather than on algorithm derivation.

The GREL language forced the programmer to view relations purely as set-valued functions by fixing this as their representation. Drusilla provides this view plus two others. Furthermore, in Drusilla the alternative results of an expression are hidden to allow the programmer to reason about relationships. Operators like GREL's `UNION` are available but are used to glue relations together directly, not to gather sets of results. Drusilla offers greater relational abstraction — relations may be reasoned about as entities in their own right.

The operators used in the Ruby language form a subset of those used in Drusilla. This is no accident — Ruby influenced the design of Drusilla as discussed in chapter 3. However, whereas Ruby is designed to be a hardware specification language, Drusilla is designed to be a general purpose programming language.

The relational language that is closest to Drusilla both in terms of design and intended use is RPL. Drusilla is a successor to RPL with a more sophisticated implementation — the aim is to demonstrate that relational programming can be implemented in more generality than in RPL. We regard Drusilla as being the next step in the evolution of relational programming.

8.2.3 Functional Programming, Logic Programming and Drusilla

Chapter 1 defined those features of functional and logic languages that account for their expressive power. Chapter 7 demonstrated that Drusilla exhibits at least some of these properties. The results of this research has provided new insight into the link between functional and logic programming.

Although Drusilla is not a functional language, it does incorporate lazy evaluation, static typing, higher-order definitions and applicative expressions.

If the central aspects of functional programming are debatable, then the central aspects of logic programming certainly are. Many people consider unification, logical variables and partially defined data structures essential. Others stress the importance of showing some theorem is inconsistent with given axioms by use of a theorem prover. If a logic programming language can be based on relational algebra instead of relational calculus then Drusilla is an example of such a language. However the absence of unification means that relations are less polymodal and some computations are less efficient than in Prolog, as discussed in chapter 7.

Comparing Drusilla with previous attempts to merge functional and logic programming, the most successful of the languages discussed in chapter 2 are LML, Eqlog, Tablog, CFP, and equational languages.

LML is a clean extension of functional programming. However the logic aspect (the theory data type) and the functional aspect (the main part of the language) have only a limited scope for interaction through set abstraction notation. Similarly constraint functional programming involves separate language components — the functions and the constraints. In Drusilla there is only one component (relations), which incorporates functional and logic aspects. CFP needs to manipulate two forms of object for computation: functions and constraints. It also

needs sets to handle non-determinism. Drusilla manipulates only relations, but these naturally permit reasoning about non-determinism.

Eqlog and Tablog are both large, complex, first-order languages. Drusilla is much smaller and higher-order.

Equations are not always a natural notation for expressing real world relationships since they express equivalences. Relations are more intuitive for logic programming since they express general relationships.

So Drusilla has its merits compared with other proposed functional-logic languages, but it does have several problems and shortcomings:

Absence of function application: It was a deliberate decision to base Drusilla entirely on data values and relations over those data values. Functions were not introduced so that relations could be explored as the sole computational mechanism. Although the operators form a rich relational algebra they are the only means by which expressions can be constructed. i.e. the operators themselves limit expressive power.

Expressions are concise to formulate when there is a natural pipeline structure for information flow through the constituent relations: composition operators are used. However, gluing problems may occur when the relations to be combined have no obvious pipeline structure — a problem similarly encountered with the FP language. This problem becomes more acute as programs increase in size. To some extent it is circumvented by the presence of relation specialisation notation — indeed specialisation was introduced precisely for this purpose. However, if new operators could be defined or if functional relations could be identified and applied functionally to arguments then expressive power would be increased considerably.

Data structures: chapter 7 identified the problem for recursion over relational data structures (extensionally represented relations). This problem could be solved by adding algebraic datatypes to Drusilla.

8.3 Conclusions on the Drusilla Implementation

8.3.1 Removing The Representation Bottleneck

Chapter 4 christened the fixed representation scheme of RPL the *representation bottleneck* since it forms a barrier to freedom of expression. The aim of the Drusilla system is to increase expressive power by abstracting the programmer away from relation representations.

The representation bottleneck is widened to the limit of computation by *typed representation inference*, an algorithm based on Milner polymorphic type inference. Typed representation inference is perhaps the most satisfying result from the research described in this thesis. Furthermore it has potential as a general mechanism for resolving operator overloading and implementing ad-hoc polymorphism in programming languages. Cardelli and Wegner [20] state that ad-hoc

polymorphism is a combination of overloading and coercion. Typed representation inference provides a mechanism for implementing such polymorphism while leaving the language implementor free to decide the exact combination of overloading and coercion to be used.

Unfortunately some expressions cannot be represented. If an expression cannot be given a representation it is because some operator is applied to a relation that has a representation for which it is undefined. This reflects the computational limits of the operators. Moded types are used to distinguish the otherwise homogeneous program and data for the programmer and are needed to reify representations back to the programmer when an expression is unrepresentable. Some expressions that have no representation may be represented through *representation manipulation.* This is algebraic manipulation based on known laws or relation equivalences and is helped by two mechanisms:

Law Analysis isolates those laws which, when used as a rewrite rule in a specific direction, improve expression representation. The typed representation information encoded in the rules by analysis obviates much of the need to retypecheck. Unfortunately there are few representation improving laws and this is a fundamental problem for manipulation, although few expressions are without representation. Also unrepresentable expressions are often uncomputable, because the typed representation rules for each operator reflect computational constraints.

Meta-level Inference provides intelligent control of the search involved in expression manipulation. However, it relies on methods that are developed in an ad-hoc manner.

This manipulation is unfortunately not particularly successful since few laws produce rules that improve representation. i.e. few rules transform away those operators that can generate the undefined representation.

Search space manipulation is algebraic manipulation that improves program execution behaviour by removing unnecessary computation. It is more successful than representation manipulation as a greater number of the laws are of use. However, its success for any given expression is difficult to judge because lazy evaluation avoids much unnecessary computation. The search should never be made worse but it might not be improved.

8.3.2 The Implementation Architecture

The architecture of the Drusilla implementation was described in chapter 6. All syntactic analysis is based on Fairbairn's 'let form follow function' [33]. While this may not be the most efficient form of parser it does allow the parser structure to be close to the BNF defining the expression grammar.

Calculus type inference is the relational form of the Milner polymorphic type inference performed at compile time in functional languages. It provides a powerful mechanism for static detection of program incorrectness and as such is undoubtedly a success. This is executed before any other analysis for two reasons:

- Other, computationally expensive, analysis techniques are not applied to obviously incorrect programs.
- Other analyses can be based on the premise that every program expression is well-formed.

Typed representation inference is applied to the expressions in a maximally strong component before symbolic manipulation for two reasons:

- This analysis isolates those definitions that have no representation and hence need to be manipulated.
- Lazy manipulation can then be used — meta-level inference can be used to direct manipulation search and typed representation information encoded in the rewrite rules can be used to obviate much re-typechecking.

