PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

An Implementation for a Higher Level Logic Programming Language

Anthony S.K. Cheng* Ross A. Paterson†

Software Verification Research Centre
Department of Computer Science
The University of Queensland
4072, Australia

Abstract

For representing high level knowledge, such as the mathematical knowledge used in interactive theorem provers and verification systems, it is desirable to extend Prolog's concept of data object. A basic reason is that Prolog data objects—Herbrand objects—are terms of a minimal object language, which does not include its own object variables, or quantification over those variables.

Qu-Prolog (Quantifier Prolog) is an extended logic programming concept which takes as its data objects, object terms which may include free or bound occurrences of object variables and arbitrary quantifiers to bind those variables. Qu-Prolog is unique in allowing its data objects to include free occurrences of object variables.

In this paper the design of the abstract machine for Qu-Prolog is given. The underlying design of the machine reflects the extended data objects and Qu-Prolog's unification algorithm.

1 Introduction

The extended logic programming language Qu-Prolog (Quantifier Prolog) [Cheng *et al.* 1991, Paterson and Hazel 1990, Paterson and Staples 1988, Staples *et al.* 1988a, Staples *et al.* 1988b] has been designed to provide improved support for language processing applications such as interactive proof systems. Its main feature is that it supports higher level symbolic data types than does Prolog. In particular, the data objects which Qu-Prolog reasons about are terms of a full first order logic syntax, which includes both object level variables and arbitrary bindings of object level variables.

The language λProlog [Miller and Nadathur 1986], which extends Prolog with typed *lambda*-terms, may also be used for these purposes. Qu-Prolog is weaker, in that its terms correspond to second-order *lambda*-terms; substitution is provided, but not application of terms to terms. However, in Qu-Prolog, as in traditional notation, term variables may refer to open terms, raising further questions of whether an object level variable occurs free in a term, or whether two object level variables are distinct.

The Qu-Prolog Abstract Machine (QuAM) [Cheng and Paterson 1990] is designed as the target for compilation of the logic programming language Qu-Prolog. QuAM is developed from the Warren Abstract Machine (WAM). New mechanisms are introduced to handle quantified terms and substitutions and flexible programming in Qu-Prolog. This paper presents the basic structure of the language and describes its implementation.

The main features of Qu-Prolog are described in section 2. In section 3, unification is extended to Qu-Prolog terms. The design of QuAM is given in section 4. Some examples are given in section 5. It is assumed that the reader has some knowledge of the design of WAM [Aït-Kaci 1990, Warren 1983] and the compilation of logic programming languages.

2 Qu-Prolog – the Language

Qu-Prolog has Prolog as a subset, and uses Edinburgh Prolog syntax for constants and structures, and for ordinary variables which are intended to range over arbitrary object level terms. These variables will be referred to as *meta* variables, in recognition of the meta level status of the Qu-Prolog language relative to the object language. In addition, Qu-Prolog introduces syntax to represent object level variables and quantifiers, as follows.

Qu-Prolog has other features not described here. These include persistent variables, which are used to manage incomplete information in the database. For a description of persistent variables and their implementation, see [Cheng and Robinson 1991].

2.1 Object Variables

Since object level variables are simply part of the object level syntax, it might seem natural to name them at the Qu-Prolog (meta) level by constants. Instead, Qu-Prolog refers to object level variables only by a type of Qu-Prolog (meta) level variable, called object-var variables. The semantics of object-var variables is that they range over object level variables. The success of this approach reflects the common intuition that object level variables are interchangeable.

*e-mail: chena@cs.uq.oz.au

†present address: Department of Computing, Imperial College, London SW7.

The phrase 'object variable' is commonly used to abbreviate 'object-var variable' since it has no other use in describing Qu-Prolog syntax. For an occasional reference to a variable of the object language, the phrase 'object level variable' will be used.

