An Integration of Reduction and Logic for Programming Languages

David A. Wright

A thesis submitted in
partial fulfillment
of the degree of

M.Sc (Applied Computer Science)

To Mom and Dad

Acknowledgements

Professor Dennis Riordan, my thesis supervisor, deserves my thanks for his critical assessment and advice during the preparation of this thesis. Also thanks to the team of "reviewers" who read an early paper on REDLOG which served as the foundation of this thesis.

My sincere thanks go to my parents for the tremendous support and encouragement that they have given me from the very beginning.

Thanks go to my fiance, Merridy, who more than anyone knows the trials and tribulations that occurred in the shaping of this work. Thanks for helping me to maintain my faith in it!

Thanks to the Rhodes Computer Science team for all that I have learnt from them over the past five years.

I was supported financially by Rhodes University and the Council for Scientic and Industrial Research during this year and owe them my thanks.

An Integration of Reduction and Logic for Programming Languages

David A. Wright

ABSTRACT

A new declarative language is presented which captures the expressibility of both logic programming languages and functional languages. This is achieved by conditional graph rewriting, with full unification as the parameter passing mechanism. The syntax and semantics are described both formally and informally, and examples are offered to support the expressibility claim made above. The language design is of further interest due to its uniformity and the inclusion of a novel mechanism for type inference in the presence of derived type hierarchies.

CONTENTS

1. Introduction

Declarative languages are a source of great interest to many computer scientists this being partly due to the simple mathematical properties and foundations of these languages. These properties are the expressive power inherent in mathematical formalisms; the potential for manual and automatic manipulation and verification of programs written in the declarative language; and the potential for parallel execution of programs due to the minimal control of flow embodied in the declarative language. This thesis concentrates on the issues of expressivity and efficiency over and above those of verification and parallelism.

Research in the field over the past ten years has split into two directions. The first of these, based on the **functional** language (represented by languages such as LISP [7], Miranda [10], Hope [3] and FP [2, 11]), claims that the use of higher-order functions, lazy evaluation with associated infinite data structures, and deterministic execution provides expressive power, while making it relatively easy to conduct verification and implement parallelism. The second direction, based on **logic** languages (such as Prolog [12]), claims that a goal directed problem solving strategy, along with the use of logical variables and non-deterministic execution yield significantly more elegant solutions to certain important problems, compared to the solutions which can be achieved with purely functional languages. Furthermore, parallelism and verification of programs are also possible with logic languages although there do remain some as yet unresolved problems (see Wise [25], Gregory [23] and Kowalski [36]).

Functional languages are renowned for their simple semantics, while logic languages have tended to have considerably more complex semantics. An important reason for this is that a logic program is, in general, less **specific** than the corresponding functional program. In a functional language all parameters must be *ground*, i.e. contain no unbound variables, and there may not be more than one definition for any particular set of parameters to a function. In languages such as Hope and Miranda this is reflected in the insistence on the principle of *non-superposition*, in which the left hand side of equations defining a function must be disjoint, and in fact may not even contain repetitions of variables. The advantage of these restrictions is that functional languages can be executed **very**

efficiently. The disadvantage is that some algorithms are more naturally expressed when these restrictions are not present. To insist on writing such algorithms in a purely functional fashion leads to obscure and non-declarative solutions. There is thus a trade-off between language efficiency and language expressiveness. The secret of good language design is to find the best balance between these factors, and in this thesis the balance sought for REDLOG is exemplified.

The language that is proposed in this thesis may be thought of as an *equational* programming language in which an equals is substituted for all other equals. This can be seen as a natural generalisation of functional programming, in which an equals is substituted for a necessarily unique equals. The full power of logic programming is incorporated, mainly through the insistence on uniformity during the design process. This leads naturally to the inclusion of variables as first class objects in the language and thus produces the need for unification amongst terms. Unification is the central feature in logic programming. Although the syntax of REDLOG is deliberately made concise, the syntax of the language Prolog forms a naturally occurring subset of REDLOGs syntax, i.e. the *form* of all 'pure' Prolog programs are REDLOG programs! This is obviously of benefit as far as portability is concerned. For teaching, REDLOG is also of value as it is possible to write programs in a purely functional or logical style or in the more powerful combined style. It is thus possible to teach the functional and logic programming paradigms as a subset of the equational paradigm while only using a single language implementation!. The design aim of regularity has resulted in the determination of a well structured, regular domain over which REDLOG operates. This lead to the REDLOG typing system and the resultant type inference and checking mechanism, which form an important part of the language and thesis.

Chapter 2 is a discussion of related work to illustrate the motivation for the thesis and provide a background for comparative purposes. Chapter 3 introduces REDLOG informally with examples and appropriate explanations.

Chapter 4 describes the complete syntax of REDLOG equations, but avoids many other syntactic facets of REDLOG such as lists, numbers, scope and the syntactic aspects of classes and modules. These are covered more completely in the Appendix.

Chapter 5 introduces the important notion of a canonical form for REDLOG and chapter 6 deals informally with the reduction process used in REDLOG. The unusual scope rules of REDLOG are dealt with in chapter 7.

Chapter 8 deals with the REDLOG type universe, type derivations within that universe and the REDLOG type inference and checking system. Chapter 9 provides a formal description, in the style of a denotational semantical specification, of REDLOG equations. Chapter 10 describes a circular- (or meta-) implementation of REDLOG and Chapter 11 discusses a number of issues in the implementation of REDLOG. Lastly, Chapter 12 discusses the conclusions which have been drawn from the REDLOG experiment.

2. Discussion of Related Work

There are numerous recent papers on languages which investigate consolidating functional and logic programming styles. The purpose of this chapter is to set REDLOG in this context. The chapter takes the form of a series of brief language summaries, and then a short discussion. Mention is made of the form of the typing system of each language if such a system exists.

The Equational Programming Language of Hoffman and O'Donnell [4] is equivalent to a restricted form of REDLOG. The restrictions include non-superposition, no repeated variables on the left-hand-side of equations (no concept of failure) and no backtracking. An Equational Programming Language program can be directly translated into a REDLOG program, but a REDLOG program may have to be considerably expanded upon during a translation to the Equational Programming Language. Typing facilities are limited to an ALGOL-like declaration of variable types.

AppLog (Cohen [16]) is a language integration along more traditional lines, as it is simply the union of two existing languages (Prolog and LISP). While gaining the advantage of sophisticated development environments available for one of the languages (LISP), AppLog suffers most from the disparate programming methodologies inherited from its parent languages. AppLog is untyped.

Eqlog (Goguen and Meseguer [18]) is a conceptually comprehensive language which uses a many-sorted logic to define first-order functions as equality axioms, and employs an operational semantics based on *narrowing* (see Reddy [42]). Narrowing is a general method for reasoning about equations, and allows equational inversion. Although this is an appealing feature, it introduces efficiency overheads and restrictions which rule out a delayed evaluation scheme and, in EqLog also prevents the use of negation and higher-order objects. Furthermore, most equations in practice are written with the intention that their ultimate value be dependant on the right-hand side of the equation. EqLog does not have a type inference mechanism, but does have useful facilities for subtyping, allowing the definition of polymorphic operators and, interestingly, coercive overloading.

LEAF (Barbuti, Bellia, Levi and Martelli [15]) is a language with a distinct logical component and a purely functional component. The integration of these two components is achieved by using the functional language as a metalanguage for the logical component (cf. chapter 8). This leads to a number of unwanted restrictions. These restrictions include the principle of non-superposition; the parameters to functional components may not appear in head terms of logical equations; only first order functions are allowed; and calls to logic clauses cannot be made from within a functional clause. In the special case that the inputs to the logic clause call are all in the inputs to the functional clause then this restriction is removed. Although not discussed in their paper [15], Barbuti et al indicate that a polymorphic type system does exist, although no mention is made of a type inference system.

FunLog (Subrahmanyam and You [21]) achieves the integration of logic and functional languages through the notion of "semantic unification" (which is incomplete). This is a potentially costly mechanism because reductions are initiated by repeated attempts at unification (Chapter 11 suggests an alternative way in which the benefits sought by the designers of FunLog may be obtained). FunLog also incorporates (in an informal way) infinite data structures, lazy evaluation and non-determinism. FunLog is untyped.

TabLog (Malachi and Waldinger [20]) uses a restricted version of a deductive-tableau proof system (see Manna and Waldinger [39]). TabLog includes negation, but does not include higher-order functions. The restricted proof system is incomplete. The deductive-tableau proof system represents an interesting design alternative to that described in this thesis, however the execution process of TABLOG is considerably more complex than that of REDLOG and implementations of the language have so far suffered from performance penalties. TabLog is untyped.

Many of these proposals (FunLog, LEAF, EqLog, AppLog) propose that there should be two (or more, see EqLog) distinct subsystems (e.g. functional, logical, equational) in one language. This, it is argued, will allow the programmer to optimise the efficiency of a program through judicious use of the more computationally expensive components. The integration of the subsystems is

achieved with a marked variation in sophistication. That of AppLog is the least satisfactory, the best is probably that of EqLog.

In contrast, REDLOG has only one system, but with the efficiency of a functional system available through a compiler optimisation under certain conditions. It is felt that having a single system is superior to having many, as there can be a free, full-feature, mixing of programming styles without the hindrance of conforming to different rules depending on which system one is using. Functional and logic programming *styles* are inherently part of REDLOG, rather than additions to it.

Another approach has been to incorporate unification (or a subset of it) within a functional framework (see Lindstrom [6], Darlington, Field and Pull [17] and Abramson [1]). However, incorporation of full unification into a functional programming language leads to a non-functional language, with the result that systems such as that proposed by Abramson adopt a restricted one-way unification. It should be remarked that the inclusion of variables as first-class citizens into these languages, as is done in REDLOG, could lead to a programming style which is essentially functional in nature, but with much of the power of full logic programming.