This architecture is also used in the Drusilla compiler. The implementation is not the most efficient but it was never intended to be. The sole aim is to demonstrate that relational programming can be made more expressive than RPL or GREL through more sophisticated implementation techniques. The creation of an efficient, production quality, compiler is a subject for future research.

8.4 Future Relational Programming Research

8.4.1 The Semantics of Drusilla

The mathematical model of Drusilla is not sufficient, nor was it ever intended, to constitute a formal semantics for Drusilla. The semantics are defined in terms of typed representation inference, the Drusilla interpreter / compiler, and the Miranda definitions of the operators. It would obviously be preferable to have a formal semantics for relational programming.

8.4.2 The Need for Function Application

The inability to apply relations functionally places a constraint on the expressive power of relational programming. In particular it may lead to gluing problems when programs increase in size, and domain and range elements become more complex.

This is a problem that should, intuitively, be solvable. The set of all (binary) functions is a subset of the set of all binary relations. Ideally the system should be able to identify functional relations and allow the programmer to apply them as functions. Static analysis could identify such relations. For example, point-to-point functions could be included as a new representation and incorporated in typed representation inference rules. Alternatively Hutton [49] describes an analysis for identifying functional relations in Ruby.

Any parameterised definition that does not use anonymous parameters (_) *must* be functional. For example, the definition of cartesian product for relations denoting sets is:

```
(r,s) cartProd (r ; inv s)
```

The domain values for this relation are pairs of relations that are described by the syntactic pattern: `(r,s)`. The range element related to any such domain element is determined by the defining expression — in this case `r ; inv s`. This definition defines a relation whose functionality can be determined from the syntax. This relation could be used functionally, like the operators, in the formulation of expressions, for example:

```
(r,s,t) tripleProd  cp (r,cp (s,t))
```

This would allow the definition of new operators.

Normally all relations are first class values and every relation has the same status as every other relation. The only distinction between relations is their possible modes of use, with exception of the operators that can be applied as functions. There is no problem in recognising operators — their syntax indicates this. Similarly in the extension suggested above the new operators can be recognised by their identifiers. This extension can be taken further to make functional relations first-class citizens. This permits functional application of formal parameters to values. For example the definition:

```
(f,r) binOpApply   f ({(1,Unit),(2,Unit),(3,Unit)},r)
```

The values in domain of this relation are pairs of relations where the first relation is known to be functional. The first relation, `f`, could be any of the built-in operators, or an explicitly defined functional relation, such as `cartProd` above.

If functional relations were to be accepted as first-class citizens and programs remain type safe then a new class of objects — functions — would have to be created. If functions were to be still recognised as relations (as indeed they are) then they would have to become a subtype of relation. However this would prevent full polymorphic type inference from taking place because there is no known algorithm for inferring polymorphic subtype relationships [20].

Thus, if functions are to be introduced and recognised as a subclass of relations then there are two possible routes:

- Remove the type system. This can be discounted given the premise of chapter 1 that type inference is a good aspect of functional programming.
- Derive a new type inference algorithm, one that is capable of inferring subtypes. This is a general topic for future research.

Knowledge of relation functionality could also be exploited at the implementation level. A *point-to-point function* representation would improve laziness for relations that recurse over infinite extensional relations as discussed in chapter 7.

8.4.3 Implementing Relational Programming Efficiently

Chapter 6 described how Drusilla programs can be compiled into Miranda programs. Compilation into a functional language that is itself compiled, such as

Haskell or LML, would greatly improve efficiency. Also Haskell contains arrays, which can represent any relation defined in extension that is of a known fixed size. The idea of using arrays for implementing sets whose size can be determined statically has previously been explored by Paige [81, 80, 18]. As he observes this permits constant time access to elements.

This idea can be taken one step further — a compiler could generate object code in a lower level language, for example C, which might further improve efficiency.

8.4.4 Querying Relational Databases

The fact that Drusilla is based on a relational algebra that includes analogues of Codd's basic algebraic operations suggests that it could be used to query relational databases. As Drusilla is based on binary relations, it would seem most natural for it to interact with a database that is founded on binary relations.

Drusilla would have more computational power than a conventional query language because it is Turing-complete. From this perspective it would be interesting to compare Drusilla with FDL, the computationally complete query language discussed in chapter 2.

8.4.5 Symbolic Inversion of Relations

The problem of inverting functions has been considered for functional programs, for example Runciman [93]. The inverse of a many-to-one function is a relation and hence must be represented by a set-valued function. However the inverse of a relation is always a relation, moreover, in a relational language all expressions are constructed from the built-in operators. Inversion of relations should therefore be a problem more tractable for relational programming.

One problem with the Drusilla system is that the inverse of an intensionally represented relation can only be represented by a characteristic function. This means that the inverse of a relation can never be used to generate range values from given domain values, only to test whether a given pair of domain and range values are related. Many expressions that cannot be represented use the relation inversion operator. This is reflected by the fact that several of the representation improving rewrite rules work by shifting or removing inversion operations. For example, the laws:

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

If the inverse of expressions could be evaluated symbolically then the absence of any tangible relation representation would cease to be of importance. This would involve a new set of rules for algebraic manipulation. Each rule in the set would map a given expression to be inverted into an equivalent expression in which only the leaves are inverted. For an expression constructed from an unary operator `f` or from a binary operator $\oplus$ the rules would be of the form:

```
inv (inv s) → s
inv (neg s) → neg (inv s)
inv (dom s) → rng (inv s)
inv (rng s) → dom (inv s)
inv (set s) → flip ; set (inv s)
inv (r ⋁ s) → inv r ⋁ inv s
inv (r ⋀ s) → inv r ⋀ inv s
inv (r \ s) → inv r \ inv s
inv (s ; r) → inv r ; inv s
inv (r @ s) → (inv r -> dom s) ⋁ inv s
inv (s img r) → dom (inv r >> s)
inv (r # s) → (inv r || inv s) >> set id ; fst
inv (r || s) → inv r || inv s
inv (p << t) → inv t >> p
inv (t >> p) → p << inv t
inv (p <- t) → inv t -> p
inv (t -> p) → p <- inv t
where
(x,y) swap (y,x)
(x,y) fst (x)
(x) id (x)
```

Figure 8.1: Rules for simplfying relation inverses

$$inv\ (f\ t) \rightarrow f'\ (inv\ t)$$
$$inv\ (r \oplus s) \rightarrow (inv\ r) \oplus' (inv\ s)$$

Here f' and $\oplus'$ are new operators for combining the inverted expressions. The actual rules are presented in Figure 8.1. Their correctness can be proved in the same way as manipulation laws — by translation to a common expression in the relational calculus — as described in chapter 5. The rules do not cover the operators `[cont]` or `card` because `[cont]` cannot be used as a compound relation designation and `card` yields a number, which obviously cannot be inverted.

