PROCEEDINGS

Guest Editor: Judy M Bishop

Organised by the SA Institute of Computer Scientists in association with the Computer Society of SA

Sponsored by PerseTel and the FRD
SPECIAL ISSUE - 7th SA COMPUTER SYMPOSIUM

PREFACE

When the first SA Computer Symposium was held at the CSIR in the early eighties, it was unique. There was no other forum at the time for the presentation of research in computer science. In the intervening decade, conferences, symposia and workshops have sprung up in response to demand, and now there are several successful ventures, some into their third or fourth iteration. Each of these addresses a specific topic - for example, hypermedia, expert systems, parallel processing or formal aspects of computing - and attracts a specialised audience, well versed in the subject and eager to learn more. For the main part, the proceedings are informal, and certainly not archival.

SACRS, though, is still unique, in that it deliberately covers a broad spectrum of research in computing, and in addition, seeks to provide a lasting record of the proceedings. To achieve the second aim, we negotiated with the SA Institute of Computer Scientists for the proceedings to form a special issue of the SA Computer Journal, and the copy you have in front of you is the result. The collaboration between the symposium committee and the journal's editorial board placed high standards on the refereeing and final presentation of the papers, to the symposium's benefit, while we were still able to maintain a fresh, audience-oriented approach to the selection of papers.

This is SACJ's first such special issue, and the largest issue (at 145 pages) to date. We hope that it is only the beginning of future such collaborations.

In all 29 papers were received, all were refereed twice, and 19 were chosen for presentation by the programme committee. All the papers were thoroughly revised by the authors on the basis of the referee's comments, and the committee's suggestions aimed at making the material more accessible to a broadly-based audience. Papers had to be new, and not to have been presented elsewhere, a requirement that is still unusual within the SA conference round.

A third goal of SACRS has been to invite keynote speakers, usually from overseas. This year, we are fortunate to present Dr Vinton Cerf, the father of the Internet and a world-renown expert on computer networks. Although his paper is not available for this special issue, it will appear later in SACJ. Through the good offices of Professor Chris Brink of UCT, we also have three other speakers from Germany, Canada and the US adding interest to the event, and two of their papers appear in this issue.

The programme committee originally devised a theme for the symposium - "Computing in the New South Africa". We received several queries as to the meaning of this theme, but unfortunately few papers that addressed it directly. One prospective author went as far as to enquire whether computer research would survive in the new South Africa. Another felt that his work was definitely not in the theme, as it was genuine, old world, basic, theoretical science! Nevertheless, there are two papers that consider one of South Africa's key issues, that of language. Others look at the success we have achieved in applying technology to mining, and the future of low-cost operating systems. In all, the mix of papers represents a balance between the theoretical and the practical, the past and the future, all firmly based in the computing of the present.

Organising the symposium has involved the hard work of several people, and I would like to thank in particular:
- Derrick Kourie, my co-organiser, and the editor of SACJ for his invaluable advice and hard work throughout the planning and implementation stages;
- Riel Smit, the production editor, for attaining such a high standard in such a short time for so many papers;
- Gerrit Prinsloo and the staff at the CSSA for their efficient and quite delightfully unfussy organisation;
- Persetel for their very generous sponsorship of R25000, and Tim Schumann for taking a genuine interest in our events;
- the Foundation for Research Development for sponsoring Vint Cerf's visit;
- and finally the Department of Computer Science of the University of Pretoria for providing the ideal working conditions for undertaking ventures of this kind, and especially Roelf van den Heever for his unfailing encouragement and support.

Judy M Bishop
Organising Chairman, SACRS 1992
Guest Editor, SACJ Special Issue
Referees

The journal draws on a wide range of referees. The following were involved in the refereeing of the papers selected for this special issue. Their role in certifying the papers and their contribution to enhancing the quality of papers is sincerely appreciated.