Although issues such as the uniformity of the language vary widely in these proposals, from the point of view of this thesis the most important difference between them is their individual semantics. It is felt that the simplicity of the REDLOG evaluation mechanism proposed below has much to recommend it to programmers. The combination of this simplicity with design regularity and type security should result in a system capable of producing high programmer productivity.

3. An Informal Introduction

The aim of this chapter is to illustrate the expressiveness of REDLOG. Most examples presented are from papers in the literature, to allow easy comparison with other proposals regarding expressivity. All examples have been executed on the test compiler for REDLOG.

3.1. Familiar Examples

First, the factorial program, presented in a functional style:

```
0! ≡ 1.
N! ≡ N * (N - 1)! if N > 0.
```

Note the use of operator notation for "!". "≡" is the REDLOG semantic equality operator (see chapter 6). The form of the program is that of a specification of the functional behavior of the "!" operator, closely paralleling the style used in mathematics. The `if N > 0` conditional term in the second equation is not strictly necessary in a sequential execution context (with left-most selection rule) for REDLOG and the equation is defined to operate on the natural numbers. In LISP the factorial example above would be as follows:

```
(DEFUN (FAC N)
    (COND
        ( (EQ N 0) 1 )
        ( (GT N 0) (PRODUCT N (FAC (DIFFERENCE N 1))) )))
```

It is important to stress at this point that REDLOG is much more than a syntactic "sugaring" of any functional or logic language. It has a fundamentally different semantics which will become apparent in later examples and be shown in later chapters on semantics. For example, the factorial equation above could be used in a non-ground mode by calling it with an unbound variable.

Next, the well known quick-sort algorithm of C.A.R. Hoare (see Tablog [20]). The implementation of this in REDLOG shows a clear mixing of functional and logical

styles:

```
qsort [] ≡ [].
qsort [X|Y] ≡ qsort L + + [X] + + qsort U where partition (X, Y, L, U).

partition(X, [], [], []).
partition(X, [F|Y], [F|L], U) if X > F and partition(X, Y, L, U).
partition(X, [F|Y], L, [F|U]) if X ≤ F and partition(X, Y, L, U).
```

The first two equations provide an apparently functional specification of quick-sort, but note the use of the unbound variables **L** and **U** in the call to partition which is clearly non-functional. This allows the returning of two results from the logically specified partition auxiliary. Partition is specified exactly as it would be in Prolog, apart from syntactic variations for "if" and "and" (the syntactic variations are allowed in REDLOG for standardisation reasons, see the Appendix). Note the use of "where" as a further syntactic variation for "if". The symbol "+ +" is the list append function, again illustrating the use of ōperator notation. As a comparison with a well known language, the Prolog definition of the two "qsort" equations is:

```
qsort([],[]).
qsort([X|Y],Z) :-
        partition(X, Y, L, U),
        qsort(L, LS),
        qsort(U, US),
        append(LS, [X|US], Z).
```

The equivalent LISP or functional program would involve two functions ("lo and "highpart", say) in place of the call to partition, introducing unwanted inefficiency. These could be defined in LISP by the functions below:

```
(DEFUN (LOWPART VAL LIST)
  (COND ((NULL LIST) NIL)
        ((LT VAL (CAR LIST)) (CONS (CAR LIST) (LOWPART VAL (CDR LIST))))
        (T (LOWPART VAL (CDR LIST)))))
```

```
(DEFUN (HIGHPART VAL LIST)
   (COND ((NULL LIST) NIL)
         ((GE VAL (CAR LIST)) (CONS (CAR LIST) (HIGHPART VAL (CDR LIST))))
         (T (HIGHPART VAL (CDR LIST)))))
```

3.2. Higher Order Equations

To illustrate the higher-order capabilities of REDLOG, a functional-style list iterator can be written as follows:

```
map (F, []) ≡ [].
map (F, [X|Y]) ≡ F X : map (F, Y).
```

Notice the first class status of functors in the above equation. An emulatation of this in Prolog would require the use of the meta-logical "call" predicate. This type of programming is also interesting because modules and classes can be passed to equations which can perform operations on them (see chapter 11 section 3). Another example of higher-order capabilities is `reduce` (Henderson [33]), a functional used to capture the notion of using a binary operator over a whole list:

```
reduce ([], F, A) ≡ A.
reduce ([X|L], F, A) ≡ F (X, reduce (L, F, A)).

sum L ≡ reduce (L, +, 0).
product L ≡ reduce (L, *, 1).
```

3.3. Data Structure Manipulation

To illustrate the use of negation and data constructors (as will be described in chapter 8, "data constructors" is an acronym for "named types"), a queue (FIFO structure) can be specified as:

```
empty newq.
~empty add (Item, Queue).
read add (Item, Queue)
                    ≡ Item if empty Queue
                    ≡ read Queue.
delete add (Item, Queue)
                    ≡ newq if empty Queue
                    ≡ add(Item, delete Queue).
```

For comparison purposes, the classic address translation example, as first seen in Warren [49] can be written in REDLOG as follows:

```
translate In ≡ trans (In, Table, 1).
trans ([], _, _) ≡ [].
trans ([def A|In], Table, N) ≡ [asgn(A,N)| trans (In, Table, N+1)] if asgn (A,N)
in Table.
trans ([use A|In], Table, N) ≡ [use Addr| trans (In, Table, N)] if asgn (A,Addr)
in Table.
A in [A|X].
A in [B|X] if A in X.
```

Here we see the extensive use of the logical variable to do automatic backpatching of address labels, and the use of the "don't care" variable. Input to translate will be something similar to [def a, use a, use b, def c, def b], with result [asgn (a,1), use 1, use 3, asgn (c,2), asgn (b,3)].

3.4. Examples from Artificial Intelligence

To exemplify the use of generate-and-test type programs with a mixed functional and logical style, the classic "eight queens" problem can be solved as:

```
eight_queens ≡ q8 (8, []).
q8 (0, Board) ≡ Board.
q8 (Row, Board) ≡ try (q8 (Row - 1, Board), generate (1, 8)).
try (Board, Col) ≡ Col:Board if no_conflict (Board, Col, 1).
```

```
no_conflict ([], Col, Add).
no_conflict ([B|Board], Col, Add)
                    if Col ≠ B and
                    Col + Add ≠ B and
                    Col - Add ≠ B and
                    no_conflict (Board, Col, Add+1).
generate (Lower, Upper)
          { = Lower
            ≡ generate (Lower+1,Upper)
          } if Lower ≤ Upper.
```

The eight queens problem is also solved in Cohen [16]. Notice that, apart from the equations for "no_conflict", the whole program is specified in a functional style. Of course, the semantics are radically different. This is an extremely compact solution to the eight queens problem (try it in LISP or Prolog!), and is also a remarkably lucid explanation of one method of solving it.

In the definition of the generate equation the use of braces to alter the scope of the cond term is depicted (see chapter 7). The possibilities for exploitation of parallelism within such an equation is apparent. This aspect of REDLOG was not available in the test compiler and thus the example executed on that compiler did not make use of the altered scope rule, but was otherwise identical.

A concise solution to the towers of Hanoi problem can be written:

```
hanoi Disks ≡ move (Disks, [] , []).
move ([], M, R) ≡ ([], M, R).
move ([Disk|L], M, R) ≡ move (NewM, NewL, [Disk|NewR])
                    where (NewL, NewR, NewM) = move (L, R, M).
```

This last example, written entirely in a functional style, could be compiled into a very efficient form by a clever compiler for REDLOG using pattern matching instead of unification and simple term rewriting in place of the normal REDLOG reduction mechanism. This does of course depend on all equations being functional in nature (it is more than simply a matter of whether or not a conditional is

used!). It is interesting to compare this example with "quicksort" which makes use of logical variables. Also note that only one result (apart from refinement via unification) is ever returned from a functor call, the result of "move" being a tuple of values (see Chapter 8).

The missionaries and cannibals problem (see Goguen [18]), a simple search problem, is solved by:

```
solve [state (0,0,right) | States] ≡ [state (0,0,right) | States].
solve [S | States] ≡ solve [NM, S | States] if NM = newmove S and not NM in
States.

newmove state (M1,C1,Side) ≡ state (M2,C2,other Side)
        if      C = move M and
                M ⩽ M1 and
                C ⩽ C1 and
                M2 = M1 - M and
                C2 = C1 - C and
                ok (M2,C2).

ok (0,_).
ok (NumM, NumC) if NumM ⩾ NumC

move 0  ≡ 1
        ≡ 2
move 1  ≡ 0
        ≡ 1.
move 2  ≡ 0.

other left ≡ right.
other right ≡ left.

goal solve [state (3,3,left), start].
```

3.5. Typing

Having read this far many readers may have assumed that REDLOG is an untyped language. This is not, however, the case: REDLOG is in fact a polymorphic, strongly typed language in the sense of Milner [41]. REDLOG has a useful notation for defining the type of an equation and also for defining (or *deriving*) new types. The REDLOG compiler also features a mechanism similar in some respects (see chapter 8) to that of Milner [41] for automatically generating type information, although it is still good practice to explicitly include some type information for documentation purposes. As an example, the type of the "solve" equation can be automatically deduced by the REDLOG compiler to be:

```
solve :: [states] →[states].
states :: = state (num, num, side) | start.
side :: = left | right.
```

The type definition for solve can be read: "solve is an equation which transforms a list of states into a list of states". The type of the towers of hanoi solution can similarily be defined as follows:

```
hanoi :: [T] →([T],[T],[T]).
move :: ([T],[T],[T]) →([T],[T],[T]).
```

This type could be further refined by the addition of a goal statement. The typing system of REDLOG and the meaning of the above definitions will be expanded on in chapter 8.