The problem with symbolic inversion is that the leaves of the expression tree must be inverted. The leaves may be relational operators and the inverse of some operators cannot be computed. For example, the inverse of union relates any relation `t` to any two relations, `r` and `s` such that `r ⋁ s = t`. If `t` is higher-order or polymorphic then the relations `r` and `s` cannot be computed. Arithmetic operator designations, for example `[+]`, can be inverted, but the inversion is represented by a characteristic function. This is self defeating since the goal is to generate a relation inverse that can be represented by a set-valued function.

8.5 Summary of Conclusions

Relational programs can have both functional and logic aspects: higher-order, polymorphic, applicative expressions that can be lazily evaluated, and permit reasoning about relationships between entities, non-determinism and search-based computation. This is significant since Hudak [43] observes that previous functional-logic integrations have not been entirely satisfactory in the context of higher-order definitions and lazy evaluation. The research sheds new light on the link between the two paradigms.

The success of the more sophisticated implementation is due to typed representation inference which widens the representation bottleneck closer to the limit of computation. The analysis better preserves relation abstraction and consequently simplifies reasoning about relationships and non-determinism. Symbolic manipulation is disappointing although helped by law analysis which isolates those laws that produce representation improving rewrite rules. A strategy based on meta-level inference seems appropriate. Programs can be interpreted but compilation is more efficient and the resulting functional program guaranteed type correct.

However a formal semantics is needed for relational programming. Introduction of function application, or at least the ability to define new operators appears to be a good route for improving expressive power. To further test how expressive a relational language is, larger programs must be tried and compiled more efficiently.

Consequently relational programming is a declarative programming paradigm that can be applied to applications where a functional or logic language might normally be used.

Appendix A

Example Compiled Drusilla Programs

This appendix gives examples of the Miranda programs produced by the Drusilla ccompiler for a number of Drusilla programs.

Figure A.1 shows the Drusilla permutations of a sequence definitons presented in chapter 7. Figure A.2 shows this program compiled to Miranda.

Figure A.3 shows the Drusilla solution to the Prolog-style ancestors program along with the compiled Miranda program.

Figure A.4 shows the Drusilla solution to the eight queens problem along with the compiled Miranda program.

```
-- permutations of a sequence
perm = {({},{})} \/ (hd # (tl ; perm) ; insert).

-- permutations of a sequence
permH = (insert,{},_)foldr.

-- non-deterministic insert
(x,_) insert (x,_)cons  \/ (hd # (tl ; (x,_)insert) ; cons).

-- fold right an operator over a sequence
(op,x,_) foldr {({},x)} \/ (hd # (tl ; (op,x,_)foldr) ; op).

hd = [cont](1,_).

tl = {({},Unit)} <- tl2.

(s) tl2 dom s << [+ 1] ; s.

(x,s) cons {(1, x)} \/ inv (inv s ; [1 +]).

seqZ = {(1,"b"),(2,"c"),(3,"d")}.

s = {(1,"a"),(2,"b"),(3,"c"),(4,"d")}.
```

Figure A.1: Drusilla permutations program

```
hd0 = concat . map local1_ . cont
 where
 local1_ (1,dUM1) =  [dUM1]
 local1_ other = []
tl21 s = [compAlAl ((domResAlSf ((domAl (s),local1_)),s))]
 where
 local1_ dUM1 = (mklist . plusF) (dUM1,1)
tl22 s = [compSfSf ((domResSfSf ((domSf (s),local1_)),s))]
 where
 local1_ dUM1 = (mklist . plusF) (dUM1,1)
tl3 = domAntAlSf (([([],True)],tl21))
cons4 (x,s) =
  [unionAlAl (([(1,x)],
   invAl (compAlSf ((invAl (s),local1_)))))]
 where
 local1_ dUM1 = (mklist . plusF) (1,dUM1)
insert5 (x,dUM1) =
  unionSfSf ((local2_,compSfSf ((dualSfSf ((hd0,
compSfSf ((tl3,local1_)))),cons4)))) dUM1
 where
 local2_ dUM1 = cons4 (x,dUM1)
 local1_ dUM1 = insert5 (x,dUM1)
perm6 = unionAlSf (([([],[])],compSfSf ((dualSfSf ((hd0,
  compSfSf ((tl3,perm6)))),insert5))))
foldr7 (op,x,dUM1) =
  unionAlSf (([([],x)],compSfAl ((dualSfSf
   ((hd0,compSfSf ((tl3,local1_)))),op)))) dUM1
 where
 local1_ dUM1 = foldr7 (op,x,dUM1)
foldr8 (op,x,dUM1) =
  unionAlSf (([([],x)],compSfSf ((dualSfSf ((hd0,
    compSfSf ((tl3,local1_)))),op)))) dUM1
 where
 local1_ dUM1 = foldr8 (op,x,dUM1)
foldr9 ((op,x,dUM1),dUM0) =
  unionAlCf (([([],x)],compCfAl ((dualSfCf ((hd0,
    compSfCf ((tl3,local1_)))),op)))) (dUM1,dUM0)
 where
 local1_ (dUM1,dUM0) = foldr9 ((op,x,dUM1),dUM0)
permH10 =
  local1_
 where
 local1_ dUM1 = foldr8 (insert5,[],dUM1)
seqZ11 = [(1,"b"),(2,"c"),(3,"d")]
s12 = [(1,"a"),(2,"b"),(3,"c"),(4,"d")]
```

Figure A.2: Drusilla permutations program compiled to Miranda

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

```
parent0 =
  ([("Drusus","Germanicus"),("Antonia","Germanicus"),
    ("Drusus","Claudius"),("Antonia","Claudius"),
    ("Germanicus","Caligula"),("Agrippina","Caligula"),
    ("Germanicus","Drusilla"),("Agrippina","Drusilla"),
    ("Caligula","Julia Drusilla"),("Caesonia","Julia Drusilla")]
male1 = ([("Drusus",True),("Germanicus",True),
          ("Claudius",True),("Caligula",True)])
female3 = (negAl(male1))
father4 = (domResAlAl((male1,parent0)))
mother5 = (domResCfAl((female3,parent0)))
id6 x = [(x)]
parentCouple7 =
   (diffAlSf((compAlAl((parent0,invAl(parent0))),id6)))
grandfather8 = (compAlAl((father4,parent0)))
grandmother9 = (compAlAl((mother5,parent0)))
sibling10 = (diffAlSf((compAlAl((invAl(parent0),parent0)),id6)))
sister11 = (domResCfAl((female3,sibling10)))
brother12 = (domResAlAl((male1,sibling10)))
uncle13 = (unionAlAl((compAlAl((brother12,parent0)),
          compAlAl((compAlAl((married7,sister11)),parent0)))))
```