Qu-Prolog object variables have the same lexical conventions as constants. In order to distinguish them, object variable notations must be declared by `object_var/1`. The declaration convention is that an explicit declaration of an object variable name also implicitly declares all variant names derived by appending an underscore followed by a positive integer. The standard library declares the atoms `x`, `y` and `z` as object variables.

As each object variable is intended to range over all object level variables, it is important to know whether two object variables denote different object level variable. This information can be supplied implicitly or by explicit use of the predicate `distinct_from/2`. For example, `x distinct_from y` asserts that `x` and `y` do not denote the same object level variable. By default, all object variables occurring in the same clause/query are distinct from each other.

Remark: In fact Qu-Prolog makes internal use of some meta level constants representing object level variables. These terms, called *local* object variables, are mentioned below but they are not discussed here in detail. Their key role is as 'new' variables, for use when changing bound variables. This newness is implemented by a convention that they are excluded from instantiations of user accessible meta variables and object-var variables.

2.2 Quantifiers

Qu-Prolog can reason about object level terms which include arbitrary quantifiers, in much the same way that Prolog can reason about terms which include arbitrary function symbols. The user declares quantifier notations as needed. Thus it is possible to have representations of $\int$ for integral calculus as well as $\forall, \exists$ for first order logic.

Distinct quantifier notations in Qu-Prolog represent distinct object level quantifiers. Qu-Prolog uses the traditional prefix notation for quantified terms. Quantifiers are declared explicitly by executing

$$op(\mathit{Precedence}, quant, Q)$$

where Q is the representation for the quantifier; Q must have the same lexical structure as a Prolog constant.

2.3 Substitutions

Throughout logical reasoning, the need for substitutions arises naturally. Qu-Prolog directly supports parallel substitution for free occurrences of object level variables.

The syntax for substitutions in Qu-Prolog is

$$[t_1/x_1, \ldots, t_n/x_n] * term$$

where $x_1, \ldots, x_n$ are object variables and $t_1, \ldots, t_n$ are arbitrary Qu-Prolog terms.

Qu-Prolog substitutions are evaluated at unification time, in accordance with the standard concept of correct substitution into quantified terms, which substitutes only for free occurrences of variables and which changes bound variables to avoid capture of free variables from the substituted terms. For a term $s_1 * \ldots * s_n * y$ where $s_1, \ldots, s_n$ is a sequence of substitutions, the substitutions are applied from right to left. That is, s_n is applied to y first. The effect of applying a substitution to a term can be observed with this example:

$$s * [t_1/x_1, \ldots, t_n/x_n] * y.$$

After applying the rightmost substitution, the result will be:

- $s * t_i$ if for some $i = 1, \ldots, n, x_i = y$, or
- $s * y$ if for all $i = 1, \ldots, n, x_i$ *distinct_from* y.

It is also possible that there is insufficient information at a particular stage to determine which of these cases applies. In that case evaluation of the substitution will be delayed. That may lead to delaying of unification subproblems, perhaps extending beyond the current unification call.

As well as substitutions appearing in user inputs, the system can generate substitutions via unification. For example, the problem *lambda* x $A =$ *lambda* y B has the solution $A = [x/y] * B$.

2.4 Example

As a small example, we give a λ-calculus evaluator in Qu-Prolog. The terms of the λ-calculus are transcribed directly, except that we use the infix constructor `@` for application. First, we declare the quantifier `lambda` and the application operator:

```
?- op(700, quant, lambda).
?- op(600, yfx, @).
```

Now the following predicate defines the structure of λ-terms:

```
lambda_term(x).
lambda_term(A@B) :-
        lambda_term(A),
        lambda_term(B).
lambda_term(lambda x A) :-
        lambda_term(A).
```

For example, the following are λ-terms:

```
x
lambda x x
(lambda x x)@y
lambda x (x@y)
```

Note that λ-terms may contain free object variables.