In the examples presented in this chapter there is much that has been tacitly assumed, these facets of REDLOG will be revealed and explored in the subsequent chapters.

4. Equation Syntax

This chapter is intended to act as a reference for the syntactic terminology of REDLOG used in the reading of the rest of the thesis.

The REDLOG alphabet is A = {D, V, F}, where:

D is a set of Data Constructor symbols,

V is a set of Variable symbols and

F is a set of Functor symbols.

A *constant symbol* is a 0-ary data constructor application.

The *canonical order* of REDLOG is the set of data terms.

A data term is:

(a) a constant symbol,

(b) a variable symbol,

(c) a tuple of the for $(t_1,...,t_n)$, where $t_1,...,t_n$ are data terms,

(d) a data constructor application of the form d t, where $d \in D$ and t are data terms.

A *match data term* is:

(a) a data term or

(b) a functor symbol application of the form f t, where $f \in F$ and t are data terms.

A *term* is:

(a) a data term,

(b) a data constructor application of the form d t, where $d \in D$ or $d \in V$ and t are terms.

(c) a tuple of the for $(t_1,...,t_n)$, where $t_1,...,t_n$ are terms,

(d) a functor symbol application of the form f t, where $f \in F$ or $f \in V$ and t are terms.

A *head term* is a match data term.

A *result term* is a term.

A *cond term* is:

(a) a term,

(b) a conjunct of the form **c1 and c2** or, equivalently, **c1,c2**, where c1 and c2 are cond terms or

(c) a disjunct of the form **c1 or c2**, where c1,c2 are cond terms.

An *equation* is:

(a) a fact **A.** where A is a head term,

(b) a functional equation

A ≡ B.

where A is a head term and B is a result term,

(c) a relational equation

A if B. or

A :- B. or

A where B.

where A is a head term and B is a cond term,

(d) a conditional equation

A ≡ B if C. or

A ≡ B :- C. or

A ≡ B where C.,

where A is a head term, B is a result term and C is a cond term,

(e) a conditional equation with alternatives

A ≡ B.

where A is a head term and B is an alternative term or

(f) a conditional equation with alternatives

A if B. or

A :- B. or

A where B.

where A is a head term and B is an alternative term.

An *alternative term* is:

(a) a result or cond term,

(b) **B ≡ A.**

where B and A are alternative terms or

(c) **B if A.** or

B :- A. or

B where A.

where B and A are alternative terms.

Negative equations are deliberately restricted to:

(a) negative facts of the form

~A.

where A is a fact and

(b) negative relational equations of the form

~A if B. or

~A :- B. or

~A where B.

where A is a head term and B is a cond term.

A *program* is a set of equations and a distinguished goal expression.

The complete syntax for modules, classes and types is given in the Appendix.

5. REDLOG Object Representation

This chapter serves as an introduction (albeit at a low-level) to the use of the principle of design regularity which has been adhered to during the construction of REDLOG.

In REDLOG all objects (variables, equations, lists, tuples, numbers, classes, modules and other defined objects) are *first class* in the sense that unification may be attempted between any two objects; all objects may be returned as results of equations and expressions; all objects may be used as substitutions and passed as parameters. This is achievable in the context of general unification (which insists on equality being first-order decidable, as Goldfarb [31] reduced Hilbert's tenth problem to that of the decision problem for second-order unification) by building all objects from an identical representation, which is exactly what the principle of design regularity would indicate! The representation chosen is outlined below.

The representation consists of *names* and ordered *groupings* (or *tuples*) of names. All meanings attached to these and all reductions performed on these are entirely a function of the abstract REDLOG machine as defined in chapters 6, 9 and 10. Note that the fact that variables may be *bound* is not a special property, but merely a function of the unification algorithm and interpretation of the abstract machine. We adopt the following notation for groupings:

() is the *null* grouping,
(G1,...,GN) is a grouping, where N ⩾ 1 and G1,...,GN are names or groupings.

Using groupings, representations for the various objects known to the REDLOG abstract machine can be chosen:

(a) **atoms** are names,
(b) **variables** are groupings of the form (V, VAR) where VAR is an atom representing the variables name,
(c) **terms** are groupings of the form (T, T1,...,TN) where T is a functor or constructor symbol and T1,...,TN are the arguments to the term,

(d) **equations** are groupings of the form (E, HT, RT, CT) where HT is the head term grouping, RT is the result term grouping and CT is the cond term grouping,

(e) **modules** are groupings of the form (M, ModName, Provides, Uses, Eqns) where ModName is the module name, Provides is a grouping of names, Uses is a grouping of names and Eqns is a grouping of equations,

(f) **classes** are groupings of the form (C, ClassName, AKO, Uses, Eqns) where AKO (A-Kind-Of) identifies the type of the class.

All other entities, such as lists, numbers, trees and tuples, can be constructed from the grouping notation. However, for efficiency reasons a particular implementation of REDLOG might choose to implement something like numbers in a form corresponding to an underlying hardware representation.

To conform with the syntax of REDLOG given in chapter 4 and the Appendix, the following syntactic "sugarings" are adopted:

(a) **T Arg** for terms,

(b) **HT ≡ RT :- CT** for equations (see chapter 4 and the Appendix for further syntactical abbreviations and forms),

(c) see the Appendix for the syntax of modules and classes.

6. Informal Operational Semantics of REDLOG

The purpose of this chapter is to provide an English description of the reduction process used in evaluating a REDLOG expression, for those readers who only wish to get an informal understanding of this aspect of REDLOG.

In REDLOG, computation can be viewed as essentially the reduction of an expression to canonical order, and the return of an environment consisting of variable bindings used in obtaining the reduced form. In a functional language (such as Miranda) only the reduced expression is returned, and in a logic programming language (such as pure Prolog) only the environment is returned. The integration of these two result types motivated both the title of the thesis and the choice of the language's name. Languages such as EqLog, FunLog and TabLog fit the model of logic programming languages, computing their reduced expression purely as a side effect of their proof procedures. This side effect is achieved by introducing special cases into the proof procedures for handling functions. In LEAF the opposite occurs: the functional component is the metalanguage for the logic programming component and the result of execution is a reduced term, the execution mechanism thus may have used the separate logic programming component at some stage in the determination of the reduced term. Languages such as AppLog and full LISP have a semantics intimately tied to the large passive store model of the ALGOL type languages (indeed, a recent incarnation of LISP, Scheme[3] [9], acknowledged itself as an algorithmic language in the vein of ALGOL-60). REDLOG's simple integration of logic and functional programming results in a single, uniform execution mechanism with much more pleasing semantics than that of the other languages.

Currently, the usual Church-Rosser property does not hold in its normal form for REDLOG as the reduced expression is not necessarily unique. To insist on uniqueness would have forced the introduction of restraints on expressiveness, such as non-superposition. In a parallel context the reduction process returns sets of results, and this set will indeed be unique, irrespective of evaluation order (the Church-Rosser property). In a sequential context, answers are produced one at a time (possibly through the use of backtracking if depth first evaluation is used) with subsequent answers in the set available on request. Thus, the result of the

process for each answer is a reduced form and a new environment created by the answer substitutions, which were set up by the unification parameter passing mechanism. Note that the Church-Rosser property is still not attained in the sequential implementation as the results are returned as an ordered sequence, and not as an unordered set as is the case in the parallel context. Some sequential implementations may in fact be deterministic, and then their sequentially produced sequences of results will have the property.

Given a goal expression to translate to its canonical order, the REDLOG system will first attempt to unify the expression with a head term of some equation already specified to it. To do this it must rewrite any arguments of the goal expression to canonical order (which is simply a recursive application of the process outlined here). Reducing all arguments of the goal expression may be wasteful of effort if the particular equation unified with the goal expression does not utilise all of its parameters. This could be achieved through a delayed reduction scheme, or a compiler optimisation (see Chapter 11).

Once it has translated the arguments of the goal expression to canonical order, the revised goal expression is **unified** with the head term of an equation in the program. If this is unsuccessful then another equation head must be tried. Assuming success, the bindings set up from the unification are *permanent* (although not necessarily ground) for the duration of the rewrite.

Even though syntactically a fact does not appear to have a result term or a cond term (see chapter 4 for definition), semantically it does, both of them being the constant symbol **TRUE** (a negative fact has the constant symbol **FALSE** as result term). Similarly a functorial equation has a hidden cond term equal to **TRUE** a a relational equation has a hidden result term equal to **TRUE** (a negative relational equation has a hidden result term of **FALSE**). Thus *every* equation has a cond term and a result term, which provides a pleasing uniformity and simplified semantics in REDLOG.

The cond term of the selected equation is then selected for reduction. This will first cause individual conjuncts and disjuncts (if any, there may be just a single term) to be reduced. To do this the terms within the conjuncts and disjuncts are

selected as goal expressions. Each such term can, of course, return any element of the REDLOG *world*. Four cases can be identified: under a sequential semantics the term in a conjunct or disjunct returns

1) the atom **TRUE** and a set of *conditional bindings*.
2) the atom **FALSE**. This indicates that the present goal is invalid and so we return the atom FAIL (for the purposes of this discussion, see Chapter 8 for an alternative representation) as the result of the equation.
3) **FAIL**. This indicates that the present set of bindings are invalid or that an incorrect choice of equation was made. If there are any other equations which can be unified with the goal expression then they are chosen for reduction. If there are no more choices of equations left then the conditional bindings must be re-evaluated and the term retried (i.e. backtracking is initiated). If the conditional bindings cannot be altered (because all other terms return FAIL on backtracking) FAIL is returned as the rewrite of this equation.
4) anything other than the above three atoms. This is an error or *exception* condition, and the user of the REDLOG system will receive appropriate notification. The REDLOG type checking system, if installed, will eliminate the chance of this happening at run-time by producing a compile-time error (see Chapter 8).