Figure A.3: Ancestors programs in Drusilla (upper) and Miranda (lower)

```
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)})).
```

```
colPositions0 n =
  [[(n,1),(n,2),(n,3),(n,4),(n,5),(n,6),(n,7),(n,8)]]
newQueen1 =
  compSfSf ((colPositions0,cont))
check2 ((i,dUM1),(m,n)) =
  unionAlSf (([(n,True)],compSfAl ((local1_,unionAlAl
  ((([(minusF ((i,m)),True)],[(minusF ((m,i)),True)])))))) dUM1
 where
 local1_ dUM1 = (mklist . minusF) (dUM1,n)
attack3 (dUM1,pos) =
  compSfSf ((cont,local1_)) dUM1
 where
 local1_ dUM1 = check2 (dUM1,pos)
relAdd4 (r,(x,y)) =
  [unionAlAl ((r,[(x,y)]))]
relAdd5 (r,(x,y)) =
  [unionSfAl ((r,[(x,y)]))]
relAdd6 (r,(x,y)) =
  [unionCfAl ((r,[(x,y)]))]
queens7 =
  overSfAl ((compSfSf ((rngAntSfSf ((dualSfSf ((compSfSf
  ((local1_,queens7)),newQueen1)),attack3)),relAdd4)),[(0,[])]))
 where
 local1_ dUM1 = (mklist . minusF) (dUM1,1)
queensNo8 n =
  [cardAl (domResAlSf (([(n,True)],queens7)))]
```

Figure A.4: Eight queens in Drusilla (upper) and Miranda (lower)

Appendix B

Typed Representation Inference Rules

$$\frac{\mathrm{r} :: \mathrm{AL}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{inv\ r} :: \mathrm{AL}[\mathrm{B} \leftrightarrow \mathrm{A}]}$$

$$\frac{\mathrm{r} :: \mathrm{SF}[\mathrm{A} \leftrightarrow B^{=}]}{\mathrm{inv\ r} :: \mathrm{CF}[B^{=} \leftrightarrow \mathrm{A}]}$$

$$\frac{\mathrm{r} :: \mathrm{CF}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{inv\ r} :: \mathrm{CF}[\mathrm{B} \leftrightarrow \mathrm{A}]}$$

Figure B.1: Typed representation inference rules for relation inverse

$$\frac{\mathrm{r} :: \mathrm{AL}[A^{=} \leftrightarrow B^{=}]}{\mathrm{neg\ r} :: \mathrm{CF}[A^{=} \leftrightarrow B^{=}]}$$

$$\frac{\mathrm{r} :: \mathrm{SF}[\mathrm{A} \leftrightarrow B^{=}]}{\mathrm{neg\ r} :: \mathrm{CF}[\mathrm{A} \leftrightarrow B^{=}]}$$

$$\frac{\mathrm{r} :: \mathrm{CF}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{neg\ r} :: \mathrm{CF}[\mathrm{A} \leftrightarrow \mathrm{B}]}$$

Figure B.2: Typed representation inference rules for relation negation

$$\frac{r :: AL[A \leftrightarrow B]}{\text{dom } r :: AL[A \leftrightarrow un]}$$

$$\frac{r :: SF[A \leftrightarrow B]}{\text{dom } r :: SF[A \leftrightarrow un]}$$

$$\frac{r :: CF[A \leftrightarrow B]}{\text{dom } r :: \bot_{TR}}$$

Figure B.3: Typed representation inference rules for relation domain

$$\frac{r :: AL[A \leftrightarrow B]}{\text{rng } r :: AL[B \leftrightarrow un]}$$

$$\frac{r :: SF[A \leftrightarrow B]}{\text{rng } r :: \bot_{TR}}$$

$$\frac{r :: CF[A \leftrightarrow B]}{\text{rng } r :: \bot_{TR}}$$

Figure B.4: Typed representation inference rules for relation range

$$\frac{r :: AL[A \leftrightarrow B]}{\text{card } r :: num}$$

$$\frac{r :: SF[A \leftrightarrow B]}{\text{card } r :: \bot_{TR}}$$

$$\frac{r :: CF[A \leftrightarrow B]}{\text{card } r :: \bot_{TR}}$$

Figure B.5: Typed representation inference rules for relation cardinality

$$\frac{\mathrm{r} :: \mathrm{AL}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{set\,r} :: \mathrm{AL}[(\mathrm{A} \times \mathrm{B}) \leftrightarrow \mathrm{un}]}$$

$$\frac{\mathrm{r} :: \mathrm{SF}[\mathrm{A} \leftrightarrow B^{=}]}{\mathrm{set\,r} :: \mathrm{SF}[(\mathrm{A} \times B^{=}) \leftrightarrow \mathrm{un}]}$$

$$\frac{\mathrm{r} :: \mathrm{CF}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{set\,r} :: \mathrm{CF}[(\mathrm{A} \times \mathrm{B}) \leftrightarrow \mathrm{un}]}$$

Figure B.6: Typed representation inference rules for set view of a relation

$$\frac{\mathrm{r} :: \mathrm{AL}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{cont\,r} :: (\mathrm{A} \times \mathrm{B})}$$

$$\frac{\mathrm{r} :: \mathrm{SF}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{cont\,r} :: \bot_{TR}}$$

$$\frac{\mathrm{r} :: \mathrm{CF}[\mathrm{A} \leftrightarrow \mathrm{B}]}{\mathrm{cont\,r} :: \bot_{TR}}$$

Figure B.7: Typed representation inference rules for relation containership

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

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

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

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

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

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

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

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

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

Figure B.8: Typed representation inference rules for relation composition

$$\frac{r :: AL[A^{=} \leftrightarrow B] \quad s :: AL[A^{=} \leftrightarrow B]}{r \,@\, s :: AL[A^{=} \leftrightarrow B]}$$

$$\frac{r :: AL[A^{=} \leftrightarrow B] \quad s :: SF[A^{=} \leftrightarrow B]}{r \,@\, s :: SF[A^{=} \leftrightarrow B]}$$

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow B]}{r \,@\, s :: \bot_{TR}}$$

$$\frac{r :: SF[A^{=} \leftrightarrow B] \quad s :: AL[A^{=} \leftrightarrow B]}{r \,@\, s :: SF[A^{=} \leftrightarrow B]}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: SF[A \leftrightarrow B]}{r \,@\, s :: SF[A \leftrightarrow B]}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow B]}{r \,@\, s :: \bot_{TR}}$$

$$\frac{r :: CF[A^{=} \leftrightarrow B^{=}] \quad s :: AL[A^{=} \leftrightarrow B^{=}]}{r \,@\, s :: CF[A^{=} \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B^{=}] \quad s :: SF[A \leftrightarrow B^{=}]}{r \,@\, s :: CF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow B]}{r \,@\, s :: \bot_{TR}}$$

