Qu-Prolog 8.10 User Guide

Peter J. Robinson

December 1, 2009

Abstract

Qu-Prolog is an extension of Prolog with built-in support for computation on symbolic data which involves quantifiers, object variables and substitutions. This user guide introduces many Qu-Prolog features via a collection of examples. Unification is informally presented through a series of unification problems given to the Qu-Prolog interpreter. A lambda calculus example and a simple Natural Deduction style theorem prover are discussed. Examples of using multiple threads, high-level communication and GUIs are also presented. Familiarity with elementary Prolog is assumed.

1 Introduction

Qu-Prolog is an extension of the well-known Prolog logic programming language. Prolog was defined originally for symbolic computation applications and this has remained one of its strengths. Prolog supports symbolic computation on only a very limited vocabulary however. In particular the symbolic data type on which Prolog is based has no recognition of syntactic objects such as quantifiers, object variables or substitutions. Qu-Prolog extends Prolog primarily by providing built-in support for computation on symbolic data which may include quantifiers, object variables and substitutions. Qu-Prolog was designed originally as an implementation and tactic language for interactive theorem provers, particularly those that carry out schematic proofs. In particular Qu-Prolog is the implementation language of the Ergo theorem prover [1, 2]. It is now also used more generally in the prototyping of program development environments, and is capable of a wide range of applications.

Here are some examples of Qu-Prolog terms. These examples show how easily mathematical and logical notations can be represented.

- Here is a term that can represent the rule for beta reduction in the lambda calculus (described further in Section 3). This example uses an atom `lambda` assumed to have been declared a quantifier symbol, an object variable `x`, and Qu-Prolog's built-in substitution operator `[-/-]`.

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

  Atoms are declared as quantifier symbols in the same way as Prolog declares infix, prefix and postfix operators (that is, using `op/3` - see Section 3 for an example).

- The following two clauses can represent simplifications for definite integrals in real number calculus. Here we use the `!!` notation, which introduces a quantifier symbol that is not an atom (or has not been explicitly declared to be a quantifier).

  ```
  simplify(!!integral(A,B) x (F+G),
  ```

```
!!integral(A,B) x F + !!integral(A,B) x G).

simplify(!!integral(A,B) x C, C*B - C*A) :- x not_free_in C.
```

The two-place predicate `not_free_in` is predefined in Qu-Prolog.

- A term that can represent the rule for existential introduction in predicate calculus reasoning (if P is true for some value of x then $\exists x P$ is true).

  ```
  [T/x]P --> ex x P
  ```

Readers should refer to the Qu-Prolog reference manual [3] for a detailed description of the syntax of Qu-Prolog and of Qu-Prolog library support.

Note that Qu-Prolog has no built-in semantics for any quantifier, except that every quantifier binds variables. The required semantics is programmed for each application, using clauses such as those illustrated above.

The Qu-Prolog language has notations which makes reference to all the terms of the object language, and in particular includes Qu-Prolog variables that range over the variables in the object language. A Qu-Prolog variable that ranges over object language variables is strictly called an *object-var* variable in Qu-Prolog. However, in practice it is often loosely referred to as an object variable, since practical programming uses the Qu-Prolog language exclusively and so never needs to use the object language directly.

The Qu-Prolog language also includes a substitution operator, as seen in the examples given above.

The remainder of this guide presents Qu-Prolog-specific features through a collection of examples. The source code of the example programs given in the following sections can be found in the `examples` directory of the Qu-Prolog release. It is assumed the reader is familiar with Prolog notation and Prolog programming. Readers who are new to Prolog are referred to [4, 5].

Qu-Prolog is generally consistent with SICStus Prolog [6]. Most SICStus programs run under Qu-Prolog.

2 Unification

Prolog unification attempts to find instantiations of variables that make two terms syntactically identical. This is fine for quantifier-free languages (like Prolog) but for languages like Qu-Prolog that include quantifier notations syntactic identity is too strong a requirement. That is because the names of bound object variables are insignificant to the meaning: quantifiers and other notations which differ only in the names of bound variables are regarded as equal (they are called α-equivalent).

For Qu-Prolog, unification is an attempt to find instantiations of variables that make terms α-equivalent. The reader is referred to [7] for a formal discussion of unification in Qu-Prolog.

BEWARE: Prolog books often use the word 'substitution' to refer to what we call instantiation. Since Qu-Prolog includes a substitution operator, we need to distinguish the two concepts carefully. Intuitively, instantiations 'move through' all terms whereas substitutions cannot affect bound variables. Strictly, instantiations operate on meta-level terms. An instantiation replaces a Qu-Prolog variable (which is a meta-level variable) by a Qu-Prolog term. The Qu-Prolog substitution operator on the other hand describes an operation on object-level terms: $[B/x]A$ describes the replacement of free occurrences of an object-level variable x in an object-level term A by copies of the object-level term B, with changes to the names of bound variables in A as needed to give the intended semantics (that is, to assign a value to free occurrences of the object variable denoted by x).

We now present examples that show key features of Qu-Prolog unification. Each example is from a session with the Qu-Prolog interpreter and is accompanied by a short explanation of the behaviour. We assume `q` has been declared as a quantifier and that `x`, `y`, `z`, `w` have been declared as object variable prefixes.

Example 1

```
| ?- q y f(y) = q x f(x).

y = y
x = x
```

The Qu-Prolog interpreter responds to each successful query by displaying the instantiations of all variables appearing in the query. In this example `x` and `y` are object variables and the unification succeeds without instantiating `x` or `y`. This is because the query asserts the α-equivalence of two terms `q y f(y)` and `q x f(x)`, and those two terms are already α-equivalent.