Under a parallel semantics each term may return sets of results (in a pipelined fashion) from the above four cases. Given sufficient processing elements, backtracking would be unnecessary.

Once the cond term has returned the value **TRUE** through evaluation of the conjuncts and disjuncts, evaluation of the result term can proceed in the context of the fixed initial bindings and the conditional bindings just selected. The result term is just treated as another goal expression and reduction proceeds as before. The resulting canonical term is returned as the rewrite of the equation, along with the environment derived.

Should this rewrite be rejected, the conditional bindings will then be released and recomputed (probably most efficiently if done in stages). Under the new bindings

the result term is again re-evaluated and if the result is different from any previous rewrite, this is then returned as the final result. Failing that, conditional bindings are again recomputed, until all choices are consumed and FAIL is returned.

Chapter 9 provides a more formal specification of the intended semantics for REDLOG and Chapter 10 contains a simple definition of REDLOG semantics using REDLOG itself as the metalanguage.

7. Scope Rules

The traditional scope rule for functional and logic programming languages has been the current function, functional abstraction or clause (see Henderson [33] and Kowalski [36]). This may prove restrictive in large scale programming as values or environments computed in one part of the program (possibly with considerable effort) may be valid in another. Although a reasonably clever compiler can usually detect this and make the appropriate optimisation, the programmer is still forced either to repeat code segments in the text or, more likely, to worsen the proliferation of the number of parameters that tend to occur in large programs written in functional or logical languages.

To solve this problem a *pre-reduction* environment is proposed. None of the other languages reviewed proposed such a mechanism, and REDLOG only adopts a simple version to be shown below. In order to denote the altered scope rules REDLOG introduces the following constructs to alter the default, equation only, scope rule:

(a) The PostCondition Rule

This takes the syntactical form:

{ <Equations> } <Neck_Symbol> <Cond_Term>

where <Equations> is one or more of <Equation> or further nested scope rules. A restriction is that the environments set up by evaluating the cond term and any unifications of the goal expression and the head term of an equation, must be disjoint.

The intended operational semantics of this construct are, after unification has been successful achieved with one of the head terms, the <Cond_Term> will be reduced to obtain an extended environment. The full form of this scope rule is not included in REDLOG, but only the restricted form below.

The restricted form is that only the scope of *alternative* parts of an equation may be modified:

<Head_Term> { <Alternative_Part> } <Neck_Symbol> <Cond_Term>

The previously mentioned restriction obviously does not apply in this case. An example of the use of this scope has already been seen in chapter 3.

(b) The PreCondition Rule

This rule has not been included in the definition of REDLOG either, but is presented here for completeness. It takes the syntactic form

if <Cond_Term>
    { <Equations> }
{ **elsif** <Cond_Term>
    { <Equations> } }
[**else**
    { <Equations> }]
endif

The operational semantics of this construct is that the <Cond_Term>'s are evaluated in the current environment, with the intention of picking an applicable equation set. The disjoint environments restriction for post-conditions is removed.

8. The REDLOG Type System

The aim of this chapter is to describe and motivate the use of the REDLOG type system and its associated type inference mechanism. The foundation of the work on type inference are the seminal papers by Milner [41] and Hindley [34]. These papers dealt with direct type inference only, and in this thesis the mechanism is extended to allow type inference in the presence of derived type hierarchies.

REDLOG is defined to operate over a single universal domain **U**, subsets of **U** and certain ordered groupings of elements of **U** which are also included in **U** (ordered groupings can also be represented as sets, of course). Even the equations which are presented to the REDLOG compiler are in reality just shorthand for sets of a certain type of ordered grouping of elements of **U**, called a *pair*. The term *mapping* will be used for this set (which is presented to the REDLOG reduction engine, at least conceptually) so as not to cause confusion with the shorthand notation. Background can be obtained from Tennent [46] (Chapter 3), but for completeness and to introduce the notation the basic concepts are included here. Of course, the structure of **U** is particular to REDLOG and therefore the section on basic concepts should be read.

The particular **U** chosen is a pragmatic issue. For instance, on a machine which represents integers in 16 bits one may chose to exclude all integers other than-32768 to 32767 from **U** and in a version of REDLOG to be used in Physics, the complex numbers may be included. However, certain sets of objects are required to be present in **U** by the REDLOG engine. These are the variables, the booleans and the empty set.

The concepts of the REDLOG type system are built up in this chapter in sections 8.1 and 8.2. Section 8.3 contains an extract from the Appendix, listing the syntax rules for type derivation. Section 8.4 describes the REDLOG type inference system and section 8.5 closes the chapter with a discussion of a potential development methodology which makes use of the REDLOG type system.

8.1. Basic Concepts

Definition

A *type* is a subset of U.

Definition

A *tuple* is an ordered grouping whose elements may belong to different types.

The notation

$$(t_0,...,t_n)$$

is used to denote a tuple.

Definition

A *pair* is a tuple with exactly two elements.

Definition

A *map* is a pair whose first element belongs to the *argument* type of the mapping and whose second element is a set of elements of the *result* type of the mapping.

Note: if the second element (the *result* element) is the empty set then the REDLOG reduction engine will interpret this as implying failure.

Definition

A *mapping* is a set of maps belonging to the same type such that the first element of all maps within the mapping are different.

Note: a mapping is itself a type.

Definition

A *typing* is a set of mappings and a type.

Note: this is commonly known as an *abstract data type*.

8.2. Type Variables and Functionality

Definition

A *type variable* is a symbol used to represent an element belonging to any type.

Type variables are a useful means of providing for generic or polymorphic functional abstractions in a programming language and are an important feature of the modern functional languages ML [8] and Miranda [10]. In the simplest scheme (as used in ML and Miranda), a type variable (written T_i in this chapter, $i \geq 0$) denotes any type as indicated by the above definition. Later, it will be seen that this notion of a type variable has to be complicated if it is to support the full power of the REDLOG type system. Type variables are used to classify mappings with a similar type structure together. The term *functionality* will be used for this classification. As an example, construct **U** using **N** (the natural numbers)as a basis, then in REDLOG the functionality

$$T_0 \rightarrow T_0$$

(where " $\rightarrow$ " is the symbol denoting a mapping. Note that this symbol is simply a syntactic sugaring for a mapping, i.e. a set of maps of the same type, but with different first elements) represents the type

{(0,{}), (0,{0}), (0,{1}), ... , (0,{n}), ...
(0,{0,0}), ...
...
(1,{}), (1,{0}), ...
...
((0),{}), ((0)),{0}), ...

...

((n,m,...),{}), ...

... },

the representation being a little simpler in a purely functional language. Note the infix use of the symbol " → ". In common with the whole of REDLOG, type constructors are defined to take exactly one argument, the use of an infix symbol being allowed for a pair as a special case.

The REDLOG equation

$$\text{succ n} \equiv \text{n} + 1$$

which has the functionality $T_0 \rightarrow T_0$, but has the type

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

as its mapping. The functionality of an equation can thus be seen as a generalisation or *abstraction* of the semantic information contained by the equation and its mapping (of course, an equation and its mapping represent the same information although expressed in a different form).

8.2.1. Other Type Constructors

Apart from pairs and mappings there are a number of other useful type constructors provided in REDLOG:

Definition

A *union* of two types T_0 and T_1 is a type consisting of all the elements of T_0 and T_1.

The infix symbol "|" is used to represent the union of two sets.

Definition

A *list* of type T is an ordered grouping whose elements all have the same type T.

A list is thus a singly typed tuple, and the notation:

$$[l_1,...,l_n]$$

is chosen to distinguish it from a normal tuple.

Definition

A *record* is a named type.

The notation

$$rn\ T$$

is used to denote a named type, where rn is the record name. This can be seen as a generalisation of the constructor of languages such as Prolog [12] or Miranda. The naming of a "null-type" is also allowed. A null-type is a syntactic sugaring for a type variable which ranges over **U**. The syntactic sugaring is to simply only write down the record name. Such objects correspond to the "atom" of languages like Prolog and LISP [7].

A suitable representation for a named type might be a pair with first element being the name and the second element the type. The empty set would then represent the null-type.

The generality of named types is such that it forms a convenient structure with which to represent all of REDLOG! For example, the equation:

$$succ\ n \equiv n + 1.$$

can be represented as the named type:

$$\equiv (\text{succ n}, + (n, 1))$$

whose type is then a named tuple of a named variable, whose type is a number, and a named tuple of a variable, whose type is a number, and a number.

Definition

A *restriction* of a type T is all elements of T which satisfy a specified predicate.

The notation

$$T \upharpoonright C$$

is used to denote a restriction.

Definition

A type is *derived* from one or more other types when it is built using the above defined type constructors.

The notation

$$T_0 ::= T_E$$

is used to denote the derivation of type T_0 from the type expression T_E.

Definition

A *type declaration* is the fixing of a names type, where the type is built using the above defined type constructors.

The notation

$$I :: T_E$$

is used to denote the declaration of the type of the name I.

8.2.2. Examples

To give an idea of the flavour of type derivations a few small examples are presented. A simple basic subtype hierarchy can be specified as:

atom :: = num | bool | char.

A string is a list of characters:

string :: = [char].