Now we can define a single-step reduction predicate on λ-terms:

```
?- op(800, xfx, =>).
(lambda x A)@B => [B/x]*A.
A@B => C@B :- A => C.
A@B => A@C :- B => C.
lambda x A => lambda x B :- A => B.
```

The first clause is the well-known β-rule. The others allow rewrites anywhere in the expression. If desired, we could also add the η-rule:

```
lambda x A@x => A :- x not_free_in A.
```

The full reduction relation in the usual reflexive, transitive closure of the single-step reduction predicate:

```
?- op(800, xfx, =>*).
A =>* C :- A => B, !, B =>* C.
A =>* A.
```

3 Unification

Qu-Prolog extends Prolog unification to cover the new data objects in the language. Two terms are unified if they are equivalent up to changes of bound variables (α equivalent). Since unification for Prolog terms is not changed (except that Qu-Prolog includes occurs checking), our discussion will concentrate on the new features.

Because Qu-Prolog unification is more difficult than ordinary unification—it is not decidable, but semi-decidable [Paterson 1989]—we often encounter sub-problems which cannot be solved at that point in the computation, but we may be able to make further progress on them later. Such sub-problems are delayed, waiting for a relevant variable (or variables) to be instantiated, at which point they are re-attempted. If the sub-problems remain unsolved at the end of query solution, they are displayed as part of the answer. This approach has proved practical in our implementation.

We have also found it useful to delay sub-problems to avoid branching. As a simple example, consider the unification problem $[X/y]{*}Z = c$, where c is a constant (a similar situation arises with structures). The unification can succeed in one of two ways:

- Imitation: $Z = c$. Here the substitution has a null effect on Z.
- Projection: $Z = y$ and $X = c$.

Hence it is impossible to determine a unique most general unifier. Rather than branch the unification problem, Qu-Prolog delays it until the binding of Z is known.

3.1 Object Variables

Since an object variable is intended to range over object level variables, and since object variables are the only Qu-Prolog terms of this type, an object variable can be instantiated only to another object variable. Further, unification fails if the object variables denote distinct object level variables. Also, whenever a meta variable is unified with an object variable, the meta variable is bound to the object variable.

3.2 Quantifiers

To motivate the treatment of unification for quantified terms, consider

$$lambda\ x\ x \quad = \quad lambda\ y\ y$$

Intuitively, the two terms are unifiable without instantiation of x or y, because the terms are the same up to change of bound variable. To unify x and y would be incorrect: the two terms are α equivalent even if x and y denote distinct object level variables. Hence during quantifier unification, Qu-Prolog uses substitution to rename the bound variables to a common bound variable. The bound variable must not appear in the unified terms. This is where the local object variables mentioned previously are used. In general, a problem of the form $q\ x\ t = q\ y\ t'$ is reduced to

$$[\nu/x] * t \quad = \quad [\nu/y] * t'$$

for some new local object variable ν, and unification continues. Here is how the approach applies to the example, (ν is a local object variable).

$$\begin{array}{rcl} lambda\ x\ x & = & lambda\ y\ y \\ lambda\ \nu\ [\nu/x] * x & = & lambda\ \nu\ [\nu/y] * y \\ [\nu/x] * x & = & [\nu/y] * y \\ \nu & = & \nu \\ & (success) & \end{array}$$

A substitution containing local object variables, when applied to a meta variable, may be removed by a rule called *inversion*: a problem of the form $[\nu/x] * X = t$ is reduced to the two problems

$$X \quad = \quad [x/\nu] * t,\ x\ not\_free\_in\ t$$

For example, we have the following reduction:

$$\begin{array}{rcl} lambda\ x\ A & = & lambda\ y\ y \\ [\nu/x] * A & = & [\nu/y] * y \\ A & = & [x/\nu] * [\nu/y] * y, \\ & & x\ not\_free\_in\ [\nu/y] * y \\ A & = & x,\ x\ not\_free\_in\ \nu \\ A & = & x \end{array}$$

Unification produces the answer $A = x$.