Example 2

```
| ?- q y A = q x f(x).

y = y
A = f(y)
x = x
```

Here Qu-Prolog finds the most general instantiation of `A` that makes the terms α-equivalent.

Example 3

```
| ?- q y A = q x C.

y = y
A = [y/x]C
x = x
C = C

provided:

y not_free_in [$/x]C
```

In this example we want to find most general instantiations of `A` and `C` which make `q y A` and `q x C` α-equivalent. Thus `A` and `C` will be the same except that in `A` all free occurrences of `x` in `C` will be replaced by `y`. Note that the answer to this unification includes the constraint `y not_free_in [$/x]C`. Here '`$`' is a system supplied constant.

To explain Qu-Prolog's response to this query we consider separately the cases when `x` and `y` refer to different object-level variables, and when they refer to the same object-level variable. This is for the purpose of explanation only however: Qu-Prolog has not made either choice. Assuming `x` and `y` refer to different object-level variables, then the two terms cannot be unified if `C` contains a free occurrence of `y` because then `q y A` has no free occurrence of `y` whereas `q x C` does. On the other hand, if `x` and `y` refer to the same object-level variable then no such restriction applies. The constraint `y not_free_in [$/x]C` covers both cases. In the first case the constraint reduces to `y not_free_in C`, and in the second case it reduces to `true`.

This example shows that unification can produce constraints on the variables. Such constraints are stored as delayed problems that are woken when appropriate variables are instantiated. An alternative to delaying would be to explore, during unification, the various possibilities for instantiating variables. This is not done in Qu-Prolog however, since the resulting programming paradigm is harder to control. Exploring possibilities for instantiating variables can however be programmed when desired.

Example 4

```
| ?- q [x,y] A = q [z,w] B.

x = x
y = y
A = [x/z, y/w]B
z = z
w = w
B = B

provided:

y not_free_in [$/z, $/w]B
x not_free_in [$/z, $/w]B
z not_free_in [w]
w not_free_in [z]
y not_free_in [x]
x not_free_in [y]
```

This example extends the previous one to include quantifier notations and substitution operator usages which bind multiple variables in parallel. A list of variables bound in parallel by a quantifier may be called a *binding list*. Qu-Prolog constrains all variables appearing in a binding list to be mutually distinct, that is they refer to different object-level variables. Constraints requiring object variables to be distinct are displayed in the answer in the form x `not_free_in [` x_1`, ...,`x_n`]` where x is distinct from each of $x_1, \ldots, x_n$. The symmetry of the distinctness relation is reflected in answers by some redundancy in the `not_free_in` constraints.

Example 5

```
| ?- [A/x]B = 3.

A = A
x = x
B = B

provided:

[A/x]B = 3
```

This example shows the Qu-Prolog approach to dealing with unifications that are 'too difficult', in the sense that there is not a unique explicit most general solution. Here there are two solutions to the problem. One is to instantiate `B` to `3` and the other is to instantiate `B` to `x` and `A` to `3`. Such problems are delayed until more information is known about the problem. In this simple example no further information becomes available and Qu-Prolog makes only a trivial response to the query. However in a wider context in which `A` or `B` are later instantiated in other computation, this subproblem would then be resumed and solved as far as possible.

Example 6

```
| ?- x not_free_in y, [x/z]y = x.

x = x
y = y
```

```
z = z

provided:

x = [x/z]y
y not_free_in [x]
x not_free_in [y]
```

This is a trivial example from another class of problems that are 'too difficult'. Problems in this class have exactly one solution but, in general, to determine this fact requires more computation than has been built into Qu-Prolog's unification algorithm (such omissions are not necessarily oversights - there can be significant performance advantages in delaying the solution of subproblems until they are easy to solve). Again, such problems are delayed until more information is known about the variables involved. In this example the required unifier is `y = z`.

Example 7

```
| ?- [A/x]B = 3, A = 2.

A = 2
x = x
B = 3
```

Example 8

```
| ?- [A/x]B = 3,  x not_free_in B.

A = A
x = x
B = 3
```

In each of Examples 7 and 8, the query of Example 5 is augmented by more information about the variables in the problem, and the delayed problem is now solved uniquely.

Qu-Prolog's `examples` directory includes in `incomplete_unify.ql` an example of a user defined predicate called `incomplete_unify` which attempts to find solutions to delayed unification problems by applying a collection of heuristics. This predicate will be discussed briefly in a later section.

Example 9

```
| ?- X = f(X).

no
```

Qu-Prolog applies 'occurs checks' to all unifications (in principle; in practice the actual calculations are omitted when the outcome can be predicted). This is important for a language that is used to implement schematic theorem provers. Doing occurs checks adds an overhead to unification and to minimise this overhead Qu-Prolog tries to avoid unnecessary occurs checks.

We now present more unification problems without commentary.

Example 10

```
| ?- F(X) = g(a).

F = g
X = a
```

Example 11

```
| ?- !!Q X B = q [x,y] f(x,y).

Q = q
X = [x0, x1]
B = f(x0, x1)
x = x
y = y

provided:
```

```
y not_free_in [x]
x not_free_in [y]
x1 not_free_in [x0]
x0 not_free_in [x1]
```

Example 12

```
| ?- q x:int f(x) = q y:T A.

x = x
y = y
T = int
A = f(y)
```

Example 13

```
| ?- q x:y f(x)  = q y:T A.

x = x
y = y
T = y
A = f(y)
```

Note that the term after the '`:`' in the binder is outside the scope of the quantifier. This syntax is provided to support the use of typed binding notations.