The special notation "$c_1...c_n$" is adopted for strings.

A tree can easily be derived if use is made of a recursive type derivation:

tree T :: = left tree T | leaf T | right tree T.

The non-zero naturals can be specified using a restriction:

nznat :: = n :: num ⌈ n > 0 and integer n.

In fact the list type constructor is not really needed as it may be defined as:

list T :: = nil | T cons list T.

although the appearance of the syntax for lists is then somewhat different from that previously defined. Note the use of the infix cons, this being a syntactic sugaring for the named pair

cons (T, list T).

An example of type declarations has already been seen in Chapter 3.

8.3. Abstract EBNF Syntax Definition of Type Derivations and Declarations

derivation :: = type-heading :: = type-constructor

declaration :: = name :: type-constructor

type-heading :: = name | parameterised-name

paramaterised-name :: = name parameter

parameter :: = variable
| basic-type
| type-constructor

type-constructor :: = union
| restriction
| tuple
| list
| mapping
| basic-type
| derived-type

8.4. Type Inference

It is the authors view that type inference and type checking are useful pragmatic features of any programming language. Type checking is useful for the additional security that results from the imposed discipline associated with it and for the improved programming methodology which arises from programming in the presence of a typed universe. Type inference is useful as it allows the programmer to chose when to avoid cluttering his program with obvious type information, this being left to be deduced by the machine although the resultant type-checking is no less strict. Explicit type information can be useful documentation though, and this point is returned to in section 8.5.

A type inference mechanism is thus defined to be part of REDLOG. Section 8.4.1

gives the type rules used by the type inference mechanism to both infer and check the types of the equations and type derivations of the system. Section 8.4.2 then outlines the actual implementation of the type inference system (written in REDLOG).

The notation "::" used for type declarations can be read as "is of type". The notation used to write the rules is similar to that of Cardelli and Wegner [29]. The part of the rule above the line is the condition and the part below the line is the inference which can be made when the condition is satisfied.

8.4.1. Type Inference Rules

8.4.1.1. Direct Type Inference

1. *Numbers*

$$\frac{N \in \mathbb{R}}{N :: num}$$

2. *Truth Values*

$$\frac{B \in \mathbb{B}}{B :: bool}$$

3. *Characters*

$$\frac{C \in \mathbb{C}}{C :: char}$$

4. *Variables*

$$\frac{v \in \mathbb{V}, t \in \mathbb{T}}{v :: T}$$

Note: **T** is the set of all types (i.e. the powerset of **U**) and **V** is the set of all

variables in U.

5. *Lists*

$\underline{a :: T_1, t :: T_1}$
[a|t] :: [T1]

Note: the principle of *type extension* ensures that the type of a list will be the type list of the least upper bound type of the elements of the list, where a least upper bound exists.

6. *Tuples*

$\underline{a :: T_1, t :: T_2}$
$(a,t) :: T_1 \times T_2$

7. *Equations*

$\underline{f :: T_1 \rightarrow T_2, a :: T_1, b :: T_2}$
$f\ a \equiv b :: T_1 \rightarrow T_2$

8. *Equation Body*

$\underline{c :: bool, r :: T_1}$
$r\ \text{if}\ c :: T_1$

9. *Equation Body with Alternatives*

$\underline{c :: bool, r :: T_1, a :: T_1}$
$r\ \text{if}\ c \equiv a :: T_1$

Note: the principle of *type extension* is important in determining that the type of an equation body with alternatives is the least upper bound of the types of the individual alternatives, assuming that a type hierarchy does exist amongst the types of the alternatives.

10. *Application*

$$\frac{f :: T_1 \rightarrow T_2,\ a :: T_1}{f\ a :: T_2}$$

Note: the principle of *type extension* provides an elegant form of *application* *type extension* to the rule of *application* presented here.

8.4.1.2. Type Inference in the Presence of a Derived Type Hierarchy

11. *Type Extension*

$$\frac{a :: T_1,\ T_1 \leq T_2}{a :: T_2}$$

12. *Type Union*

$$\frac{T_1 \leq T_2,\ T_3 \leq T_2}{T_2 ::= T_1 \mid T_3}$$

13. *Type Restriction*

$$\frac{T_1 \leq T_2,\ c :: bool}{T_1 ::= T_2\ \hat{}\ c}$$

8.4.2. Implementation

The implementation of the type inference system has been completed in Arity Prolog v4.0 for efficiency reasons (a very fast compiler is available for this system), but is presented here in REDLOG for clarity of exposition.

The translation to REDLOG bears a marked resemblance to the rules above. The peripheral details of the system, such as the driver loop are not presented.

The implementation keeps an environment as its work space from which it infers

new facts which are then incorporated into the environment. Extensive use is made of the logical variable to achieve this.

Each rule is defined as a functional of two arguments, the current environment and the program fragment whose type is currently being inferred. The result of the functional is a new environment (of course the actual environment of the REDLOG Reduction Machine will have been refined in the process, these two should not be confused).

To begin with, a simple rule, rule 2 in section 8.4.1.1, is presented as translated to REDLOG:

$$Env \vdash B :: bool \equiv Env \text{ if boolean } B.$$

This may be read "B can be deduced to be of type bool from Env with result ⊥ if B is a member of the set of booleans". Apart from the complications introduced by the environment, the resemblance of the REDLOG rule and rule 2 is obvious.

The translation of the other rules proceeds in a similar manner. The rule for lists produces two REDLOG rules:

$$Env \vdash [] :: [T] \equiv Env.$$
$$Env \vdash [A|L] :: [T] \equiv (Env \vdash A :: T) \vdash L :: [T].$$

The second of these rules may be read "[A|L] can be deduced to be of type [T] from Env with result new Env if it can be deduced from the environment that A is of type T and L is of type [T]". A translation using a single rule is possible by utilising an alternative.

The REDLOG rule for equations is:

$$Env \vdash (N\ A \equiv B) :: (AT \rightarrow BT)$$
$$\equiv\ Env \vdash N :: (AT \rightarrow BT) \vdash A :: AT \vdash B :: BT.$$

This can be read "N A ≡ B is of type AT → BT if N is of type AT → BT and

A is of type AT and B is of type BT" if the Env argument is ignored.

To illustrate a possible method for producing an error message, the rule for application can be augmented with the following functor:

on_error :: ((EnvType,Typing)→EnvType, [string])→EnvType
Ok on_error Message ≡ Ok ≡ err_output Message.

where err_output is a functor which transmits a list of strings to a terminal and returns a null environment. The rule for application can then be defined as:

Env ⊢ F A :: T ≡ Env ⊢ F :: (AT → T) ⊢A :: AT
on_error
[to_string F, " must be a functor and ", to_string A, " must be a compatible argument"].

The translation of the other rules follow a similar pattern.

8.4.2.1. Type Inference in the Presence of Derived Type Hierarchies

As was first noted in Collier [30], the introduction of type hierarchies introduces extra complexity into the type inference system. Since a type variable may now represent a number of types in a hierarchy and it is desired to always find the strictest type, it is necessary to alter the notion of type variable to incorporate dependencies as was done by Collier. One method of dealing with this in the actual type inference mechanism would be to constantly take the intersection of the types associated with a variable and the types expected by the functor to which the variable is an argument. Should this intersection produce the null set for the variable then it would be necessary to see if the principle of type extension could not produce a resolvent of the clash of types. If not then the program is not well typed. This varies slightly from the method of Collier in that not all derived types are carried around, in an effort to reduce the already large time costs involved.

An alternative method would be to allow a type variable to only ever possess one

value, this being the type which is the most general of those encountered for that type variable in the particular type hierarchy. Should a functor demand a stricter type for the type variable then this type is selected. This can safely be done since all functors which accept a type which is a superset of the stricter type will obviously accept the subset type as argument. To summarise, if a clash of types is found for a type variable then instantiate that type variable to the common ancestor of the types causing the clash, if the ancestor exists. If the ancestor does not then the program is not well typed.

The above method will be called the law of *type variable resolution*. It works in partnership with the law of *composition resolution*. This can be summarised as follows: if a clash arises between the type of the argument of a function and the type expected by the function then determine whether one of the clash types is a direct ancestor of the other. If so then instantiate the smaller type to the ancestor, else the program is not well-typed.

It is felt that this method may offer a solution to the combinatorial time costs incurred by the previous method due to Collier. However, this has not been formally verified.

8.4.3. An Example of Type Inference

As an illustration of how the implementation of direct type inference works, a short example is presented here.

The type of the quick-sort algorithm shown in chapter 3 can be inferred by the following stages, presuming that there are standard definitions for the operators:

```
++      ::      ([T],[T])→[T].
>       ::      (num,num)→bool.
<       ::      (num,num)→bool.
```

Using the first quick-sort equation and rules 7 and 5 the type of quick-sort is inferred to be:

```
qsort :: [T]→[T].
```

Using the second equation of quick-sort, the type of "++" is used as a check to insure that the result type of quick-sort is a list of elements of some type; the rule for applications (10) also demands that the argument type is compatible and so the types of the L and U variables is sought, leading to the demand of the type of the partition equation (the result type of this is demanded by rule 8 in checking that it is boolean); by a repetition of this process and using all three equations for partition and the number based relational operators > and ≤, the type of partition is determined to be:

```
partition :: (num,[num],[num],[num])→bool.
```

which agrees with expectations so far and results in the type:

```
qsort :: [num]→[num].
```

for quick-sort, as expected with number based relational operators.

8.5. A Method for Type Specification

As has already been mentioned, the notation used for specifying the types of equations and variables and the notation for deriving new types forms a useful part of the documentation of a program (an aside: many people prefer the term *script* to that of *program* for a declarative language solution to a problem, so as to avoid any suggestion of proceduralism). A suggestion for taking this further would be to employ the method of *type based specification* as an early part of the development cycle.