Figure B.9: Typed representation inference rules for relation override

$$\frac{r :: AL[A^{=} \leftrightarrow B] \qquad s :: AL[A^{=} \leftrightarrow C]}{r \,\#\, s :: AL[A^{=} \leftrightarrow (B \times C)]}$$

$$\frac{r :: AL[A \leftrightarrow B] \qquad s :: SF[A \leftrightarrow C]}{r \,\#\, s :: AL[A \leftrightarrow (B \times C)]}$$

$$\frac{r :: AL[A^{=} \leftrightarrow B^{=}] \quad s :: CF[A^{=} \leftrightarrow C]}{r \,\#\, s :: CF[A^{=} \leftrightarrow (B^{=} \times C)]}$$

$$\frac{r :: SF[A \leftrightarrow B] \qquad s :: AL[A \leftrightarrow C]}{r \,\#\, s :: SF[A \leftrightarrow (B \times C)]}$$

$$\frac{r :: SF[A \leftrightarrow B] \qquad s :: SF[A \leftrightarrow C]}{r \,\#\, s :: SF[A \leftrightarrow (B \times C)]}$$

$$\frac{r :: SF[A \leftrightarrow B] \qquad s :: CF[A \leftrightarrow C]}{r \,\#\, s :: CF[A \leftrightarrow (B \times C)]}$$

$$\frac{r :: CF[A^{=} \leftrightarrow B] \qquad s :: AL[A^{=} \leftrightarrow C^{=}]}{r \,\#\, s :: CF[A^{=} \leftrightarrow (B \times C^{=})]}$$

$$\frac{r :: CF[A \leftrightarrow B] \qquad s :: SF[A \leftrightarrow C^{=}]}{r \,\#\, s :: CF[A \leftrightarrow (B \times C^{=})]}$$

$$\frac{r :: CF[A \leftrightarrow B] \qquad s :: CF[A \leftrightarrow C]}{r \,\#\, s :: CF[A \leftrightarrow (B \times C)]}$$

Figure B.10: Typed representation inference rules for dual composition

$$\frac{r :: AL[A \leftrightarrow B] \qquad s :: AL[C \leftrightarrow D]}{r \parallel s :: AL[(A \times C) \leftrightarrow (B \times D)]}$$

$$\frac{r :: AL[A^{=} \leftrightarrow B] \qquad s :: SF[C \leftrightarrow D]}{r \parallel s :: SF[(A^{=} \times C) \leftrightarrow (B \times D)]}$$

$$\frac{r :: AL[A^{=} \leftrightarrow B^{=}] \quad s :: CF[C \leftrightarrow D]}{r \parallel s :: CF[(A^{=} \times C) \leftrightarrow (B^{=} \times D)]}$$

$$\frac{r :: SF[A \leftrightarrow B] \qquad s :: AL[C^{=} \leftrightarrow D]}{r \parallel s :: SF[(A \times C^{=}) \leftrightarrow (B \times D)]}$$

$$\frac{r :: SF[A \leftrightarrow B] \qquad s :: SF[C \leftrightarrow D]}{r \parallel s :: SF[(A \times C) \leftrightarrow (B \times D)]}$$

$$\frac{r :: SF[A \leftrightarrow B^{=}] \qquad s :: CF[C \leftrightarrow D]}{r \parallel s :: CF[(A \times C) \leftrightarrow (B^{=} \times D)]}$$

$$\frac{r :: CF[A \leftrightarrow B] \qquad s :: AL[C^{=} \leftrightarrow D^{=}]}{r \parallel s :: CF[(A \times C^{=}) \leftrightarrow (B \times D^{=})]}$$

$$\frac{r :: CF[A \leftrightarrow B] \qquad s :: SF[C \leftrightarrow D^{=}]}{r \parallel s :: CF[(A \times C) \leftrightarrow (B \times D^{=})]}$$

$$\frac{r :: CF[A \leftrightarrow B] \qquad s :: CF[C \leftrightarrow D]}{r \parallel s :: CF[(A \times C) \leftrightarrow (B \times D)]}$$

Figure B.11: Typed representation inference rules for parallel composition

$$\frac{r :: AL[A \leftrightarrow B] \qquad s :: AL[A \leftrightarrow B]}{r \cup s :: AL[A \leftrightarrow B]}$$

$$\frac{r :: AL[A^{=} \leftrightarrow B] \qquad s :: SF[A^{=} \leftrightarrow B]}{r \cup s :: SF[A^{=} \leftrightarrow B]}$$

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

$$\frac{r :: SF[A^{=} \leftrightarrow B] \qquad s :: AL[A^{=} \leftrightarrow B]}{r \cup s :: SF[A^{=} \leftrightarrow B]}$$

$$\frac{r :: SF[A \leftrightarrow B] \qquad s :: SF[A \leftrightarrow B]}{r \cup s :: SF[A \leftrightarrow B]}$$

$$\frac{r :: SF[A \leftrightarrow B^{=}] \qquad s :: CF[A \leftrightarrow B^{=}]}{r \cup s :: CF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \qquad s :: AL[A^{=} \leftrightarrow B^{=}]}{r \cup s :: CF[A^{=} \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B^{=}] \qquad s :: SF[A \leftrightarrow B^{=}]}{r \cup s :: CF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \qquad s :: CF[A \leftrightarrow B]}{r \cup s :: CF[A \leftrightarrow B]}$$

Figure B.12: Typed representation inference rules for relation union

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

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

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow B]}{r \cap s :: AL[A \leftrightarrow B]}$$

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

$$\frac{r :: SF[A \leftrightarrow B^{=}] \quad s :: SF[A \leftrightarrow B^{=}]}{r \cap s :: SF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: SF[A \leftrightarrow B^{=}] \quad s :: CF[A \leftrightarrow B^{=}]}{r \cap s :: SF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: AL[A \leftrightarrow B]}{r \cap s :: AL[A \leftrightarrow B]}$$

$$\frac{r :: CF[A \leftrightarrow B^{=}] \quad s :: SF[A \leftrightarrow B^{=}]}{r \cap s :: SF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow B]}{r \cap s :: CF[A \leftrightarrow B]}$$

Figure B.13: Typed representation inference rules for relation union

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

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

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow B]}{r \setminus s :: AL[A \leftrightarrow B]}$$

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

$$\frac{r :: SF[A \leftrightarrow B^{=}] \quad s :: SF[A \leftrightarrow B^{=}]}{r \setminus s :: SF[A \leftrightarrow B^{=}]}$$

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

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

$$\frac{r :: CF[A \leftrightarrow B^{=}] \quad s :: SF[A \leftrightarrow B^{=}]}{r \setminus s :: CF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow B]}{r \setminus s :: CF[A \leftrightarrow B]}$$