As a further example, consider

$$lambda\ x\ A = lambda\ y\ x$$

Since x does occur free on the right and cannot occur free on the left, this unification problem should fail. In Qu-Prolog unification, that failure is detected when, at the time of calculation of $A = [x/\nu]*[\nu/y]*x$, the constraint $x\ not\_free\_in\ [\nu/y]*x$ is generated and tested; and after substitution evaluation, the test fails.

Such *not_free_in* constraints may be delayed if they cannot be immediately decided. For example, the unification problem

$$lambda\ x\ A \quad = \quad lambda\ y\ [x/z] * Z$$

gives the solution

$$\begin{aligned} A &= [x/\nu] * [\nu/y] * [x/z] * Z \\ & \quad \textit{provided:} \\ & \quad x \ \textit{not\_free\_in} \ [\nu/y] * [x/z] * Z \end{aligned}$$

In the absence of further information about Z, the *not_free_in* test must be delayed.

3.3 Occurs Checking

Unlike Prolog, occurs checking is included as standard in Qu-Prolog unification. However, it is not always possible to determine whether a variable occurs in the final form of a term. For example, it is impossible to determine whether X occurs in the term $[X/y] * Z$ without knowing more information about Z. If Z is bound to y, X occurs in the term. On the other hand, if Z is bound to a constant c, X does not occur.

Thus, if we are considering a sub-problem of the form $X = t$, we cannot always reduce the problem. We use two conservative syntactic conditions:

- If X occurs in t outside of any substitution, and t is not of the form $s * X$, the unification fails, for the X must appear in t no matter how other variables are instantiated.
- If X does not appear in t, including substitutions, X is instantiated to t.

If neither of these conditions is met, the unification sub-problem must be delayed, pending further instantiation of X.

4 The Qu-Prolog Abstract Machine

One of the design criteria for QuAM is that the efficiency of ordinary Prolog queries within Qu-Prolog must be maintained wherever possible. Thus, most of the features of WAM are retained and the description below will concentrate on the differences between QuAM and WAM. The current implementation of QuAM differs from the present description in that it uses an experimental representation for structures, intended for future enhancements to the Qu-Prolog language with higher-order predicates and multiple-place quantifiers. The present paper focuses on other aspects of the machine, so we omit these details here, assuming a WAM-like representation of structures. Because of the difference of the representation of the structures, no performance evaluation will be given. A description of the current implementation can be found in [Cheng and Paterson 1990].

4.1 Data Objects

Unbound Variables

Because of the association with delayed problems described below, the representation of a self reference cell for unbound variables as in WAM is inapplicable. A data cell with a VARIABLE tag is used to indicate an unbound variable in Qu-Prolog. The value field of the data cell contains a pointer to a list of delayed problems associated with the variable (Figure 1). Although the representation of variables is different to WAM, the classification into temporary and permanent variables, the age determining method and the rules of binding a variable are retained.

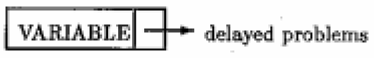

Figure 1: An Unbound Qu-Prolog Variable

The REFERENCE tag is retained to indicate that one variable is bound to another one. When two heap variables are bound together, the one created more recently points to the one created earlier on the heap. The delayed problems from the younger one are appended to those of the older one.