John Barrow
Ronnie Becker
Danie Behr
Sonia Berman
Liesbeth Botha
Theo Bothma
Chris Brink
Peter Clayton
Ian Cloete
Antony Cooper
Elise Ehlers
Quintin Gee
Andy Gravell
Wendy Hall
David Harel
Scott Hazelhurst
Derrick Kourie
Willems Labuschagne
Doug Laing
Dave Levy
Graham McLeod
Hans Messerschmidt
Deon Oosthuizen
Ben Oosthuysen
Neil Pendock
D Petkov
Martin Rennhackkamp
Ceces Roon
Jan Roos
John Shochot
Morris Sloman
Riel Smit
Pat Terry
Walter van den Heever
Lynn van der Vegte
Herman Venter
Herna Viktor

UNISA
University of Cape Town
University of Pretoria
University of Cape Town
University of Pretoria
University of Pretoria
University of Cape Town
Rhodes University
Stellenbosch University
CSIR
Rand Afrikaanse University
DALSIG
University of Southampton
University of Southampton
The Weizmann Institute of Science
University of the Witwatersrand
University of Pretoria
UNISA
ISM
University of Natal
University of Cape Town
University of the Orange Free State
University of Pretoria
University of Pretoria
University of the Witwatersrand
University of Natal
Stellenbosch University
University of Pretoria
University of Pretoria
University of the Witwatersrand
Imperial College, London
University of Cape Town
Rhodes University
University of Pretoria
University of Pretoria
University of Fort Hare
Stellenbosch University
HILOG — a Higher Order Logic Programming Language

Roland A. Paterson-Jones Peter T. Wood
Department of Computer Science, University of Cape Town, Rondebosch, 7700

Abstract

HILOG is a new logic programming language with a higher-order syntax allowing the elegant expression of many tasks requiring meta-predicates in Prolog. We are in the process of developing a compiler and programming environment for HILOG which is based on extending the Warren Abstract Machine used in modern Prolog implementations.

Keywords: logic programming, compilers
Computing Review Categories: 1.2.3, D.3.4, F.4.1

Received March 1992, Accepted April 1992

1 Introduction

In logic programming the need to write programs which manipulate predicates, functions, and even atomic formulae often arises. For example, a program defining the transitive closure of any binary relation, \( R \), can be written in Prolog as follows:

\[
\text{closure}(R, X, Y) \leftarrow \\
C = \text{..} [R, X, Y], \text{call}(C).
\]

\[
\text{closure}(R, X, Y) \leftarrow \\
C = \text{..} [R, X, Z], \text{call}(C), \\
\text{closure}(R, Z, Y).
\]

It can be seen that Prolog has has several built-in procedures which allow the discussion of predicates, functions and formulae. In the above example, the Prolog builtins \text{call} and \text{\ldots} fulfill this function.

However, Prolog is an offspring of first-order predicate calculus, which only allows simple terms to be bound to variables. The higher-order features which allow the manipulation of predicates and formulae are ad-hoc additions, without a basis in the formal semantics underlying Prolog itself. This is manifest in the unreadable form of the code for our transitive closure — without a solid knowledge of the two builtin predicates, the code is difficult to fathom.

There have been several attempts to remedy this problem by introducing higher-order logic programming languages. Unhappily, truly higher-order predicate calculi prove to be computationally very expensive. HILOG was introduced in [5] as an attempt to heal the rift. In essence it is a logic with a higher-order syntax, allowing greater expressive power than Prolog. However the underlying semantics are first-order which means that it is efficiently implementable.

A predicate for generic transitive closure can be written as follows in HILOG.

\[
\text{closure}(R)(X, Y) \leftarrow \\
R(X, Y).
\]

\[
\text{closure}(R)(X, Y) \leftarrow \\
R(X, Z), \text{closure}(R)(Z, Y).
\]

As can be seen, HILOG variables have the freedom to range over predicates, allowing this very natural code.

2 HILOG

The syntax of HILOG closely resembles that of Prolog. A program comprises a sequence of rules and facts. A rule comprises a head and a body. A fact is a bodiless rule. The
body comprises a sequence of (predicate) terms.

HILOG, however, makes no distinction between variables, predicate symbols, function symbols and terms themselves. Thus any term can appear as a predicate, including a variable and a complex term. The rule

\[ p(\text{foo}_1, \text{foo}_2) \rightarrow \]
\[ q(\text{one}, \text{two})(\text{X}), \]
\[ Y(\text{foo}_3, \text{foo}_4), \]
\[ Z. \]

is legal in HILOG.

The standard convention of using identifiers starting with an upper case letter for variables, and identifiers starting with a lower case letter for predicate and function symbols as well as language objects (hereafter referred to as constants) will be adopted.

The higher-order features of HILOG can be exploited to provide elegant solutions to various classical problems. We consider examples of these in the remainder of this section.