Figure B.14: Typed representation inference rules for relation union

$$\frac{r :: AL[A^{=} \leftrightarrow B] \quad s :: AL[A^{=} \leftrightarrow un]}{r\ img\ s :: AL[B \leftrightarrow un]}$$

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: SF[A \leftrightarrow un]}{r\ img\ s :: AL[B \leftrightarrow un]}$$

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow un]}{r\ img\ s :: CF[B \leftrightarrow un]}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: AL[A \leftrightarrow un]}{r\ img\ s :: AL[B \leftrightarrow un]}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: SF[A \leftrightarrow un]}{r\ img\ s :: \bot_{TR}}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow un]}{r\ img\ s :: \bot_{TR}}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: AL[A \leftrightarrow un]}{r\ img\ s :: \bot_{TR}}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: SF[A \leftrightarrow un]}{r\ img\ s :: \bot_{TR}}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: CF[A \leftrightarrow un]}{r\ img\ s :: \bot_{TR}}$$

Figure B.15: Typed representation inference rules for image

$$\frac{s :: AL[A^{=} \leftrightarrow un] \quad r :: AL[A^{=} \leftrightarrow B]}{s \ll r :: AL[A^{=} \leftrightarrow B]}$$

$$\frac{s :: AL[A \leftrightarrow un] \quad r :: SF[A \leftrightarrow B]}{s \ll r :: AL[A \leftrightarrow B]}$$

$$\frac{s :: AL[A^{=} \leftrightarrow un] \quad r :: CF[A^{=} \leftrightarrow B]}{s \ll r :: CF[A^{=} \leftrightarrow B]}$$

$$\frac{s :: SF[A \leftrightarrow un] \quad r :: AL[A \leftrightarrow B]}{s \ll r :: AL[A \leftrightarrow B]}$$

$$\frac{s :: SF[A \leftrightarrow un] \quad r :: SF[A \leftrightarrow B]}{s \ll r :: SF[A \leftrightarrow B]}$$

$$\frac{s :: SF[A \leftrightarrow un] \quad r :: CF[A \leftrightarrow B]}{s \ll r :: CF[A \leftrightarrow B]}$$

$$\frac{s :: CF[A \leftrightarrow un] \quad r :: AL[A \leftrightarrow B]}{s \ll r :: AL[A \leftrightarrow B]}$$

$$\frac{s :: CF[A \leftrightarrow un] \quad r :: SF[A \leftrightarrow B]}{s \ll r :: SF[A \leftrightarrow B]}$$

$$\frac{s :: CF[A \leftrightarrow un] \quad r :: CF[A \leftrightarrow B]}{s \ll r :: CF[A \leftrightarrow B]}$$

Figure B.16: Typed representation inference rules for domain restriction

$$\frac{s :: AL[A^{=} \leftrightarrow un] \quad r :: AL[A^{=} \leftrightarrow B]}{s \leftarrow r :: AL[A^{=} \leftrightarrow B]}$$

$$\frac{s :: AL[A^{=} \leftrightarrow un] \quad r :: SF[A^{=} \leftrightarrow B]}{s \leftarrow r :: SF[A^{=} \leftrightarrow B]}$$

$$\frac{s :: AL[A^{=} \leftrightarrow un] \quad r :: CF[A^{=} \leftrightarrow B]}{s \leftarrow r :: CF[A^{=} \leftrightarrow B]}$$

$$\frac{s :: SF[A \leftrightarrow un] \quad r :: AL[A \leftrightarrow B]}{s \leftarrow r :: AL[A \leftrightarrow B]}$$

$$\frac{s :: SF[A \leftrightarrow un] \quad r :: SF[A \leftrightarrow B]}{s \leftarrow r :: SF[A \leftrightarrow B]}$$

$$\frac{s :: SF[A \leftrightarrow un] \quad r :: CF[A \leftrightarrow B]}{s \leftarrow r :: CF[A \leftrightarrow B]}$$

$$\frac{s :: CF[A \leftrightarrow un] \quad r :: AL[A \leftrightarrow B]}{s \leftarrow r :: AL[A \leftrightarrow B]}$$

$$\frac{s :: CF[A \leftrightarrow un] \quad r :: SF[A \leftrightarrow B]}{s \leftarrow r :: SF[A \leftrightarrow B]}$$

$$\frac{s :: CF[A \leftrightarrow un] \quad r :: CF[A \leftrightarrow B]}{s \leftarrow r :: CF[A \leftrightarrow B]}$$

Figure B.17: Typed representation inference rules for domain anti-restriction

$$\frac{r :: AL[A \leftrightarrow B^{=}] \quad s :: AL[B^{=} \leftrightarrow un]}{r \gg s :: AL[A \leftrightarrow B^{=}]}$$

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: SF[B \leftrightarrow un]}{r \gg s :: AL[A \leftrightarrow B]}$$

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: CF[B \leftrightarrow un]}{r \gg s :: AL[A \leftrightarrow B]}$$

$$\frac{r :: SF[A \leftrightarrow B^{=}] \quad s :: AL[B^{=} \leftrightarrow un]}{r \gg s :: SF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: SF[A \leftrightarrow B^{=}] \quad s :: SF[B^{=} \leftrightarrow un]}{r \gg s :: SF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: CF[B \leftrightarrow un]}{r \gg s :: SF[A \leftrightarrow B]}$$

$$\frac{r :: CF[A \leftrightarrow B^{=}] \quad s :: AL[B^{=} \leftrightarrow un]}{r \gg s :: CF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: SF[B \leftrightarrow un]}{r \gg s :: CF[A \leftrightarrow B]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: CF[B \leftrightarrow un]}{r \gg s :: CF[A \leftrightarrow B]}$$

Figure B.18: Typed representation inference rules for range restriction

$$\frac{r :: AL[A \leftrightarrow B^{=}] \quad s :: AL[B^{=} \leftrightarrow un]}{r \rightarrow s :: AL[A \leftrightarrow B^{=}]}$$

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: SF[B \leftrightarrow un]}{r \rightarrow s :: AL[A \leftrightarrow B]}$$

$$\frac{r :: AL[A \leftrightarrow B] \quad s :: CF[B \leftrightarrow un]}{r \rightarrow s :: AL[A \leftrightarrow B]}$$

$$\frac{r :: SF[A \leftrightarrow B^{=}] \quad s :: AL[B^{=} \leftrightarrow un]}{r \rightarrow s :: SF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: SF[B \leftrightarrow un]}{r \rightarrow s :: SF[A \leftrightarrow B]}$$

$$\frac{r :: SF[A \leftrightarrow B] \quad s :: CF[B \leftrightarrow un]}{r \rightarrow s :: SF[A \leftrightarrow B]}$$

$$\frac{r :: CF[A \leftrightarrow B^{=}] \quad s :: AL[B^{=} \leftrightarrow un]}{r \rightarrow s :: CF[A \leftrightarrow B^{=}]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: SF[B \leftrightarrow un]}{r \rightarrow s :: CF[A \leftrightarrow B]}$$

$$\frac{r :: CF[A \leftrightarrow B] \quad s :: CF[B \leftrightarrow un]}{r \rightarrow s :: CF[A \leftrightarrow B]}$$

Figure B.19: Typed representation inference rules for range anti-restriction

Bibliography

[1] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 9 – 31. Ellis Horwood, 1987.