Unbound Object Variables

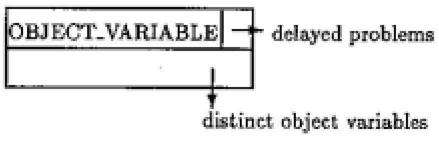

Figure 2: An Unbound Object Variable

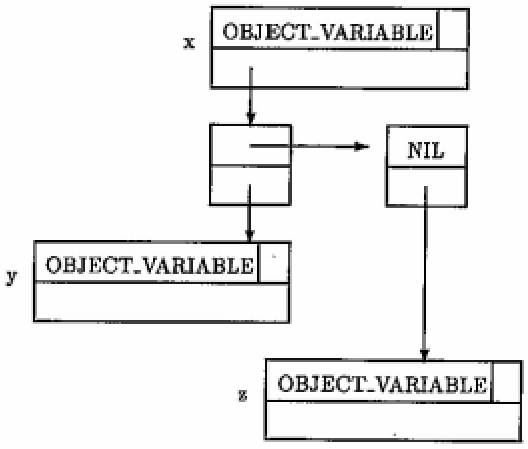

Figure 3: x *distinct_from* y and x *distinct_from* z

A separate tag OBJECT_VARIABLE is given to the object variables to distinguish its function from the variables. The value field has the same purpose as the value field in variables. The second cell in an object variable points to a list of object variables from which it is distinguished (Figures 2, 3). Rather than record all object variables in the distinctness list, an ALL_DISTINCT tag is placed in this cell for local object variables.

The classification method, the binding rules and the age determining method used for variables is also applied to object variables.

The OBJECT_REFERENCE tag indicates that an object variable is bound to another object variable. When two object variables are bound together, the distinctness information from both object variables are merged together and placed in the older object variable and the delayed problems will be woken up.

Quantified Terms

Qu-Prolog currently allows 1 place quantifiers (*i.e.* quantifiers with one bound variable) only. To represent quantified terms in Qu-Prolog, a tag QUANTIFIER is introduced, analogous to the STRUCTURE tag of the WAM. Such a value points to a three contiguous cells, containing the quantifier atom, a reference to the bound object variable, and the body of the quantified term (Figure 4).

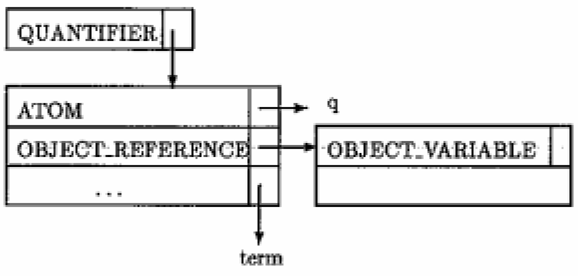

Figure 4: Quantified Term q x term

Substitution Operators

In QuAM, an application of one or more substitutions to a term is represented as a data cell, marked with a SUBSTITUTION_OPERATOR tag and pointing to a pair of cells. The first cell contains a pointer to the list of substitutions, while the second is a data cell denoting the term (Figure 5). The list of substitutions is stored in reverse order, with the innermost substitution at the front, to simplify evaluation.

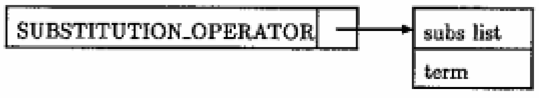

Figure 5: *sub* $*$ *term*

An ordinary parallel substitution is represented as a data cell with the property tag, containing a pointer to a pair of cells. The first of the cells is a pointer to the parallel substitution, while the second represents the rest of the substitution list. A parallel substitution involving n pairs of object variables and terms is represented as a block of $2n+1$ cells; the first contains the size of the substitution, while the renaming $2n$ cells refer to the object variables and terms. Again the substitution pairs are stored in the reverse order for easy evaluation (Figure 6).