3 The Lambda Calculus

The first example program we consider is a simple recognizer and evaluator for terms of the lambda calculus. First we briefly review the nature of the lambda calculus for those not already familiar with it.

The lambda calculus is a simple formal language which contains only some basic notation, called lambda terms, for describing functions, and some basic rules for transforming ("reducing") function notations. Use of these rules typically simplifies lambda terms so as to make clearer which values they denote. Because the (untyped) lambda calculus described here is so simple and general, some of its terms represent rather unusual functions. However that need not be a distraction.

In brief, terms of this lambda calculus are constructed recursively from variables and a lambda quantifier λ as follows.

- Each variable is a lambda term.
- If A and B are lambda terms, then so is $A(B)$, which intuitively represents the application of a function A to an argument B.
- If A is a lambda term and x is a variable, then $\lambda x A$ is a lambda term, which intuitively represents the function whose value at x is A.

A term of the form $\lambda x A$ is referred to as a *lambda abstraction*.

It is seen below that Qu-Prolog can directly execute this recursive definition so as to implement a lambda term recognizer.

The evaluator presented below is an implementation of a top-down, left-first evaluator (a "lazy" evaluator). The first part gives rules for reduction as a finite sequence of single reduction steps (" contractions"). The second part gives the usual contraction rules for the lambda calculus.

Here is the Qu-Prolog code (in file `lambda.ql`) for this example.

```
%
% Quantifier and operator declarations.
%
?- op(500, quant, lambda).   % declare lambda as a quantifier

?- op(800, xfx, =>).        % single reduction

?- op(800, xfx, =>*).       % zero or more reductions

% a recogniser for lambda terms
```

```
lambda_term(x).              % object variables are lambda terms

lambda_term(A(B)) :-
        lambda_term(A),
        lambda_term(B).

lambda_term(lambda x A) :-
        lambda_term(A).

% lambda evaluation

A =>* C :-
        A => B,
        B =>* C, !.
A =>* A.

(lambda x A)(B) => [B/x]A.             % beta rule

A(B) => C(B) :- A => C.                % evaluation within
                                       % a subterm
A(B) => A(C) :- B => C.

lambda x A => lambda x B :- A => B.

lambda x A(x) => A :- x not_free_in A.  % eta rule (optional)
```

This file can be compiled to object code by executing

```
qc -c lambda.ql
```

Below is a session with the Qu-Prolog interpreter showing the result of sample queries to the (compiled) lambda evaluator.

First, we start the interpreter and load the compiled code.

```
$ qp
Qu-Prolog Version 6.0
```

```
| ?- load(lambda).

yes
```

Instead of loading the compiled code the source file could have been consulted using `[lambda]` or `consult(lambda)`.

Next, some simple lambda reductions. The first does a single application of the beta rule. This example illustrates that when Qu-Prolog encounters a substitution operator in a term, in this case `[x1/x]x`, it automatically seeks to evaluate it - in this case to `x1`.

```
| ?- (lambda x x)(x1) =>* R.

x = x
x1 = x1
R = x1
```

The next example also makes a simple application of the beta rule. However this example illustrates that two object variables `x2` and `x` are not automatically assumed to have values which are distinct object-level variables. Recall that `x2` and `x` are actually meta-variables which range over object-level variables. Intuitively they may take values which are either distinct object-level variables or the same object-level variable. Hence the following example does not fully evaluate the substitution in the term `[x1/x](x2(x))` which is generated by the beta rule.

```
| ?- (lambda x x2(x))(x1) =>* R.

x = x
x2 = x2
x1 = x1
R = ([x1/x]x2)(x1)
```

The next example is similar to the previous example except that the object variables `x2` and `x` are constrained to have distinct values by the `x` `not_free_in x2` assertion. The added constraint causes full evaluation of the substitution.

```
| ?- x not_free_in x2, (lambda x x2(x))(x1) =>* R.

x = x
x2 = x2
x1 = x1
R = x2(x1)

provided:

x2 not_free_in [x]
x not_free_in [x2]
```

Next we see an example of what can happen when a term containing schematic variables is matched against patterns in rewrite rules.

```
| ?- (lambda x A)(B) =>* R.
```

This query goes into infinite recursion with the variables `A` and `B` instantiated by patterns in the rules. For such applications one-sided unification is often required. Qu-Prolog can restrict the form of unification used by 'freezing' all variables in selected terms, thus preventing them from being instantiated. The next example illustrates this. It is an alternative form of the previous example, in which all variables in the term to be reduced are frozen.

```
| ?- Term = (lambda x A)(B), freeze_term(Term), Term =>* R.

Term = (lambda x A)(B)
x = x
A = A
B = B
R = [B/x]A
```

Since all the variables in `Term` are frozen, they cannot be instantiated when applying the one-step reduction rules.

As we have seen, the program for reducing lambda terms given above is essentially the same as its specification. It is perhaps not surprising therefore that this program is somewhat inefficient. After finding some subterm to reduce, it starts the reduction process again from the top-level term.