To perform a type based specification of a problem the programmer examines the problem from the topmost level and regards its solution as being conducted by a "black box". The shape of the input to and output from this "black box" is then the transformation (if any) of types involved in solving the problem. This can be specified using the mapping type operator. The shape of the input and output (their types) may then be derived. Next, possible intermediate stages of type transformation may be specified and "black boxes" or mappings defined for them. This process is then continued until the complete type structure of the program is

defined, along with all the mappings to be performed. The final stage is then to compile this type specification, relying on the type inference and checking component of the compiler to verify the type specification. This process will probably involve several iterations.

All that remains is to implement the "black boxes", a series of equations. The advantage of the type based specification method is the abstracted view of the solution to a problem enjoyed by the programmer. This encourages the programmer to critically examine the data structures used in solving the problem. This issue of *representation* is acknowledged by many researchers to be the most important in Artificial Intelligence, and arguably so in many other fields. The breaking down of the transformations into intermediate steps also results in a favorably abstracted approach to *top-down specification*.

9. Formal Semantics of REDLOG

A continuation-based denotational semantics in the style of Tennent [47], Schmidt [44] and Stoy [45] is presented. None of the languages of chapter 2 provide a formal definition in the style of denotational semantics. Where there are differences in notation, the notation of Schmidt will be adopted. These differences are purely syntactic in nature and in no way affect the semantics of the definition. The semantics of types and type derivations are omitted from the semantics presented here, see Chapter 8 for a discussion of their meanings.

9.1. Abstract Syntax

S	$\in$	Scripts
E	$\in$	Equations
T	$\in$	Terms
H	$\in$	Head_Terms
R	$\in$	Records
Te	$\in$	Type_Expression
C	$\in$	Conditional_Terms
V	$\in$	Variables
Id	$\in$	Identifiers
N	$\in$	Numerals
B	$\in$	Booleans
Ch	$\in$	Chars

S	::=	E goal H
E	::=	E_1 E_2 | H {[≡T] [if C]}.
T	::=	N | B | Ch | [T] | (T_1,T_2) | R T
H	::=	Id T
C	::=	T and C | T or C | not C | T

9.2. Semantic Algebras

I. Script Outputs

Domain

Answer = *Denotable_Value* x *Subst*

II. Denotable Values

Domain

v∈ *Denotable_Value* = *Int* + *Bool* + *Char* +
(*Denotable_Value*→*Denotable_Value*) +
(*Denotable_Value* x *Denotable_Value*) +
(*Denotable_Value* + *Denotable_Value*) +
Denotable_Value List +
Identifier

III. The Truth Values

Domain

Bool∈ 𝔹

Operations

true,
false : *Bool*

not : *Bool*→*Bool*

and,
or : *Bool* x *Bool*→*Bool*

(_→_□_) : *Bool* x D_1 x D_1→D_2

IV. The Integers

Domain

Int∈ ℤ

Operations

..., minus one, zero, one, ... : *Int*

plus : *Int* x *Int*→*Int*

minus: *Int* x *Int*→*Int*

times: *Int* x *Int*→*Int*

div : *Int* x *NzInt*→*Int*

equals,

lessthan,

lessthanequal,

greaterthan,

greaterthanequal: *Int* x *Int*→*Bool*

V. The Characters

Domain

Char∈ ℂ

VI. The Lists

Domain

List = *Nil* + (*D* cons *List*)

Operations

Nil : *Unit*

cons : *D* x *List*→*List* (written infix)

append : *List* x *List*→*List* (written infix)

VII. Environments

Domain

s∈ *Subst* = *Identifier*→*Denotable_Value*

Operations

emptysubst : *Subst*

emptysubst = λi.i

accesssubst : *Identifier*→*Subst*→*Denotable_Value*

accesssubst = λiλs.s(i)

updatesubst : *Identifier*→*Denotable_value*→*Subst*→*Subst*

updatesubst = λiλvλs.[i T v] s

VIII. The One Element Algebra

Domain

Unit (only a single element in the domain)

Operations

() : *Unit*

IX. Continuations, REDLOG's Control Algebra

Domain

c∈ *Cmd_Cont* = *Subst*→*Answer*

sc∈ *Success_Cont* = *Failure_Cont*→*Cmd_Cont*

fc∈ *Failure_Cont* = *Cmd_Cont*

Operations

succeeded : *Success_Cont*

succeeded = λfc.λs. in(Store, s)

failed : *Failure_Cont*

failed = λs. in(String, "failure")

X. Evaluation Strategies

Domain

Strategy = *Success_Cont*→*Failure_Cont*→*Cmd_Cont*

XI. Identifiers

Domain

Identifier = *Id* + (*Id, Denotable_Value List*)

Operations

MGU : *Identifier*→*Identifier*→*Subst*→(*Bool* x *Subst*)

9.3. Valuation Functions

S:Script→*Answer*

S⟦E goal T⟧ =

let (c, s) = T⟦T⟧ (E⟦E⟧ emptysubst) succeeded failed

in (c s, s)

E:Equation→*Subst*→*Subst*

E⟦E_1 E_2⟧ =

λ s. (E ⟦E_1⟧ ○ E ⟦E_2⟧) s

E⟦H≡T if C⟧ = λs.

updatesubst H⟦H⟧ O⟦T if C⟧ s

O:Eqn_Body→(*Strategy*→*Subst*→*Cmd_Cont*)

O⟦T if C⟧ =

λscλfcλs.

let (tr, s', c) = C⟦C⟧sc fc s in

tr→T⟦T⟧s □c s

H:Head_Term→*Identifier*

H⟦Id T⟧ = (Id, T)

H⟦Id⟧ = Id

C:Conditional_Term→*Strategy*→*Subst*→(*Bool* x *Subst* x *Cmd_Cont*)

C⟦T and C⟧ =

λscλfcλs.

let (tr, s', c) = C⟦T⟧sc fc s in

tr→let (tr', s", c') = C⟦C⟧c s' in

tr'→(true, s", c')□c' s"□fc s'

C⟦T or C⟧ =

λscλfcλs.

let (tr, s', c') = C⟦T⟧sc fc s in

tr→(true, s', c')

□let (tr, s", c") = C⟦C⟧c s in

tr→(true, s", c")□fc s

C⟦not C⟧ =

λscλfcλs.

let (tr, s', c') = C⟦C⟧sc fc s in

tr→fc s□(true, s', c')

C⟦T⟧ = λscλfcλs.

let (c', s') = (T⟦T⟧s) s in

c' s→(true, s', sc)□fc s

T:Term →*Subst* →*Cmd_Cont* →*Cmd_Cont* →*(Cmd_Cont x Subst)*

T⟦N⟧ = N⟦N⟧

T⟦B⟧ = B⟦B⟧

T⟦Ch⟧ = Ch⟦Ch⟧

T⟦$[T_1,...,T_n]$⟧ =

λs. let (c',s') =

collectlist T⟦T_1⟧s ... T⟦T_n⟧s

T⟦$(T_1,...,T_n)$⟧ =

λs. let (c',s') =

collecttuple T⟦T_1⟧s,..., T⟦T_n⟧s

T ⟦FT⟧ = λ sλ c. FT ⟦FT ⟧ s c

FT:Functor_Term→*Subst*→*Cmd_Cont*→*Cmd_Cont*

→*(Cmd_Cont x Subst)*

FT ⟦Id⟧ = λ sλ fcλ sc .

let (tr, s') = MGU Id s in tr→(sc, s') □fc s

FT⟦V⟧ = λs. accesssubst V s

FT ⟦FT T⟧ = λ sλ c. FT ⟦ FT ⟧ s c T⟦T⟧

9.4 Shortcomings of the Definition

As already noted, the semantics do not cater for type declarations or derivations. Chapter 8 provides adequate coverage of these concepts. A better semantic definition might be to split the environment into two: a *function environment* and a *substitution environment*. The advantage of doing this is expositional, although it also has relevance to an implementor of REDLOG: should the implementation employ a split environment? Splitting the environment could result in performance improvements. Chapter 10 contains a circular semantics in which a split environment is used.

10. Implementing REDLOG in REDLOG

An outline implementation of REDLOG is conducted in REDLOG in this chapter as a companion to Chapter 9, the denotational semantics of REDLOG. This is also done to continue the tradition started by the 'LISP in LISP' interpreter [7], which is considered to be a useful way of judging the complexity of a programming language by some researchers. The interpreter is a less formal semantic specification of REDLOG and uses a split environment. None of the languages of chapter 2 provide meta-implementations.

The implementation is developed as a series of rules which model the semantic meaning denoted by a REDLOG equation. The method of type specification propounded in Chapter 8 section 5 of this thesis is used in section 10.1. The rules themselves are then implemented in section 10.2.

10.1. Type Specification of REDLOG

The outer level of interpretation is represented by

```
interp :: (syntax,defns,subst)→resulttype.

resulttype :: = (denotable_value,subst) | defns.
```

The symbol "interp" represents the "black box" which performs the transformation of the syntactic form of REDLOG to a "denotable_value" and its environment using the definitions for equations and types in "defns" and an initial environment (usually empty). "defns" is assumed to contain all primitive operations. The option to return a "defns" type will become clear in the next section.

The next stage indicated in Chapter 8 is to define the input and output type domains. The syntactic domain is considered to be a list of equations or type derivations:

```
syntax :: = [equation | type_derivation].
```

To simplify the exposition the type domain for equation is chosen to be the set of REDLOG equations! This is in fact the most natural method, and is commonly used in Prolog programs (where the built in predicate "read" is used to read in terms, see Clocksin and Mellish [12]). A similar choice is made for type derivations.