[2] H. Abramson. A prological definition of HASL a purely functional language with unification based conditional binding expressions. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations [31]*, pages 73 – 129. Prentice-Hall, 1986.

[3] 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.

[4] D. Bailey. The University of Salford Lisp/Prolog system. *SOFTWARE — Practice and Experience*, 15(6):594–610, June 1985.

[5] R.G. Bandes. Constraining-unification and the programming language Unicorn. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations*, pages 397 – 410. Prentice-Hall, 1986.

[6] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. LEAF: A language which integrates logic, equations and functions. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations [31]*, pages 201 – 238. Prentice-Hall, 1986.

[7] M. Bellia et al. A two-level approach to logic plus functional programming integration. In A.J. Nijman J.W. de Bakker and P.C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe*, volume 258, pages 374 – 393. Springer-Verlag, June 1987. LNCS 258.

[8] M. Bellia and G. Levi. The relation between logic and functional languages: A survey. *The Journal of Logic Programming*, 3(3):217 – 236, 1986.

[9] R. Bird and P. Wadler. *Introduction To Functional Programming*. Prentice-Hall, 1988.

[10] H. Boley. RELFUN: A relational/functional integration with valued clauses. *ACM SIGPLAN Notices*, 21(12):87 – 98, December 1986.

[11] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[12] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Logic programming within a functional framework. Dipartimento di Informatica, Univerita di Pisa, Corso Italia, 40, I - 56100, Pisa, Italy, 1990.

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

[14] A. Bundy and B. Welham. Using meta-level inference for selective application of rewrite rule sets in algebraic manipulation. In W. Bibel and R. Kowalski, editors, *5th Conference on Automated Deduction*, pages 24 – 38. Springer-Verlag, 1980. LNCS 87.

[15] 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.

[16] P. Buneman, R.E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Transactions on Database Systems*, 7(2):164 – 186, June 1982.

[17] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.

[18] 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.

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

[20] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471 – 521, December 1985.

[21] 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.

[22] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377 – 387, 1970.

[23] E.F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65 – 98. Prentice-Hall, New York, 1972.

[24] S. Cohen. The APPLOG language. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations*, pages 239 – 276. Prentice-Hall, 1986.

[25] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of 9th International Conference on Principles of Programming Languages*, pages 207–212. ACM Press, January 1982.

[26] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[27] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot G. Lindstrom, editor, *Logic programming, Functions, Relations and Equations*, pages 37 – 70. Prentice-Hall, 1986.

[28] J. Darlington and Y-K Guo. The unification of functional and logic languages — towards constraint functional programming. Department of Computing, Imperial College, University of London.

[29] C.J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, 4th edition, 1986.

[30] S.K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418 – 450, July 1989.

[31] D. DeGroot and G. Lindstrom. *Logic Programming, Functions, Relations and Equations*. Prentice-Hall, 1986.

[32] N. Dershowitz and D.A. Plaisted. Equational programming. In D. Michie, J.E. Hayes, and J. Richards, editors, *Machine Intelligence*. Ellis Horwood, 1986.

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

[34] R. Fateman, A. Bundy, R. O'Keefe, and L. Sterling. Commentary on: Solving symbolic equations with PRESS. *ACM SIGSAM*, 22(2):27 – 40, April 1988.

[35] S.M. Freudenberger, J.T. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26 – 45, January 1983.

[36] J.H. Gallier. *Logic For Computer Science: Foundations of Automatic Theorem Proving*. Wiley, 1987.

[37] J. A. Goguen and J. Meseguer. Eqlog: Equality, types and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations*, pages 295 – 364. Prentice-Hall, 1986.

[38] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[39] P. Gray. *Logic, Algebra and Databases*. Ellis Horwood, 1984.

[40] P. Henderson. Purely functional operating systems. In P. Henderson J. Darlington and D.A. Turner, editors, *Functional Programming and its Applications*, pages 177 – 192. Cambridge University Press, 1982.

[41] C.M. Hoffman and M.J. O'Donnell. Programming with equations. *Transactions on Programming Languages and Systems*, 4(1):83–112, January 1982.

[42] C.J. Hogger. *Introduction To Logic Programming*. Academic Press, 1984.

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

[44] P. Hudak and P. Wadler (editors). Report on the programming language Haskell — a non-strict, purely functional language. Department of Computer Science, University of Glasgow, August 1991.

[45] G. Huet and D.C. Oppen. Equations and rewrite rules — a survey. In R.V. Book, editor, *Formal Language Theory — Perspectives and Open Problems*. Academic Press, 1980.

[46] J. Hughes. Compile time analysis of functional programs. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 117 – 153. Addison-Wesley, 1988.

[47] J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17 – 42. Addison-Wesley, 1988.

[48] J. Hughes and J. O'Donnell. Expressing and reasoning about non-deterministic functional programs. In K. Davis and J. Hughes, editors, *Proceedings of the 1989 Glasgow Functional Programming Workshop*, pages 308 – 328. Springer-Verlag, 1990.

[49] G. Hutton. Functional programming with relations. In *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, pages 126 – 140, 1991.

[50] M. Jarke and J. Koch. Query optimisation in database systems. *Computing Surveys*, 16(2), June 1984.

[51] B. Jayaraman and F.S.K. Silbermann. Equations, sets and reduction semantics for functional and logic programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 320 – 331. ACM Press, 1986.

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

[53] G. Jones and M. Sheeran. The study of butterflies. Technical Report PRG-TR-14-90, Oxford University Computing Laboratory, 1990.

[54] K.M. Kahn. Uniform — a language based upon unification which unifies much of lisp, prolog and act 1. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 411 — 438. Prentice-Hall, 1986.

[55] E. Kant. The selection of efficient implementations for a high-level language. *ACM SIGPLAN Notices*, 12(8):140 – 146, August 1977.

[56] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263 – 297. Pergamon Press, 1970.

[57] W.A. Kornfield. Equality for prolog. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations*, pages 279 — 294. Prentice-Hall, 1986.

[58] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP 74*, pages 569 – 574. North Holland, 1974.

[59] R.A. Kowalski. Algorithm = logic + control. *Communications of The ACM*, August 1979.