Complex terms can be used as predicates in HILOG. This allows the definition of generic predicates, such as the transitive closure predicate from Section 1:

\[ \text{closure}(\text{R})(\text{X}, \text{Y}) \rightarrow \]
\[ \text{R}(\text{X}, \text{Y}). \]
\[ \text{closure}(\text{R})(\text{X}, \text{Y}) \rightarrow \]
\[ \text{R}(\text{X}, \text{Z}), \text{closure}(\text{R})(\text{Z}, \text{Y}). \]

\text{closure} is used here as a second-order predicate which, when given a predicate as its argument, returns the transitive closure of this predicate. This HILOG representation, as we have seen above, has a far more natural form than an equivalent Prolog program. Furthermore it is slightly more robust — a HILOG query such as

?\text{-} \text{closure}(\text{R})(\text{foo}, \text{foo}_2).

will return the names of all binary predicates containing (\text{foo}, \text{foo}_2) in their transitive closures. Prolog struggles to express this.

This leads us on to another important application of HILOG, viz. schema browsing. In HILOG, browsing can be performed through the same language as that used for data retrieval. For instance (after [5])

\[ \text{bin...rel}(\text{Y})(\text{X}) \rightarrow \]
\[ \text{X}(\text{Y}, \text{Z}). \]
\[ \text{bin...rel}(\text{Y})(\text{X}) \rightarrow \]
\[ \text{X}(\text{Z}, \text{Y}). \]

?\text{-} \text{bin...rel}(\text{john})(\text{X}).

will list each binary relation containing \text{john} as a tuple element.

The ability to work with sets has received considerable attention in the logic programming community. Several attempts have been made to incorporate set manipulation into a logic programming framework ([1, 4]). However their use has to be severely restricted in order to avoid logical paradoxes and computational intractability.

HILOG's loose syntax allows a natural representation for sets. We need only observe that a relation is merely a set of tuples and that the extensions of a predicate are relations. Also, a unary relation is isomorphic to a set of elements. So, given that we can manipulate predicates entirely independently of their tuples in HILOG, we can use predicate names as sets. The following rule from [6] defines the set of satisfied employees working for each manager:

\[ \text{satempl}(\text{Boss})(\text{Empl}) \rightarrow \]
\[ \text{supervises}(\text{Boss}, \text{Empl}), \]
\[ \text{salary}(\text{Boss}, \text{B_sal}), \]
\[ \text{salary}(\text{Empl}, \text{E_sal}), \]
\[ \text{E_sal} > \text{B_sal}. \]

The 'set-term' \text{satempl}(\text{Boss}) denotes the set of satisfied employees working for the given \text{Boss}. The term \text{satempl} represents a set of sets, one set for each \text{Boss}. The set-term could be used, for example, in the following manner, defining packages of benefits with groups of satisfied employees:

\[ \text{package1}(\text{health.ins}). \]
\[ \text{package1}(\text{life.ins}). \]
\[ \text{package2}(\text{free.car}). \]
\[ \text{package2}(\text{long.vacs}). \]

\[ \text{benefits}( \text{package1}, \text{satempl}(\text{john})). \]
\[ \text{benefits}( \text{package2}, \text{satempl}(\text{bob})). \]

We could ask for the benefit packages that \text{bob} enjoys as follows:

?\text{-} \text{benefits}(\text{X}, \text{Y}), \text{Y}(\text{bob}).

Notice the natural use of \text{Y} as a set in the first term, and the easy expression of \text{bob} \in \text{Y} as \text{Y}(\text{bob}).

We could query the actual benefits as follows:

?\text{-} \text{benefits}(\text{X}, \text{Y}), \text{Y}(\text{bob}), \text{X}(\text{Z}).

where \text{Z} will be instantiated with the benefits of the benefit packages.

In essence HILOG circumvents the complexities of sets by only allowing access to elements one-at-a-time.

Viewing each distinct syntactic symbol as a single intention provides a neat hook for objects in HILOG. We associate a HILOG symbol with an object's identity, and then associate relevant properties to this identity. For example (after [12]):

\[ \text{dan}(\text{person}). \]
\[ \text{dan}(\text{name}, \text{‘Daniel’}). \]
\[ \text{dan}(\text{birthdate}, 12.\text{oct.}56). \]

defines an object with identifier \text{dan}, of class \text{person}, with attributes \text{name} and \text{birthdate} of the given values.

\[ \text{bill}(\text{student}). \]
\[ \text{bill}(\text{name}, \text{‘William’}). \]
\[ \text{bill}(\text{birthdate}, 13.\text{sep.}71). \]
\[ \text{bill}(\text{entry..date}, 4.\text{sep.}89). \]

defines an object with identifier \text{bill} of class \text{student}, and with the given attributes.