The alternative program below is more efficient but more complicated. It implements a form of 'eager' evaluation. The program reduces terms bottom-up and, wherever possible, avoids re-reducing terms it has already processed. It also avoids building copies of structures when none of its substructures can be reduced. The `IsReduced` variable serves as a flag to determine if any reduction has taken place. It is instantiated to `true` if a reduction occurs and otherwise remains uninstantiated. As a further optimization the program applies only a single-step beta reduction to a term $(\lambda x A)(B)$ after evaluation of $\lambda x A$ and B, unless B is also a lambda abstraction.

```
A =>* Result :-
        reduce(A, B, IsReduced),
        (   IsReduced == true
            ->
                simplify_term(B, Result)
            ;
                Result = B
        ).

reduce(A(B), Result, IsReduced) :-
        reduce(A, C, IsReduced),
        reduce(B, D, BIsReduced),
        IsReduced = BIsReduced,   % either is reduced
        reduce_application(C, D, Result, IsReduced),
        !.
reduce(lambda x A, Result, IsReduced) :-
        reduce(A, B, IsReduced),
        IsReduced == true,
        !,
        Result = lambda x B.
reduce(X, X, _).
```

```
reduce_application(lambda x F, A, Result, true) :-
        !,% beta reduction
        (   quant(A)            % A is a lambda abstraction
            ->
                reduce([A/x]F, Result, _)
            ;
                Result = [A/x]F
        ).
reduce_application(F, A, Result, IsReduced) :-
        IsReduced == true,
        Result = F(A).
```

To finish this section we present a type check/inference program for lambda terms. The lambda terms considered here also include atoms (that is, the above recursive definition of lambda terms is extended by adding the clause: each atom is a lambda term). Prolog variables are used to represent polymorphic types.

For the untyped lambda calculus, the program given below can be thought of as an implementation of goal directed proof in a type-assignment system [8], where the first two clauses correspond to 'arrow elimination' and 'arrow introduction' rules respectively.

For the typed lambda calculus, the program can be used, for example, as an implementation of type inference in an implementation of Huet's unification algorithm for typed lambda terms [9].

```
?- op(800, xfx, ~>).         %  for function types
% type checker/evaluator
% TypeAssign is an open list used to associate basic terms -
% vars, obvars and atoms with their type.
% Note that the type of each bound variable is added to the front of
% the TypeAssign list to provide a local context for typing the body
% of each quantified term.

type(A(B), Y, TypeAssign) :-
```

```
        !,
        type(A, X ~> Y, TypeAssign),
        type(B, X, TypeAssign).
type(lambda x A, T ~> TA, TypeAssign) :-
        !,
        type(A, TA, [x^T|TypeAssign]).
type(X, TX, TypeAssign) :-% basic term
        in_type(X^TX, TypeAssign).

% in_type(X^Tx, TypeAssign) is true iff Tx is the type
% assigned to X in TypeAssign.

in_type(X^Tx, TypeAssign) :-    % not there
        var(TypeAssign),
        !,
        TypeAssign = [X^Tx|_].  % instantiate open list
in_type(X^Tx, [Y^Ty|_]) :-  % found
        X == Y,
        !,
        Tx = Ty.
in_type(X^Tx, [_|TypeAssign]) :-
        in_type(X^Tx, TypeAssign).
```

Here is an example query to the typing program.

```
| ?- type((lambda x a(x))(b(x1)),T,L).

x = x
x1 = x1
T = T
L = [a ^ (A ~> T), b ^ (B ~> A), x1 ^ B|C]
```

Note that `x1` is given a type in `L`, because `x1` is free, whereas the bound variable `x` is not given a type.

4 A simple theorem prover

In this section we present a simple Natural Deduction style interactive theorem prover capable of carrying out schematic proofs in classical predicate calculus. This example is influenced by a similar example presented by Felty [10]. The complete program is given in the `examples` directory.

In this prover `-->` represents 'turnstile'. The hypotheses on the left hand side of `-->` are represented by a Prolog list, and the current subproblems are represented by a Prolog list.

Here is the database of rules used in the prover.

```
/*
 * Database of ND rules
 *
 *  E.G.          [Gamma --> A, Gammma --> B]
 *     (and_i)   -----------------------------
 *                      Gamma --> A and B
 */

rule(discharge(N), Gamma --> A, [[] --> true]) :-
        nth_item(N, A, Gamma).
rule(and_i, Gamma --> A and B, [(Gamma --> A), (Gamma --> B)]).
rule(or_i1, Gamma --> A or B, [Gamma --> A]).
rule(or_i2, Gamma --> A or B, [Gamma --> B]).
rule(imp_i, Gamma --> A => B, [[A|Gamma] --> B]).
rule(neg_i, Gamma --> ~A , [[A|Gamma] --> false]).
rule(all_i, Gamma --> all x A, [Gamma --> A]) :-
        x not_free_in Gamma.
rule(ex_i, Gamma --> ex x A, [Gamma --> [T/x]A]).
rule(false_i, Gamma --> A, [Gamma --> false]).

rule(and_e(N), Gamma --> C, [[A,B|NewGamma] --> C]) :-
        nth_and_rest(N, A and B, Gamma, NewGamma).
rule(imp_e(N), Gamma --> C, [NewGamma --> A,
                             ([B|NewGamma] --> C)]) :-
        nth_and_rest(N, A => B, Gamma, NewGamma).
```

```
rule(or_e(N), Gamma --> C, [[A|NewGamma] --> C,
                            ([B|NewGamma] --> C)]) :-
        nth_and_rest(N, A or B, Gamma, NewGamma).
rule(neg_e(N), Gamma --> C, [(NewGamma --> A),
                        ([false|NewGamma] --> C)]) :-
        nth_and_rest(N, ~A, Gamma, NewGamma).
rule(all_e(N), Gamma --> C, [[[T/x]A | Gamma] --> C]) :-
        nth_item(N, all x A, Gamma).
rule(ex_e(N), Gamma --> C, [[A|NewGamma] --> C]) :-
        nth_and_rest(N, ex x A, Gamma, NewGamma),
        x not_free_in NewGamma,
        x not_free_in C.

rule(modus_ponens, Gamma --> C,
     [(Gamma --> Lemma), ([Lemma|Gamma] --> C)]).
```

The predicate nth_and_rest/4 is used to extract both the n'th element of a list and the list without this element. Note that the use of this predicate and the not_free_in/2 predicate provide a way of constraining the application of rules.