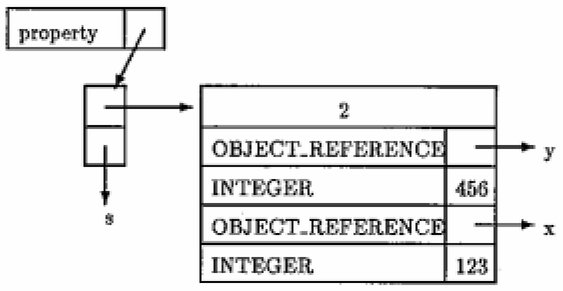

Figure 6: A Parallel Substitution $s * [123/x, 456/y]$

Each substitution list contains a marker describing the property of the substitution list. It is used during unification to assist the determination of whether or not the unification can be solved by projection. In general, a problem of the form $s * A = t$, where t is a constant, structure, quantified term or object variable, can always be reduced by imitation. If s is known not to contain any terms of the same top-level structure as t, then the problem cannot be solved by projection. Thus branching is eliminated and we can proceed by imitation. Otherwise, the unification problem will be delayed to avoid branching. In most cases, the whole substitution list must be examined in order to eliminate projection. In special cases, the marker will contain enough information to make a complete search unnecessary.

It is also convenient to know if a substitution list consists solely of renamings generated by quantifier unification, as such a list can be safely inverted. Thus, each substitution list is marked as one of:

- invertible: the substitution list consists solely of renamings.
- object variables only: the substitution list is not invertible, but its range contains only object variables.
- others: the range of the substitution list contains constants, structures, quantifiers or meta variables.

4.2 Data Areas

QuAM supports the same data areas as in WAM. The heap provides space to store data objects as well as the distinctness information and linking cells required for delayed problems. The local stack holds choice points and environments. The choice points are enlarged to reflect the extra data areas and registers.

Because the delayed problems list and distinctness information must be reset to their previous value upon backtracking, the method of trailing (*i.e.* resetting the address to null) used in WAM is inapplicable. Each entry in the trail is extended to be a pair of addresses and previous values to provide extra information for backtracking.

In addition to the standard WAM data areas, a delayed problems stack that holds any delayed problem generated during unification is provided. Apart from containing pointers to the arguments for the delayed problem, it has a type tag and a solved tag. The type tag indicates whether the delayed problem is a unification or a *not_free_in* problem (Figure 7). The solved tag is set whenever the problem is solved.

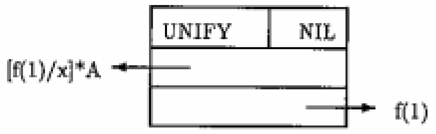

Figure 7: Delayed Problem: $[f(1)/x] * A = f(1)$

When a query is solved, any delayed problem that remains is printed as a constraint to the solution. Storing the delayed problems in a separate area allows fast access to the problems when the solution is printed.

4.3 Registers

There are a few extra registers used in QuAM:

- the top of the delayed problems stack,
- a list of formerly delayed problems that have been woken up,
- The *substitution pointer* register points to the entry in the parallel substitution where the next component is to be added.

As well as the X registers, there is an associated set of registers, known as the XS (X substitution) registers, which hold the substitution of a term when the substitution and the term of a SUBSTITUTION_OPERATOR data cell are broken up during dereference. This procedure enables the substitution to be passed from the outer structure to the inner terms effectively.

Because each Y register is one data cell in size, and an OBJECT_VARIABLE is two cells in size, a Y register cannot hold an OBJECT_VARIABLE directly, and instead contains a reference to an OBJECT_VARIABLE in the heap.

4.4 Instruction Set

For each new data object provided in QuAM, there are *put* and *get* instructions to build and unify the data object. The new instructions are:

put_object_variable X_i
: Create a new object variable on the heap, and place a reference to it in X_i.

get_object_variable X_i X_j
: Copy the object variable reference in X_j into X_i.

put_object_value X_i X_j
: Copy the object variable reference in X_i into X_j.

get_object_value X_i X_j
: Unify XS_j, X_j with the object variable referenced by X_i.

put_quantifier q X_i X_j X_k
: Construct a quantified term, with quantifier q, bound object variable X_i and body X_j, and place a reference to it in X_k.

get_quantifier q X_i X_j X_k
: Match the term in XS_k, X_k with a quantified term, with quantifier q and bound object variable X_i. The body is placed in XS_j, X_j.

In each of the last two instructions, the register X_i must have been previously set to an object variable.