Subclassing can be expressed simply as:

\[ S(\text{person}) \rightarrow S(\text{student}). \]

where the (predicate) variable \text{S} ranges over object identifiers.

Methods can be defined on classes by means of HILOG rules:

\[ \text{P}(\text{age}, \text{Age}) \rightarrow \]
\[ \text{P}(\text{person}), \]
\[ \text{P}(\text{birthdate, BDate}), \]
\[ \text{today}(\text{TDate}), \]
\[ \text{date.diff}(\text{TDate, BDate, Age}). \]

54 SACJ/SART No 7, 1992
This method computes the age property of objects, based on birthdate and today's date. Note that students inherit this method.

3 Implementation

HiLOG has the same basic underlying mechanism of first-in-line rule selection, and left-to-right processing of conjunctive queries as Prolog. The difference lies in rule triggering, and in the matching process. Both of these are predictable since, firstly, the extended term structure of HiLOG requires an extended matching algorithm and, secondly, the freedom of structure of predicate terms requires a different rule selection algorithm.

Matching

Matching in HiLOG is a simple extension of the Prolog matching algorithm. It is done as follows (jargon-wise, a complex term such as X(a)(b,c,D(c)) comprises a head, X(a), and arguments b, c, and D(c)):

- an unbound variable matches any term by binding itself to the value of the term,
- a constant matches an identical constant (apart, of course, from an unbound variable), and
- a complex term matches (apart from an unbound variable) another complex term iff
  - each complex term has the same number of arguments (i.e. the same arity),
  - the heads match, and
  - corresponding arguments match.

For example, d(X)(1,2,3), matches with Y(e)(1,2,3) binding X—> e and Y—> d, and with Z(A,B,C) binding Z—> d(X), A—> 1, B—> 2 and C—> 3.

On the other hand, d(X)(1,2,3) does not match e(Y)(1,2,3), nor does it match Z(a,b,c), nor does it match Z(A,B,C,D), nor d(X)(2).

Rule Triggering

Rule triggering in HiLOG is done by matching the whole query term with the whole rule head. This is clearly necessary as in HiLOG the 'predicate' terms no longer need consist of a predicate and an argument list, but can have the structure of any legal term. Thus a HiLOG program cannot be partitioned by predicates, like Prolog, as this has no meaning in HiLOG. Every rule is thus treated as a candidate for every query in HiLOG, whereas in Prolog a query only looks at rules for its specific predicate. This leads to severe efficiency problems in HiLOG.

Indexing is used in Prolog to reduce the number of candidate rules for a query. Essentially the structure of the argument terms is used to reject rules whose arguments are obviously not going to match the query arguments. For example, a query term with a constant foo in one of its argument positions looks only at rules with the same constant foo or a variable in the same position. Most Prolog implementations index on the first argument of a predicate.

Indexing is crucial to the efficient implementation of HiLOG. In order for HiLOG to approach the efficiency of Prolog, a far more extensive system of indexing is needed, in particular, to dispatch queries to their particular predicates.

We have considered several indexing schemes. One of them is a simple extension of the Prolog index, comprising a two level index: indexing is performed first on the query term head to determine the predicate, and then on the first argument (as for Prolog). Of course in HiLOG the predicate might not be defined yet:

?- X(foo).

Nor may any indexing be possible at all:

?- X.

These pathological cases have to be catered for in the indexing scheme.

Another scheme is the use of an indexing tree which is traversed according to the structure of the query term. The leaf nodes of the tree then contain the list of (pointers to) candidate rules for a query.

The problem of indexing in HiLOG is certainly not yet adequately resolved — the success of the language in practice depends on a satisfactory solution to this crucial problem.

Dynamic Code

The extended syntax of HiLOG allows the dynamic calling of predicates, i.e. query terms whose predicates are unknown at compile-time:

?- X(foo, foo2).

Prolog deals with dynamic code — via the call builtin — by performing a run-time compilation of the dynamic query and then executing this temporary code. In order for this to work, Prolog needs the predicate of the dynamic query to be well-instantiated at the time of the dynamic call. There is no way in Prolog of querying predicate names, as they are not considered objects of the language as they are in HiLOG. HiLOG circumvents the need for dynamic compilation (and the call builtin) by allowing arbitrary terms in the predicate position.