The kernel of the prover uses Qu-Prolog's implicit parameters to store the state of the proof. Implicit parameters provide a form of backtrackable destructive assignment with logical semantics and are accessed via the predicates ip_set/2,3 and ip_lookup/2,3. Access to the proof state is intended to be through the predicates of the kernel.

Here is the definition of the kernel.

```
/* Create a new proof state */
new_proof(Name, T) :-
        ip_set(statement, theorem(Name, T)),
        ip_set(proof_state, [([] --> T)]).

/* End a proof */
end_proof  :-
        ip_lookup(proof_state, []),
        ip_lookup(statement, theorem(Name, T)),
        save_theorem(Name, T).
```

```
/* End a proof branch */
end_branch :-
        ip_lookup(proof_state, [([] --> true)|R]),
        ip_set(proof_state, R).

/* Apply inference rule to 1st problem of the proof state */
trans(Rule) :-
        ip_lookup(proof_state, [HeadProblem|OtherProblems]),
        rule(Rule, HeadProblem, NewHead),
        incomplete_retry_delays,
        append(NewHead, OtherProblems, Problems),
        ip_set(proof_state, Problems).

/* Look up the 1st problem of the proof state */
head_problem(P) :-
        ip_lookup(proof_state, [P|_]).

/* Look up all the problems of the proof state */
all_problems(P) :-
        ip_lookup(proof_state, P).

/* Extract the name and statement of the theorem */
statement(T) :-
        ip_lookup(statement,T).

/* Save the theorem in the database */
save_theorem(Name, Theorem) :-
        collect_vars(Theorem, Vars),
        retry_delays,
        collect_constraints(Vars, Distinct, NFI, Others),
        list_to_conj(Distinct, true, C1),
        list_to_conj(NFI, C1, C2),
        list_to_conj(Others, C2, Conjunction),
        assert((theorem(Name, Theorem) :- Conjunction)).
```

Note that in the application of each rule a call to `incomplete_retry_delays` is made. In some cases, the application of a rule may cause a delayed unification problem to be introduced. We have taken the view for this prover that such problems should be solved immediately, if possible. The call to `incomplete_retry_delays` uses heuristics in order to solve such unification problems. The code for this is given in the `examples` directory of the Qu-Prolog release.

Now that the rules and kernel are defined, users are able to write tactics and user interfaces that use the 'methods' of the kernel.

The following simple interface is supplied with the code in the examples directory. The predicate `display_head_problem` displays the first unproved problem (if there is one) with the hypotheses above the line and the goal below the line, and `no branches left` if there are no unproven problems.

```
get_command(C) :-
        auto,
        nl,
        display_head_problem,
        nl, nl,
        write(':: '),
        readR(C).

prove(Name, T) :-
        new_proof(Name, T),
        freeze_vars(T),
        get_command(C),
        proof_interpreter(C).

proof_interpreter(quit).
proof_interpreter(end_proof) :-
        end_proof, !.
proof_interpreter(show_constraints) :-
        !,
        show_constraints,
        get_command(NewC),
        proof_interpreter(NewC).
proof_interpreter(undo) :-
        !, fail.
proof_interpreter(hint) :-
```

```
        !,
        hint,
        get_command(NewC),
        proof_interpreter(NewC).
proof_interpreter(C) :-
        call(C),
        get_command(NewC),
        proof_interpreter(NewC).
proof_interpreter(C) :-
        writeR(C),
        write(' failed'),
        nl,
        fail.
proof_interpreter(_) :-
        get_command(NewC),
        proof_interpreter(NewC).
```

The `prove`/2 predicate starts a proof by using `new_proof`/2 of the kernel, freezes variables of the statement and starts the proof interpreter. It is typically the case that, when a user is proving a schematic theorem, the intention is not to instantiate schematic variables of the statement. The call to `freeze_vars`/1 prevents variables from the statement of the theorem from being instantiated during the proof.

The calls to `readR` and `writeR` are used to maintain a connection between the schematic variables of the proof and the variables appearing in the input/output by associating variable names with variables. This mechanism provides users with the ability to interact with schematic variables appearing in the proof.