The domain of denotable values in REDLOG is:

denotable_value ::= num | bool | char | variable | name | name T | [T] | (T_1,T_2) | $T_1 \rightarrow T_2$ | T ⊢ C.

This is of course a specification for the standard universal domain of REDLOG, although a superset can be chosen if desired, as was intimated at the beginning of Chapter 8.

The domain of substitutions is:

subst ::= [subst].

subst ::= (name, denotable_value) | (variable, denotable_value).

According to Chapter 8 the next stage is to specify the types of the intermediate structures used to solve the problem. The problem of implementing REDLOG in REDLOG is in fact a trivial one and so there are no intermediate forms: the translation is direct!

10.2. Implementing the Rules

Since there only emerges one mapping from the type specification of the problem. it is only necessary to implement this to complete our translator!

The rules are as follows:

interp (N,D,S) ≡ (N,S) if number N.

interp (B,D,S) ≡ (B,S) if boolean B.

```
interp (C,D,S) ≡ (C,S) if char C.

interp (V,D,S) ≡ (V,S) if variable V.

interp (I,D,S) ≡ (I,S) if identifier I.

interp ([],D,S) ≡ ([],S).

interp ([R|Rest],D,S) ≡ ([RI|RestI],S'')
        where (RI,S') = interp (R,D,S) and (RestI,S'') = interp (Rest,D,S').

interp ((R,Rest),D,S) ≡ ((RI,RestI),S'')
        where (RI,S') = interp (R,D,S) and (RestI,S'') = interp (Rest,D,S').

interp (F A,D,S) ≡ (FI AI,S'')
        where (FI,S') = interp (F,D,S) and (AI,S'') = interp (A,D,S').
```

interp (E_1.,D,S) ≡ interpdef (E_1,D).

interp (E_1. E_2,D,S) ≡ interp(E_2,interpdef (E_1,D),S) if equation E_1.

interp (E goal G,D,S) ≡ interp (G,interp (E,D,S),S).

interpdef :: (equation,defns)→defns.
interpdef ((F A ≡ B),D) ≡ [(F A ≡B) | D].

The astute reader familiar with Prolog may notice the use of the expression "variable V" in one of the rules above, and may wonder if this is a "metalogical" facility being introduced via the back-door as is done with the "var" predicate in Prolog. This is not in fact the case, as variables in REDLOG are truly first-class. Variables are legitimate members of the universal domain and thus a type recogniser may be defined for variables.

Note how the interpreter returns the current state of the environment if it is not

asked to perform any reductions. As can be seen from the above definition of REDLOG in REDLOG, circular- (or meta-) implementations can be very compact when a sufficiently powerful language is used to implement itself. The same limitation in not including updating of type declarations and derivations as the denotational semantics is present here. Note the splitting of the environments as suggested in the previous chapter.

It would be an interesting experiment to formalise REDLOG sufficiently that it could be substituted as the core language of the denotational semantics method used in the previous chapters. The advantage of integrating the two would probably lie in more compact semantic definitions.

11. Other Areas of Language Design Interest

In this chapter a number of areas of design which have not been explicitly mentioned and which were not of direct interest to the main areas of the thesis are discussed.

11.1. Reduction Order

A point glossed over in previous chapters is REDLOG's reduction order so this is clarified here. The work of Burton [28] and Lindstrom [6] have influenced the methodology selected for REDLOG.

Unlike Miranda [10] whose default reduction order is call-by-name, REDLOG has call-by-value as its default reduction order. The advantage of call-by-name is that it will always produce termination when call-by-value terminates, but termination of call-by-name does not imply termination of call-by-value. Overcomputation is often also avoided with call-by-name and a useful programming style which makes use of infinite data structures is efficiently realisable with the call-by-name reduction order. The advantage of call-by-value is chiefly that programs will often require an order of magnitude less space in which to run, and garbage collection can be facilitated as structures used to hold waiting-to-be-reduced structures under call-by-name can be reduced immediately and then unused nodes garbage collected. Another reason for the selection of call-by-value for REDLOG is the demand of the unification procedure for reduced arguments so that equation selection can be performed.

In the spirit of REDLOG, a combination of these advantages is sought. After Lindstrom [6], strictness analysis is to be used by a future REDLOG compiler to determine exactly when an argument need not be reduced. This relies on knowledge not only of the subsequent use of the argument (as in Lindstrom), but also of the needs of the unification mechanism in determining which equation(s) should be selected. This will allow the use of infinite constructs in the functional style.

It should be noted that the use of infinite constructs are not entirely ruled out

by the current REDLOG system. Use of the logical variable can be made to stand for infinite constructs, using unification to generate new logical variables as the structure is refined.

11.2. The Unification Algorithm

The actual unification algorithm to be used is worthy of consideration when implementing REDLOG. A good candidate is Martelli and Montanari [40] or some conceptual simplification thereof. One idea is to allow unification loops (by removing the occur check), and to regard the result as a regular infinite data structure. This would considerably extend the power of REDLOG, while possibly further enhancing performance.

11.3. The MetaLanguage and Tools

The programming environment is increasingly being recognised as more than just a combination of programming language implementation, operating system and editor (see Good [32]). Development tools are seen as an important aid to productivity by many people from industry and academia, and range from simple pretty printers and module managers to sophisticated correctness proving systems (Good [32]). It is therefore important to establish some sort of metalanguage to allow the creation of system tools which will manipulate the source program code in a flexible and natural manner.

The introduction of a second, different, language to act as the metalanguage of a programming environment, is obviously a complicating factor and so the primary development language is usually chosen as the metalanguage (as in C/UNIX [27]). However, this is often an inappropriate choice if the primary language is not particularly expressive (a simple C compiler written in C would involve several thousand lines of source code!). With REDLOG this is not the case. The uniform object representation of REDLOG and resultant first class status for *all* language objects makes REDLOG especially suited for use as a metalanguage (cf. LEAF [15], in which a single distinguished component is selected as the metalanguage, thus restricting the power of the system). A REDLOG in REDLOG compiler is an order of magnitude shorter than a C in C compiler! (the assumption being that both

languages are being compiled to architectures designed for the support of the particular language.)

The provision of powerful abstraction facilities in REDLOG, such as the class and module construct and the hierarchical nature of the language itself, allows flexible "programming in the large", and encompasses "object-orientated" programming.

11.4 Existing Implementation

An early compiler which translated a small subset of REDLOG into standard (DEC-10) Prolog was developed in standard Prolog on an IBM-PC AT. This has been extended with a powerful new front-end which translates substantially more of REDLOG. All the examples in this thesis have been executed.

The front end of the compiler is a three stage mechanism involving parsing, type checking and type inference and then conversion to a form suitable for processing by the earlier compiler. The parsing was achieved using the Definite clause Grammar technique described in Clocksin and Mellish [12]. The typing system was implemented in a very similar manner to that described in chapter 8 section 4.2, although REDLOG was not used in the implementation for the reason mentioned at the beginning of that section.

The final compilation to Prolog has much scope for improvement. The aim of the current system was simply to obtain a working compiler in the shortest time possible. No advantage was taken of potential optimisations in the form of the REDLOG equations. One of the ideas could be employed is a means for attaining improved efficiency through the use of the one way nature of the result term of a REDLOG equation. Prolog is a useful target code for a trial implementation such as this one is, but a target code of a lower-level nature might be more useful if a production compiler were being implemented. A more sophisticated compiler of this nature, if written in Prolog, would probably be of greater length than the current compiler (which is about two thousand lines).

11.5 Parallel Languages

An important issue in future implementations of languages such as REDLOG is their ability to be supported on a parallel process interpretation architecture. This is an area which has been investigated by a number of researchers, the three most prominent of which are Ehud Shapiro and his language Concurrent Prolog [24], Steve Gregory and his language PARLOG [23] and Michael Wise and his language and architecture EPILOG [25]. The architecture of EPILOG is reviewed in Wright [26].

All three languages have a number of common characteristics. The most obvious of these is some form of "guard" command which is used as an additional constraint on equation selection. REDLOG's conditional term can be seen to fit this requirement in a remarkably natural way, as opposed to the somewhat artificial and operationally orientated approach of the other three languages which propose that the guard be read as a special kind of conjunction.

The rest of the clause is usually placed after the guard command in these languages. REDLOG's functional result term is suitable for this purpose, and the explicit flow of information embodied in the result term is of particular importance. Instead of using unsightly annotations to the variables in these clauses, as is done in the three parallel languages, use can be made of the explicit relationship defined amongst the elements of a REDLOG result term to efficiently obtain this information. This area is worthy of extensive investigation (and another thesis!) on its own.

12. Conclusion

The REDLOG experiment has successfully created a language in which a free mixing of programming language styles may be employed by a programmer in the solution to a problem, while at the same time remaining a language with a single execution mechanism. REDLOG is a new approach to programming, rather than a patched logic programming or functional programming language. It is believed that the resultant conceptual simplicity of the language is the way to achieve a major gain in programmer productivity.

Compared to much of the related work, REDLOG's method of integrating logic and functional programming seems most natural. With any mechanism or algorithm simplicity often results in greater generality and power and thus the REDLOG approach seems promising. The syntax and semantics have been described both formally and informally, which is of some importance for an implementor of the language.

The uniformity of REDLOG, in which all objects, from variables to classes and modules, are first-class, is also an important factor in gaining expressive power. This regularity should both reduce the learning curve involved in acquiring a thorough knowledge of using REDLOG, and reduce the size of programs written in REDLOG. Another important aspect of REDLOGs uniformity is its type system which can be used to represent the whole of REDLOG in a concise and regular way.