Builtin predicates, such as write, in Prolog and HiLOG are not treated as normal queries, but rather as calls to internal code which performs their specific operations. This can cause problems in HiLOG with code such as:

?- X=write,...,X(hallo).

HiLOG does no run-time inspection of the dynamic query and thus cannot recognise it as distinct from a normal query. A HiLOG system thus has to include amongst its system predicates, an interpretive version of each of the builtins. For example, there would be a rule for the builtin write:

write(X) ← <call the write builtin>.

In particular it is necessary to have interpretive rules for the query conjunction , and disjunction , meta-predicates, as well as for the control predicates repeat, true, fail, !, if-then-else (⇒ ;) etc. which are handled in Prolog by the clause compiler.

Although the interpretive predicates take up space, the resulting dynamic code execution is significantly faster than
that of Prolog, as the compilation step is discarded. In practice, the space taken up by each of the interpretive predicates is not prohibitive due to the highly compact nature of the internal form of the rules (cf. Section 4). The interpretive rule for write given above uses no more than fifteen bytes of memory.

Term Parsing
Term parsing in Prolog is directed by the dynamic definition of prefix, postfix, and infix operators. The extended syntax of HILOG allows a deeper interplay between operators and predicate extension (the placing of a tuple behind a predicate). For example, assuming we have defined the constant "$p" as a prefix operator, the term "$p(X)" is normally parsed as "$p(X)\)\). However this could conceivably be parsed as "$p(X)\)\), a generic predicate term where the operator binds only to the term head.

For this reason predicate extension is treated as a pseudo-operator in HILOG. The priority of predicate extension is defined as higher than all of the normal operators, but lower than the highest priority available to the system. For "$p(X)\)\)" to be parsed as "$p(X)\)\)\), we simply define "$p" to be a prefix operator of priority higher than that of predicate extension.

Our implementation of HILOG is based on the compilation of HILOG clauses. It is described in more detail in the next section.

4 The HILOG Abstract Machine

Modern Prolog implementations are generally based on an emulated machine for Prolog called the Warren Abstract Machine (WAM) [2, 13]. The WAM comprises a machine architecture and a set of machine instructions which implement the term handling and query evaluation associated with Prolog. Prolog rules and queries are compiled to WAM-code which is then executed on the WAM emulator.

We have designed and implemented a HILOG Abstract Machine (HAM), which is an extension of the WAM incorporating the extensions which are necessary for HILOG. Changes are made to incorporate the extended syntax, the new indexing scheme, and aspects associated with dynamic code.

Term Representation
In the HAM, terms are stored in cells. The cells comprise a tag which describes their type — constant (CON), variable (REF), list (LIS) and complex structure (ARG) — and a value. The value of a constant cell is simply the internal code for the constant. The value of a REF, LIS or ARG cell is a pointer: a REF cell points to the current instantiation of the variable, and LIS and ARG cells point to a sequence of cells containing their component terms: two cells for a list (head and tail) and $n + 1$ cells for a structure of arity $n$ (one for the head and one for each argument). The arity of a structure is stored as part of the tag. An unbound variable is represented by a REF cell which points to itself.

There is one minor difference between the WAM and the HAM’s internal term storage: in Prolog complex terms must have a specific functor, unlike HILOG where arbitrary terms can appear in the functor position. Thus in the WAM the first cell pointed to by an ARG cell is a special <functor | arity> cell, whereas in the HAM a general term cell appears here. In the WAM the arity of the structure is stored in this special functor cell, whereas in the HAM it is stored as part of the ARG cell tag.

Query Evaluation
Queries are evaluated in the HAM by constructing the query term (in cell format) on the internal stack, and then calling the rule code to find matching rules. An internal HAM register, Areg, points to this query term. Again this is a slight departure from the WAM practice. In Prolog, a rulebase is partitioned by predicate whereas in HILOG it is not. Thus in Prolog, once the predicate of a query term is ascertained, there is no longer any need to store it as only the rules for that predicate will be called. The WAM therefore only stores the arguments of the query term, and uses an array of query argument registers, Areg1 – Aregn, for this purpose.