Note that a call is made to `auto` in `get_command`. This is a simple tactic for applying rules. Here is its definition.

```
auto :-
        heuristic_table(Code, Message),
        call(Code),
        !,
        write(auto:Message),
```

```
        nl,
        auto.
auto.

heuristic_table(end_branch, end_branch).
heuristic_table(trans(discharge(N)), discharge(N)).
heuristic_table(trans(all_i), all_i).
heuristic_table(trans(and_i), and_i).
heuristic_table(trans(imp_i), imp_i).
heuristic_table(trans(neg_i), neg_i).
heuristic_table(trans(imp_e(N)), imp_e(N)).
heuristic_table(trans(or_e(N)), or_e(N)).
heuristic_table(trans(and_e(N)), and_e(N)).
heuristic_table(trans(ex_e(N)), ex_e(N)).
```

To finish this section we present two proof attempts, each in two different styles.

```
| ?- prove(th1, all x1 ex x2 A => ex x2 all x1 A).
auto : imp_i

1:  all x1 (ex x2 A)
---------------------------------------------------
ex x2 (all x1 A)

:: ex_i.
auto : all_i

1:  all x1 (ex x2 A)
---------------------------------------------------
[B/x2, x0/x1]A

:: all_e(_).
auto : ex_e(1)
auto : discharge(1)
auto : end_branch
```

```
no branches left

:: end_proof.

x1 = x1
x2 = x2
A = A

provided:

x2 not_free_in A
x2 not_free_in [x1]
x1 not_free_in [x2]
```

In order for this to be a theorem, `incomplete_retry_delays` has added constraints that state that `x1` and `x2` should be different and that `x2` should not occur free in `A`. A more sophisticated prover could be written that detected if any constraints involving variables of the statement of the theorem have been added by `incomplete_retry_delays` and in this case cause failure of that attempted unification. If this test is added the above proof would not succeed.

```
prove(th2, ex x2 all x1 A => all x1 ex x2 A).
auto : imp_i
auto : all_i
auto : ex_e(1)

1:  [x0/x2](all x1 A)
---------------------------------------------------
[x3/x1](ex x2 A)

:: all_e(_).

1:  [x0/x2, B/x1]A
2:  [x0/x2](all x1 A)
---------------------------------------------------
```

```
[x3/x1](ex x2 A)

:: ex_i.
auto : discharge(1)
auto : end_branch

no branches left

:: end_proof.

x2 = x2
x1 = x1
A = A
```

This proof, on the other hand, succeeds without constraining variables of the statement.

We now present attempts at proofs of the above theorems but using a different notation.

```
prove(th3, all x1 ex x2 P(x1,x2) => ex x2 all x1 P(x1, x2)).
auto : imp_i

1:  all x1 (ex x2 P(x1, x2))
---------------------------------------------------
ex x2 (all x1 P(x1, x2))

:: x1 not_free_in P, x2 not_free_in P, x1 not_free_in x2.

1:  all x1 (ex x2 P(x1, x2))
---------------------------------------------------
ex x2 (all x1 P(x1, x2))

:: ex_i.
auto : all_i

1:  all x1 (ex x2 P(x1, x2))
---------------------------------------------------
```

```
P(x0, A)

:: all_e(_).
auto : ex_e(1)

1:  P(B, x3)
2:  all x1 (ex x2 P(x1, x2))
---------------------------------------------------
P(x0, A)

:: show_constraints.


x2 not_free_in [x1]
x1 not_free_in [x2]
x0 not_free_in [x3]
x3 not_free_in [x0]
x3 not_free_in B
x0 not_free_in A
x3 not_free_in A
x1 not_free_in P
x2 not_free_in P
x0 not_free_in P
x3 not_free_in P
```

For the notation used above, it is usual to consider `x1` and `x2` to be not free in `P` and to consider that `x1` and `x2` represent different variables. Such constraints are added at the beginning of the proof.

Note that this proof cannot be completed. For example, an application of the discharge rule would fail because the constraint `x3 not_free_in A` prevents `P(B, x3)` and `P(x0, A)` from unifying. The proof therefore correctly fails.

```
| ?- prove(th4, ex x1 all x2 P(x1, x2) => all x2 ex x1 P(x1, x2)).
auto : imp_i
auto : all_i
auto : ex_e(1)
```

```
1:  [x0/x1](all x2 P(x1, x2))
---------------------------------------------------
[x3/x2](ex x1 P(x1, x2))

:: x1 not_free_in P, x2 not_free_in P, x1 not_free_in x2.

1:  [x0/x1](all x2 P(x1, x2))
---------------------------------------------------
[x3/x2](ex x1 P(x1, x2))

:: ex_i.

1:  [x0/x1](all x2 P(x1, x2))
---------------------------------------------------
P(A, x3)

:: all_e(_).
auto : discharge(1)
auto : end_branch

no branches left

:: end_proof.

x1 = x1
x2 = x2
P = P

provided:

x1 not_free_in P
x2 not_free_in P
x2 not_free_in [x1]
x1 not_free_in [x2]
```

This proof succeeds without further constraining variables.

5 Multiple Threads and Communication

Qu-Prolog 8.10 has built-in support for multiple threads of Prolog computation. Thread execution is controlled by a scheduler that is responsible for time-slicing, signal handling and managing blocking and unblocking of I/O and messages. Threads within a single Qu-Prolog process carry out independent computations but share the static code area and the dynamic database.

The Qu-Prolog library contains predicates for creating and deleting threads, for symbolically naming threads, and for controlling thread execution. These predicates are described in the reference manual.

Qu-Prolog threads, possibly in different processes or even on different machines, can communicate with each other in two ways. One way is via the use of sockets and TCP/IP. This is a very low-level form of communication and is provided as a way of communicating with pre-existing internet services, such as HTTP and FTP servers. Details of this form of communication is given in the reference manual.

The other form of communication uses Pedro. Pedro is a subscription/notification server that also supports peer-to-peer communication.

In order for a client to communicate using Pedro, it must first connect. After that the client can subscribe to notifications of a certain form or send notifications. For details of this the reader is referred to the Qu-Prolog and Pedro reference manuals.