Note that some of these instructions use the XS registers, while others ignore them, expecting any substitution to be incorporated into the term in the X register. Thus during head matching substitutions are conveniently accessible in the substitution registers, allowing efficient dereferencing, and sharing of substitutions. However, if such a value is to be a sub-term, its substitution (if any) must be re-incorporated into the term.

There is a set of *put* instructions to build substitutions, but no corresponding set of *get* instructions. This is because a substitution occurring in the head must be built in the same way as if it had occurred in the body, and then the substituted term must be unified with the corresponding head argument (or component). The instructions available are:

put_subs_operator X_i X_j
: Combine XS_j and X_j into a SUBSTITUTION_OPERATOR, and place a reference to it in X_i.

put_empty_subs X_i
: Set XS_i to an empty substitution.

put_parallel_subs n X_i
: Prepend a parallel substitution, consisting of n pairs (each supplied with the next instruction), to XS_i.

put_parallel_subs_pair X_i X_j
: Add a pair, substituting X_j for the object variable referred to by X_i, to the parallel substitution currently under construction.

put_subs X_i X_j
: Transfer a substitution from XS_i to XS_j.

set_object_property X_i
: Set the property tag on XS_i to "object variables only".

determine_property X_i
: determine the property tag of XS_i.

The only new procedural instructions are:

do_delayed_problems
Solve any woken delayed problems. This instruction is executed after the head has been matched.

not_free_in
Perform a *not_free_in* test during quantifier unification.

4.5 Dereference

Because of the presence of substitution, additional operations are included into the dereference algorithm. The substitutions are evaluated during dereference whenever possible. Given an object variable, the substitution will map the object variable to its value. Depending on the type of the data object encountered in the term, dereference also simplifies the substitution before returning.

5 Examples

A number of small examples are given here to highlight the design differences between QuAM and WAM.

5.1 Quantified Terms

Quantified terms are constructed in a similar fashion to the unary structures, except for the object variable. The following sequence of instructions shows how a quantified term `lambda x x` is built in register X_1:

```
put_object_variable X0                 % x
put_quantifier lambda X0 X0 X1
```

Matching a quantified term is slightly more complicated than structure matching. Apart from matching the term from outside in (*i.e.* match the quantifier before matching the body), it must establish that the bound variable of the quantified term in the head does not occur freely in the body of the quantified term from the query. Thus, a not_free_in instruction must be executed before the quantifier matching is performed. The following instructions match the argument X_0 with the head argument `(lambda x A)@B`:

```
get_structure @/2 X0
unify_variable X2                  % lambda x A
unify_variable X0                   % B
put_object_variable X3              % x
put_empty_subs X3
not_free_in X3 X2
get_quantifier lambda X3 X2 X2   % A
```

5.2 Substitutions

QuAM is designed to create substitutions independent of the term. The term is created before the substitution. The example `[a / x, b / y] * A` illustrates this property.

```
put_variable X0 X0            % A
put_empty_subs X0
put_object_variable X1        % y
put_atom 'b' X2
put_object_variable X3        % x
put_atom 'a' X4
put_parallel_subs 2 X0        % * A
put_parallel_subs_pair X1 X2 % [b/y] * A
put_parallel_subs_pair X3 X4 % [a/x,b/y]*A
determine_property X0
```

If the substitution is nested inside another term, an extra step is needed. A SUBSTITUTION_OPERATOR data object is created to group the substitution and its associated term together. To construct the term `f([a/x, b/y] * A)`, the following additional instructions are required:

```
put_subs_operator X0 X0   % group together
put_structure f/1 X1
unify_value X0
```

Whenever a substitution is associated with a term in the head, that term together with the substitution will be built by *put* instructions and general unification will be called. For example, consider the following clause from the λ-calculus evaluator:

```
(lambda x A)@B => [A/x]*B.
```