For example the query term p(foo,foo2) is stored as:

In the HAM

A0: < ARG | adr >

Stack

A1: < CON | foo >
A2: < CON | foo2 >
adr: < CON | foo >
< CON | foo2 >

There are four sets of instructions in the HAM which deal with query term construction and matching:

- The put- instructions place query terms into the query term register A0, and complex sub-terms into other term registers used during term construction.
- The set- instructions build the arguments of a complex term or list onto the internal stack. These instructions only occur after a putarg or putlis instruction. As an example, the query
  ?- p(foo,X).
  is constructed with the HAM code:
  putarg 2 /*structure of arity 2*/
  setcon p /*head*/
  setcon foo /*arg1*/
  setvar Y1 /*internal var Y1 represents X*/
- The get- instructions match a query term with a rule head. The query term is of course held in the register A0 during rule-head matching.
- The unify- instructions match the arguments of a complex term or list. For example, the code for the rule
  q(X,Y) ← . . . .
  will start with the HAM code
  getarg 2 /*structure of arity 2*/
  unifycon q /*head*/
  unifyvar Y1 /*arg1*/
  unifyvar Y2 /*arg2*/
  These instructions are essentially the same as the equivalent WAM instructions. The WAM get- and put- instructions
take an extra argument for the argument register used.

There are several inefficiencies in the HAM’s query evaluation when compared to the WAM. Firstly, there is an extra step in the construction of the query term, namely that of the construction of the term head. Secondly, the term consumes more stack space since, in the WAM, a query term uses stack space only for a complex argument term. Thirdly, the process of matching a query term with a rule head again has the extra step of predicate matching.

Unfortunately, very little can be done to improve this situation. The predicate construction and matching steps are essential for queries and rule heads whose ‘predicate’ is either undefined (an unbound variable) or generic (a complex term). We have experimented with the storage of the query term arguments in argument registers rather than on the stack. The instructions for term matching and term construction have to be modified under this scheme, and new instructions are needed to cope with dynamic queries. On the whole, however, this new scheme would appear to hold some promise.

**Execution Control**

HILOG is based on SLD-type resolution. Conjugate queries are satisfied by satisfying the component queries from left to right. Candidate rules for a query are attempted from top to bottom in the rulebase.

Queries are executed in the HAM by constructing the query term (and placing it into the register A0), and then issuing a call instruction. The call instruction routes the query to the appropriate index encoding code, or in the worst case routes it to the code for the first rule in the rulebase. We have not yet implemented indexing, so our system at present uses the latter option, treating every rule in the rulebase as a candidate for every query. The call instruction in the WAM has an extra argument for the predicate of the query term. Queries are sent straight to the rules for this predicate.

A rule is executed (after being call’ed by a query) by matching the head of the rule with the query term (in A0) and then executing the body of the term as a conjugate query. Each triggered rule uses a stack environment to store the return address and the values of the variables of the rule, very much like a subroutine in procedural programming. The HAM instructions allocate and deallocate create and destroy the rules stack environment. If the head of the rule does not match the query term or any of the body variables fail, the rule fails, invoking backtracking.

The proceed instruction is used at the end of a conjugate query (and at the end of a rule) to indicate successful satisfaction of the query. The variable bindings associated with the successful query are returned automatically through REF cell chains to the variables of the top-level query.

As an example of HAM code, we will consider the HILOG rule

\[
p(X, foo) => q(X, Y), Y(foo).
\]

The HAM code for this rule is

\[
\text{allocate}
\]

\[
\text{getarg 2}
\text{unifycon q}
\text{unifyvar Y1}
\text{unifycon foo}
\]

\[
\text{putarg 2}
\text{setcon p}
\text{setvar Y1}
\text{setvar Y2}
\text{call}
\]

\[
\text{putarg 1}
\text{setval Y2}
\text{setcon foo}
\text{call}
\]

\[
\text{deallocate}
\text{proceed}
\]

Backtracking is implemented in the HAM by the establishment of choice points wherever there is more than one candidate rule for a query. Choice points are pushed onto the internal stack and contain information defining the present state of variable instantiation as well as the next rule to attempt. When a rule fails, and backtracking occurs, the choice point is used to undo the spurious effects of the failed branch, and to send execution to the next candidate rule. The HAM instructions which manipulate the choice points are closely linked to the nature of the indexing scheme, and so they will not be presented here.

The HAM has instructions which implement tail-end optimization, a generalisation of tail-end recursion. This results in very efficient code for rules with one body term (chain rules) — no environment is needed — and for tail-end recursive rules which are turned into iterative rules.