[60] R.A. Kowalski. *Logic for Problem Solving*. Elsevier, North Holland, 1979.

[61] R.A. Kowalski and M.H. Van Emden. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733 – 742, 1976.

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

[63] S-c Liu. Automatic data structure choice in SETL. Technical Report NSO-15, Courant Institute of Mathematical Sciences, Computer Science Dept, New York University, September 1979.

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

[65] 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.

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

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

[68] B. MacLennan. Four relational programs. Technical Report NPS52-86-023, Naval Postgraduate School, Monterey, California, November 1986.

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

[70] Y. Malachi. *Nonclausal Logic Programming*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA. 94305, March 1986.

[71] Y. Malachi, Z. Manna, and R. Waldinger. Tablog: A new approach to logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations*, pages 365 – 394. Prentice-Hall, 1986.

[72] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):92 – 121, January 1980.

[73] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33 – 70. North Holland, 1963.

[74] E. Meijer. *Calculating Compilers*. PhD thesis, Katholieke Universteit Nijmegen, 1992.

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

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

[77] B. Möller. A relational programming style. Papers of the 42nd meeting of I.F.I.P. Working Group 2.1, January 1991.

[78] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming, Toulouse*, pages 217 – 239. Springer-Verlag, 1984. LNCS 167.

[79] M. O. Newton. A combined logical and functional programming language. Technical Report 5172 : TR : 85, California Institute of Technology, Pasadena, California, 91125, 1985.

[80] R. Paige. Real-time simulation of a set machine on a RAM. In R. Janicki and W. Koczkodaj, editors, *Computing and Information, Vol II*, pages 69 – 73. Canadian Scholar's Press, Toronto, May 1989. ICCI 89.

[81] R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code — a case study. *Journal of Symbolic Computation*, 4(2):207 – 232, August 1987.

[82] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[83] R. Popplestone. Relational programming. In J. Hayes, D. Michie, and L. Mikulich, editors, *Machine Intelligence 9*, pages 3 – 25. Ellis Horwood, 1979.

[84] A. Poulovassilis and P. King. Extendiong the functional data model to computational completeness. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology — EDBT '90, International Conference on Extending Database Technology, Venice, Italy*, pages 75–91. Springer-Verlag, March 1990.

[85] C. Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[86] U.S. Reddy. On the relationship between logic and functional languages. In *Logic Programming, Functions, Relations and Equations*, pages 3 – 36. Prentice-Hall, 1986.

[87] J. A. Robinson and E. E. Sibert. Loglisp: An alternative to prolog. In J. Hayes, Y-H Pao, and D. Michie, editors, *Machine Intelligence 10*, pages 399 – 420. Ellis Horwood, 1982.

[88] J.A. Robinson. A machine-oriented logic based on the resolution principle. *ACM Journal*, 12:23–41, January 1965.

[89] S.J. Rosenchein and S.M. Katz. Selection of representations for data structures. *ACM SIGPLAN Notices*, 12(8), August 1977.

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

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

[92] C. Runciman. Resolving overloaded expressions in Ada. Technical Report YCS.35, Department of Computer Science, University of York, England, October 1980.

[93] C. Runciman. An inversion technique for functional programs. Technical Report YCS.76, Department of Computer Science, University of York, England, July 1985.

[94] J. Sanderson. *A Relational Theory of Computing.* Springer-Verlag, 1980. LNCS 80.

[95] M. Sato and T. Sakurai. QUTE: A functional language based on unification. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations*, pages 131 – 156. Prentice-Hall, 1986.

[96] G. Schmidt and T. Ströhlein. *Relations and Graphs.* Springer Verlag, 1990.

[97] 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.

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

[99] J.T. Schwartz. On programming — an interim report on the SETL project. Technical report, Courant Institute of Mathematical Sciences, Computer Science Dept, New York University, June 1975.

[100] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming With Sets — an Introduction to SETL*. Springer-Verlag, 1986.

[101] 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.

[102] J.M. Smith and Y-T Chang. Optimizing the performance of a relational algebra database interface. *Communications of the ACM*, 18(10):568 – 579, October 1975.

[103] G. Smolka. FRESH: A higher-order language based upon unification. In D. DeGroot G. Lindstrom, editor, *Logic programming, Functions, Relations and Equations*, pages 469 – 524. Prentice-Hall, 1986.

[104] J.M. Spivey. *The Z Notation A Reference Manual*. Prentice Hall, 1989.

[105] M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25 – 42, 1990.

[106] L. Sterling, A. Bundy, L. Byrd, R. O'Keefe, and B. Silvers. Solving symbolic equations with PRESS. *Journal of Symbolic Computation*, 7:71 – 84, 1989.

[107] C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, August 1967.

[108] P.A. Subrahmanyam and J-H You. FUNLOG: A computational model integrating logic programming and functional programming. In D. DeGroot and G. Lindstrom, editors, *Logic programming, Functions, Relations and Equations*, pages 157 – 198. Prentice-Hall, 1986.

[109] P. Suppes. *Axiomatic Set Theory*. Dover Publications, 1972.

[110] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73 – 89, September 1941.

[111] M. Tarver. DIALOG: A theorem-proving environment designed to unify functional and logic programming. Technical report, University of Edinburgh, Department of Computer Science, May 1989.

[112] A.M. Tenenbaum. Type determination for very high level languages. Technical Report NSO-3, Courant Institute of Mathematical Sciences, Computer Science Dept, New York University, October 1974.

[113] I. Toyn. *Exploratory Environments for Functional Programming.* PhD thesis, Department of Computer Science, University of York, York, England, April 1987. YCST 87/02.

[114] D.A. Turner. An overview of miranda. In D.A. Turner, editor, *Research Topics in Functional Programming,* pages 1 – 16. Addison - Wesley Publishing company, 1990.

[115] P.J. Voda and B. Yu. RF-Maple: A logic programming language with functions, types and concurrency. Technical report, Department of Computer Science, University of British Columbia, April 1984.

[116] 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.

[117] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc.* In C. Hall, J. Hughes, and J.T. O'Donnell, editors, *Proceedings of the 1988 Glasgow Workshop on Functional Programming,* pages 203 – 219. Springer-Verlag, 1989.

[118] D.S. Wile. Bare relations: Adding relational access to lisp. In *IFIP 1989,* 1989.

[119] D.S. Wile. Adding relational abstraction to programming languages. *ACM SIGSOFT International Workshop on Formal Methods in Software Development 9 – 11 May 1990, Software Engineering Notes,* 15(4):128 – 139, September 1990.

[120] P.H. Winston. *Artificial Intelligence.* Addison-Wesley, second edition, 1984.

[121] J. Woodcock and M. Loomes. *Software Engineering Mathematics.* Pitman, 1988.