The type inference mechanism, and its method for dealing with derived type hierarchies, is an essential tool for program development. The method for deriving new types is pleasingly familiar, and the ability of the type inference mechanism to efficiently perform in the presence of such derived type hierarchies is important.

Throughout the design of REDLOG a careful balance between expressive power and efficiency was striven for, leading to a language which can be used for disciplined specifications, and software production.

12.1. Future Directions

An interesting project would be to develop a complete programming environment written entirely in REDLOG. This would involve the development of an efficient compiler for full REDLOG (in a bootstrapping fashion), before continuing with the development of the rest of the environment. This efficient compiler would use the front-end mentioned in the previous section, and would then use a new code generation phase. This could be broken down into two stages, firstly an intermediate code which would reflect a suitable architecture for REDLOG, and then a final phase of code generation onto a conventional architecture.

Another area currently being examined is a process interpretation for REDLOG. It is hoped that such a radical change of the operational semantics of REDLOG will have little effect on the form of the REDLOG language itself. Many of the constructs used by the EPILOG [25] language are naturally embodied in the form of REDLOG which currently exists, providing promising support for this hope.

The development of a more concise semantics definition method than the current denotational semantics definition is another area worth investigating. The suitability (and length) of a semantics definition method is related directly to the language being defined, at least with the current technology.

A weakness of the current implementation of REDLOG is the lack of infinite constructs. Chapter 11 section 1 has already pointed out how these could be introduced into REDLOG, but these suggestions have not been subjected to the practical rigors of implementation.

A further weakness is the lack of standardised input and output facilities. This is a deliberate move pending the implementation of infinite constructs which can be used to elegantly implement such facilities.

13. Appendix: EBNF Syntax Definition of REDLOG Equations

Module ::= **module** Module_Identifier Interface Definitions

Interface ::= Provides [Interface]
| Uses [Interface]

Provides ::= **provides** {Domain_Spec}

Uses ::= **uses** identifier {, Identifier} (;|Carriage_Return)}

Class ::= **class** Class_Identifier **ako** Identifier Provides Definitions

Definitions ::= **equations** {Equation | Domain_spec}

Domain_Spec ::= Derivation
| Declaration

Derivation ::= Type-heading ::= Type-constructor

Declaration ::= Name :: Type-constructor

Typeheading ::= Name | Parameterised-name

Paramaterised-name ::= Name Parameter

Parameter ::= Variable
| Basic-type
| Type-constructor

Type-constructor ::= Union
| Restriction
| Tuple
| List
| Mapping
| Basic-type
| Derived-type

Equation ::= Head_Term {[Result_Part] [Cond_Part]}. | Domain_Spec.

Head_Term ::= Data_Term
| Functor_Symbol [(Data_Term{,Data_Term})]

Result_Part ::= ≡Term

Cond_Part ::= Neck_Symbol Cond_Term

Cond_Term ::= Term
| Term Junctive_Symbol Cond_Term
| Negation Cond_Term

```
Neck_Symbol   ::= :-
                | if
                | where
Junctive_Symbol ::= and
                | ,
                | or
Negation      ::= not
                | ~
Term          ::= Data_Term
                | (Functor_Symbol
                | Constructor_Symbol) [(Term{ , Term } ) ]
Data_Term     ::= Variable
                | Constructor_Symbol[(Data_Term{, Data_Term})]
Functor_Symbol ::= Identifier
Constructor_Symbol ::= Identifier
Variable      ::= Identifier
Name          ::= Identifier
```

Syntactical abbreviations and sugarings for numbers and lists, as seen in chapter 3, are also defined for standard REDLOG. The syntax for scope rules is not defined here.

14. References and Bibliography

Functional Languages and Extensions

[1] Abramson, H., "A Prological Definition of HASL: A Purely Functional Language with Unification-Based Conditional Binding Expressions", *New Generation Computing*, Vol. 2, No. 1, pp 3-35, 1984.

[2] Backus, J., "Can Programming be Liberated from the Von-Neumann Style? A Programming Language and its Algebra", *ACM Turing Award Lecture*, 1978.

[3] Burstall, R.M., D.B. MacQueen and D.T. Sanella "HOPE: an Experimental Applicative Language", *Lisp Conference*, ACM, 1980, pp136-143.

[4] Hoffman, C.M. and M.J. O'Donnell, "Programming with Equations", *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 1, January 1982, pp83-112.

[5] Hughes, J.M. "The Design and Implementation of Programming Languages", *Ph.D Thesis*, Oxford University 1984.

[6] Lindstrom, G., "Functional Programming and the Logical Variable", *Proc. Principles of Programming Languages*, ACM, 1985, pp266-280.

[7] McCarthy, J. et al. "Lisp 1.5 Programmers Manual", *The MIT Press*, 1962.

[8] Milner, R., "A Proposal for Standard ML", *Proc. Symp. LISP and Functional Languages*, ACM, New York, August 1984, pp184-197.

[9] Rees, J. and W. Clinger (Eds), "Revised^3 Report on the Algorithmic Language Scheme", *SIGPLAN Notices*, Vol. 21 No. 12, December 1986.

[10] Turner, D.A., "Miranda: A non-strict Functional language with Polymorphic Types", *Proc. IFIP Int. Conf. on Functional Programming and computer Architecture*, Nancy France, Sept. 1985 (Springer-Verlag Lecture Notes in Computer Science, Vol. 205).

[11] Wright, D.A., "Rhodes-FP and Declarative Languages", *B.Sc. (Hons) Thesis*, Rhodes University, 1986.

Prolog and Extensions

[12] Clocksin, W.F. and C.S. Mellish "Programming in Prolog", *Springer-Verlag* 1981.

[13] Colmerauer, A. "Prolog and Infinite Trees", in K.L. Clark and S. -A. Tarnlund, Eds., *Logic Programming*, Academic Press, 1982.

[14] Covington, M.A. "Eliminating unwanted loops in Prolog", *SIGPLAN Notices*, 20, 1(1985), pp20-26.

Integrated Languages

[15] Barbuti, R., M. Bellia, G. Levi and M. Martelli, "LEAF: A Language which Integrates Logic, Equations and Functions", In *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom, Eds., Prentice-Hall, 1986.

[16] Cohen, S., "The Applog Language", In *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom, Eds., Prentice-Hall, 1986.

[17] Darlington, J., A.J. Field and H. Pull, "The Unification of Functional and Logic Languages", In *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom, Eds., Prentice-Hall, 1986.

[18] Goguen, J.A. and J.Meseguer "EQLOG: Equality, Types, and Generic Modules for Logic Programming", *Journal of Logic Programming 1*, 2(1984), pp 179-210.

[19] Kornfeld, W.A., "Equality for Prolog", *Proc. 7th Int. Joint Conf. on Artificial Intelligence*, 1983, pp 514-519.

[20] Malachi, Y., and R. Waldinger, "TABLOG: A New Approach to Logic Programming", *Proceedings of the Symposium on LISP and Functional Programming*, ACM, 1984, pp323-330.

[21] Subrahmanyam, P.A. and J.-H. You "Pattern Driven Lazy Reduction: a Unifying Mechanism for Functional and Logic Programs", *Proc. Principles of Programming Languages*, ACM, 1984, pp228-234.

[22] Wright, D.A., "An Integration of Reduction and Logic for Programming Languages", *M.Sc Thesis*, Rhodes University (in preparation).

Parallel Prologs and Implementation

[23] Gregory, S., "Parallel Logic Programming in PARLOG", *Addison-Wesley*, 1987.

[24] Shapiro, E.Y., "A Subset of Concurrent Prolog and its Interpreter", TR-003, *ICOT*, 1983.

[25] Wise, M.J., "Prolog Multiprocessors", *Prentice-Hall*, 1986.

[26] Wright, D.A., "Two Promising Approaches to Fifth Generation Systems", *Technical Document 87/36*, Rhodes University 1987.

Background

[27] Bourne, S.R., "The Unix System V Environment", *Addison-Wesley*, 1987.

[28] Burton, F.W., "Functional Programming for Concurrent and Distributed Computing", *The Computer Journal*, Vol. 30 No. 5, October 1987, pp437-450.

[29] Cardelli, L. and P. Wegner, "On Understanding Types, Data Abstraction and Polymorphism", *ACM Computing Surveys*, Vol. 17 No. 4, December 1985, pp471-522.

[30] Collier, P., "Type Inference in the presence of a Basic Type Hierarchy", private communication, 1987.

[31] Goldfarb, D., "The Undecidability of the Second Order Unification Problem", *Journal of Theoretical Computer Science*, 13(1981), pp 225-230.

[32] Good, D.I., "Mechanical Proofs about Computer programs", in *Mathematical Logic and Programming Languages*, The Royal Society, Prentice-Hall, 1985.

[33] Henderson, P., "Functional Programming: Application and Implementation", *Prentice-Hall*, 1980.

[34] Hindley, R., "The Principal Type Scheme of an Object in Combinatory Logic", *Trans. Am. Math. Soc.*, 146, December 1969, pp29-60.

[35] Huet, G., "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems", *Journal of the ACM*, 27, 4(1980), pp 797-821.

[36] Kowalski, R.A., "The Relation between Logic Programming and Logic Specification", in *Mathematical Logic and Programming Languages*, The Royal Society, Prentice-Hall, 1985.

[37] Kowalski, R.A., "Logic for Problem Solving", *North-Holland*, 1979.

[38] Lloyd, J.W., "Foundations of Logic Programming", *Springer-Verlag*, 1984.