**The Cut**

The cut ! is implemented through HAM instructions which manipulate the choice points on the stack. A cut instruction essentially strips choice points from the stack back to a stack contained in the HAM register B0. B0 is set by the call instruction to the most recent choice point, so a cut instruction within the call will strip choice points up to and including the choice point for the query call.

The HAM’s cut instructions for normal rules are identical to those of the WAM and will not be described in detail here. Interpretive code, however, necessitates some extensions to the WAM scheme. A dynamic conjugate query such as

\[-..., X='.', '\ldots', X(P, I(Q, R)), \ldots\]

may contain instances of the cut within the sub-queries P, Q and R. However, if this dynamic code is executed normally, the cut register B0 will be reset incorrectly at each level of the conjugate query.

This problem is solved in the HAM by using an extra register B00 which stores the cut point of the parent query, during a dynamic call. If the dynamic call is to a predicate such as ',', or ';', the cut register B0 is reset to the value of B00, thus fooling the system into thinking that the inter-
mediate call did not exist, and causing a subsequent cut to cut to the correct choice point.

5 Conclusion

HiLOG is a new language, upwardly compatible with Prolog, which offers an exciting new expressive power. It offers a clean syntax and semantics to many of the metapredicates available in Prolog, and allows the expression of properties and structures which Prolog struggles to handle neatly.

We are working on a HAM-based implementation of HiLOG. At present we have a complete HAM emulator for which a simple HiLOG clause compiler and interactive programming environment exists. An indexing scheme has been designed in detail. We still have to implement the indexing scheme and write a good clause compiler.

Future work includes the realization of HiLOG as a logic programming language with all of the features of Prolog. This will entail the extension and redefinition of many of the Prolog built-in predicates. Another area of interest is the implementation of an object-logic programming environment in HiLOG, justifying the original creation of this novel logic.

References

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines.

- Use wide margins and 1\frac{1}{4} or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) \LaTeX file(s), either on a diskette, or via e-mail/ftp – a \LaTeX style file is available from the production editor;
2. In camera-ready format – a detailed page specification is available from the production editor;
3. As an ASCII file accompanied by a hard-copy showing formatting intentions:

- Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

Further instructions on how to reduce page charges can be obtained from the production editor.

4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for \LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 3 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

Contents

Machine Translation from African Languages to English
DG Kourie, WJ vd Heever and GD Oosthuizen .................................. 1

Automatically Linking Words and Concepts in an Afrikaans Dictionary
PZ Theron and I Cloete ........................................................ 9

A Lattice-Theoretic Model for Relational Database Security
A Melton and S Shenoi ......................................................... 15

Network Partitions in Distributed Databases
HL Viktor and MH Rennhackkamp ........................................... 22

A Model for Object-Oriented Databases
MM Brand and PT Wood ...................................................... 27

Quantifier Elimination in Second Order Predicate Logic
D Gabbay and HJ Ohlbach .................................................... 35

Animating Neural Network Training
E van der Poel and I Cloete ................................................. 44

HiLOG - a Higher Order Logic Programming Language
RA Paterson-Jones and PT Wood ........................................... 53

ESML - A Validation Language for Concurrent Systems
PJA de Villiers and WC Visser .............................................. 59

Semantic Constructs for a Persistent Programming Language
SB Sparg and S Berman ....................................................... 65

The Multiserver Station with Dynamic Concurrency Constraints
CF Kriel and AE Krzesinski .................................................. 75

Mechanizing Execution Sequence Semantics in HOL
G Tredoux ........................................................................ 81

Statenets - an Alternative Modelling Mechanism for Performance Analysis
L Lewis ............................................................................. 87

N Pendock ........................................................................ 95

Galileo: Experimenting with Graphical User Interfaces
R Apteker and JM Bishop ..................................................... 99

Placing Processes in a Transputer-based Linda Programming Environment

Accessing Subroutine Libraries on a Network
PH Greenwood and PH Nash ............................................... 117

A Multi-Tasking Operating System Above MS-DOS
R Foss, GM Rehmet and RC Watkins .................................... 122

Using Information Systems Methodology to Design an Instructional System
BC O'Donovan ................................................................ 126

Managing Methods Creatively
G McLeod ...................................................................... 131

A General Building Block for Distributed System Management
P Putter and JD Roos ........................................................... 141