For peer-to-peer messages the client must first register a name for itself. The name must be unique for that machine. Once it has registered a name it can send peer-to-peer messages to other agents.

When a Qu-Prolog process is invoked with a `-A` *process-name* switch, the process first connects to Pedro and then registers this name. After that the process is able to use the peer-to-peer message send and receive predicates to communicate with other registered processes. Upon termination the process deregisters itself and disconnects.

Each peer-to-peer message has three components: the actual message; the address of the thread the message is intended for; and the sender address. In Qu-Prolog, the sender address is managed internally and so the user does not need to worry about it when sending a message.

In Qu-Prolog the addresses are called *handles* and are terms of the form

```
ThreadID:ProcessName@MachineAddress
```

where ThreadID is the name of the thread, ProcessName is the name of the process (as registered with Pedro) and MachineAddress is either the machine name or machinne IP address. The thread part of the message can be elided. This will be treated as meaning thread0 by a Qu-Prolog process. Non Qu-Prolog clients will typically ignore the thread part of the message unless sending to a particular Qu-Prolog thread. If the address is for a process on the same machine then @MachineAddress can be elided and further, if the address is for the same process, then :ProcessName can be elided.

The special address self refers to the current thread.

The identity of a thread is an atom representing the symbolic name of the thread. A thread is named at creation time and it can change its name at any time.

Qu-Prolog 8.10 uses two collections of predicates for Pedro communication. The first group uses predicates that provide a close approximation to the message send and receive functions of the Pedro API. The second group takes a more high level approach and is built on top of the first group.

The first example below gives an example using the first group. The process below reads incoming messages, displays each message together with the sender and to addresses on standard output and forwards each message to the to address. This program is in the file router_monitor.ql in the examples directory of the release.

```
main(_) :-
    router_monitor.

router_monitor :-
    repeat,
    ipc_recv(Msg , From),
    ( Msg == quit
      ->
        true
```

```
    ;
      Msg = to(RealMsg, To),
      From = FThread:FProcess@FMachine,
      To = ToThread:ToProcess@ToMachine,
      write('Message: '), write(Msg),
      write_term_list([nl, wa('From: '), w(FThread), tab(3),
                       wa(FProcess), tab(3), wa(FMachine)]),
      write_term_list([nl, wa('To: '), w(ToThread), tab(3),
                       wa(ToProcess), tab(3), wa(ToMachine)]),
      nl,
      ipc_send(from(RealMsg, From), To),
      fail
  ).
```

If this file is compiled as

```
qc -o router_monitor router_monitor.ql
```

and then invoked as

```
router_monitor -A router_monitor
```

then it will process the incoming messages as described above.

The call `ipc_recv(Msg, From)` blocks waiting on a message and when a message arrives it succeeds with `Msg` and `From` instantiated to the message and the sender handle.

Finally, the call `ipc_send(from(RealMsg, From), To)` sends the message on to the to field of the received message and sets the from field to the sender of the received message.

Another process can send a message to the monitor as follows.

```
ipc_send(Message, thread0:router_monitor@some_machine,
         Some_Handle).
```

The handle above is composed of the thread ID `thread0` (the main thread – and in this case the only thread), a process name `router_monitor`, a machine address `some_machine`. If this is the handle for a running monitor process then this message will be displayed by the monitor process and forwarded to `Some_Handle`.

The remaining examples of this section use multiple threads and the high-level communication predicates.

The following example uses the high-level communications layer to implement most of the Linda model [12] for interprocess communication. The Linda model implementation has two components: the Linda server (in `linda_server.ql` in the examples directory); and support for Linda clients (in `linda_client.ql`).

The Linda server stores Linda tuples (as Prolog facts) and responds to the following requests from clients to add, remove and read tuples from the tuple space.

- `out(T)` – Add the tuple `T` to the tuple space.
- `in(T)` – Block until a matching tuple is added to the tuple space, then remove the matching tuple and return it to the client.
- `rd(T)` – The same as `in(T)` except that the matching tuple is not removed.
- `inp(T)` – The same as `in(T)` except that it returns a failure rather than blocks if no matching tuple is found.
- `rdp(T)` – The same as `rd(T)` except that it returns a failure rather than blocks if no matching tuple is found.

The Linda server should be started using the switch `-A linda_server_process` to name the process.

The main thread of the linda server is initialized with a call to `linda/0`. This call names the main thread and starts a repeat-fail loop that waits for connect messages from clients and forks a thread for each client. Each forked thread is responsible for processing requests from its client.

```
linda :-
    thread_set_symbol(linda_server_thread),
```

```
    linda_loop.

linda_loop :-
    repeat,
    connect <<- RtAddr,
    thread_fork(_, linda_thread(RtAddr)),
    fail.
```

The goal `connect <<- RtAddr` is a high-level message receive which blocks until a `connect` message is received. Note that the sender address is passed as an argument to the goal to be executed in the forked thread. This informs the forked thread of its client address.