In section 5.1 above, we gave the translation of the matching of the first argument, leaving x in X_3, A in X_2 and B in X_0. The following instructions match the second argument (in X_1):

```
put_subs_operator X0 X0 % group together B
put_subs_operator X2 X2 % group together A
put_empty_subs X0              % *B
put_parallel_subs X0 1         % *B
put_parallel_subs_pair X3 X2  % [A/x]
determine_property X0
get_value X0 X1  % unify with the argument
```

Note that A and B must both be combined with their substitutions, if any. In the case of A, this allows the value to fit into a cell in the substitution pair. In the case of B, the substitution must be incorporated into the value, and the substitution register set to empty, so that the new substitution will be outside any existing substitutions.

If the substitution is nested within another term, the outer term is matched by the *get* instructions, while the substitution is built and unified with the appropriate component.

6 Conclusions

QuAM has been implemented in C under the SUN 4 environment. The compiler was initially implemented in NU-Prolog [Naish 1986], and subsequently transferred to Qu-Prolog, which includes Prolog as a subset.

Qu-Prolog, including the extensions and features mentioned here, has been motivated particularly by the need to rapidly prototype interactive proof systems, and currently it is the implementation language for a substantial experimental proof system [Robinson and Tang 1991].

Acknowledgements

John Staples, Peter Robinson, Gerard Ellis and Dan Hazel have made substantial contributions to the design and implementation of QuAM. This research was supported by the Australian Research Council.

References

[Aït-Kaci 1990] H. Aït-Kaci, The WAM: a (Real) Tutorial, Report No. 5, Paris Research Laboratory (PRL), France, 1990.

[Cheng and Robinson 1991] A.S.K. Cheng and P.J. Robinson, An Implementation for Persistent Variables in Qu-Prolog 3.0, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1991.

[Cheng and Paterson 1990] A.S.K. Cheng and R.A. Paterson, The Qu-Prolog Abstract Machine, Technical Report No. 149, Key Centre for Software Technology, Department of Computer Science, University of Queensland, February 1990.

[Cheng *et al.* 1991] A.S.K. Cheng, P.J. Robinson and J. Staples, Higher Level Meta Programming in Qu-Prolog 3.0, *Proc. of 8th International Conference on Logic Programming*, Paris, June 1991.

[Miller and Nadathur 1986] D.A. Miller and G. Nadathur, Higher-order Logic Programming, *Proc. of 3rd International Conference on Logic Programming*, London, July 1986.

[Naish 1986] L. Naish, Negation and Quantifiers in NU-Prolog, *Proc. of 3rd International Conference on Logic Programming*, London, July 1986.

[Paterson 1989] R.A. Paterson, Unification of Schemes of Quantified Terms, Technical Report No. 154, Key Centre for Software Technology, Department of Computer Science, University of Queensland, Dec. 1989.

[Paterson and Hazel 1990] R.A. Paterson and D. Hazel, Qu-Prolog 3.0 – Reference Manual, Technical Report No. 195, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1990.

[Paterson and Staples 1988] R.A. Paterson and J. Staples, A General Theory of Unification and Solution of Constraints, Technical Report No. 90, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1988.

[Staples *et al.* 1988a] J. Staples, P.J. Robinson, R.A. Paterson, R.A. Hagen, A.J. Craddock and P.C. Wallis, Qu-Prolog: an Extended Prolog for Meta Level Programming, *Proc. of the Workshop on Meta Programming in Logic Programming*, University of Bristol, June 1988.

[Staples *et al.* 1988b] J. Staples, R.A. Paterson, P.J. Robinson and G.R. Ellis, Qu-Prolog: Higher Level Symbolic Computation, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1988.

[Robinson and Tang 1991] P.J. Robinson and T.G. Tang, The Demonstration Interactive Theorem Prover: Demo2.1, Technical Report 91-4, Software Verification Research Centre, University of Queensland, September 1991.

[Warren 1983] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, 1983.