The forked thread first sends a message to its client to inform the client of the address of the forked thread. It then enters a loop to process client requests.

```
linda_thread(A) :-
    connected ->> A,
    thread_loop(A).

thread_loop(A) :-
    repeat,
    message_choice
    (
    out(T) <<- A ->
        assert(T),
        inserted ->> A
    ;
    in(T) <<- A ->
        thread_wait_on_goal(retract(T)),
        ok(T) ->> A
    ;
    rd(T) <<- A ->
        thread_wait_on_goal(clause(T, true)),
        ok(T) ->> A
    ;
    inp(T) <<- A ->
        (   retract(T)
```

```
            ->
                ok(T) ->> A
            ;
                fail ->> A
        )
    ;
    rdp(T) <<- A ->
        (   clause(T, true)
            ->
                ok(T) ->> A
            ;
                fail ->> A
        )
    ;
    disconnect <<- A ->
        thread_exit
    ;
    notify(T) <<- A ->
        thread_fork_anonymous(_, notify_thread(T,A))
    ),
    fail.

notify_thread(T,A) :-
        notified >> A,
        thread_wait_on_goal(clause(T, true)),
        notify_match(T) ->> A.
```

The `connected ->> A` call sends the `connected` message to its client.

Requests are processed using the `message_choice/1` predicate. The argument to `message_choice` is a term that uses the if-then-else construct of Prolog where the test is replaced by a message/address pattern - see the reference manual for a full description.

The call to `message_choice/1` above blocks until a message is received that matches one of the message patterns. When such a message is received, the message is removed from the message queue and the goal associated with the first matching message pattern is called.

Note that all message patterns in this example have the client address in the address field and so this thread only responds to messages from its client.

A call to the predicate `thread_wait_on_goal`/1 blocks until some change has been made to the dynamic database. When the call unblocks the supplied goal is called and if the call succeeds then the call to `thread_wait_on_goal` succeeds. Otherwise it reblocks.

Although not part of the Linda model, a client can send a `notify` message to the Linda server thread. This is included as another example of the use of threads. This request causes another thread to be forked that waits until a term matching the supplied pattern has been asserted, at which point the thread notifies the client. It is the clients responsibility to check for the `notify_match` message.

The file `linda_client.ql` in the `examples` directory gives predicates that support interaction with the linda server. A client connects to the server by using the `linda_connect` predicate defined below.

```
linda_connect :-
    connect ->> linda_server_thread:linda_server_process,
    connected <<= A,
    thread_symbol(TID),
    assert(idaddr(TID,A)).
```

A call to this predicate first sends a `connect` message to the main thread of the server and then waits for a `connected` response. The sender of the `connected` message is the thread created to interact with this client. The identity of the client thread together with the address of the server thread is then asserted. The address is needed so that the client can send Linda queries to the server thread (see below) and the thread identity is also asserted so that a single process can contain several Linda client threads – each interacting with their own server thread.

A client can send an `in` query, for example, to the server using the following predicate.

```
linda_in(T) :-
    thread_symbol(TID),
```

```
    idaddr(TID,A),
    in(T) ->> A,
    ok(T) <<= A.
```

A call to this predicate first looks up the address of the server thread responsible for this client and then sends an `in` message and waits for a response from the server thread.

The goal `ok(T) <<= A` searches the incoming message buffer for a message that unifies with this pattern. The call blocks until a matching message arrives.

The `examples` directory also includes the file `db.ql` that gives another simple example of the use of threads and messages to implement a shared database.

The following example from the examples directory (`consumer.ql` and `producer.ql`) shows how subscriptions and notifications can be used to implement a produce/consumer model. Note that there can be any number of producers and any number of consumers.

```
% The consumer

start :-
    pedro_connect,
    pedro_subscribe(producer(T), true, _),
    message_loop.

message_loop :-
    repeat,
    M <<- _,        % get the first message
    write(M), nl,
    M = producer(quit).


% The producer

start :-
    pedro_connect.
```

```
send(Msg) :-
    pedro_notify(producer(Msg)).
```

References

[1] Holger Becht, Anthony Bloesch, Ray Nickson and Mark Utting, Ergo 4.1 Reference Manual, Technical Report 96-31, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, November 1996.

[2] Mark Utting, Ray Nickson and Owen Traynor, Theory Structuring in Ergo 4.1, *Computing: The Australasian Theory Symposium*, volume 18(3) of *Australian Computer Science Communications*, Michael E. Houle and Peter Eades eds, pages 137–146, 1996.

[3] P. J. Robinson and M. J. Walters, Qu-Prolog 6.0 reference manual Technical Report 00-21, Software Verification Research Centre, The University of Queensland, St. Lucia, QLD 4072, Australia, December 2000.

[4] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, 3rd, revised and extended ed., Springer-Verlag, Berlin, New York, 1987.

[5] Leon Sterling, and Ehud Shapiro, *The Art of Prolog*, The MIT Press, Cambridge, Massachusetts, 1994.

[6] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T., Sjoland and J. Widen, SICStus Prolog User's Manual, SICS technical report T93:01, January 1993.

[7] P. Nickolas and P. J. Robinson, The Qu-Prolog Unification Algorithm: Formalisation and Correctness, *Theoretical Computer Science* 169 (1996) 81-112.

[8] J.Roger Hindley and Jonathan P. Seldin, *Introduction to Combinators and λ-Calculus*, London Mathematical Society Student Texts, Cambridge University Press, Cambridge, 1986.

[9] G.P.Huet, A Unification Algorithm for Typed λ-Calculus, *Theoretical Computer Science* 1 (1975) 27-57.

[10] A. Felty, Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language, *Journal of Automated Reasoning*, Vol. 11, No. 1, pp. 43-81, 1993.

[11] F. G. McCabe ICM Reference Manual. Fujitsu Labs of America, http://www.nar.fla.com/icm/manual.html, 1999.

[12] N. Carriero and D. Gelernter Linda in context. *CACM*, 32(4),1989, pp. 444–458.