Qu-Prolog 5.0: The Tool for implementing Rational Agents

SVRC, University of Queensland
Dept. of Computing, Imperial College

September 20, 1999

1 Prolog Language

Qu-Prolog is a logic programming language primarily intended for the implementation of interactive theorem provers. As such, it contain all the features expected of a standard Prolog: data base manipulation, meta-level predicates, etc. plus additional predicates which facilitate the manipulation of unifiers and related aspects.

2 Multi-threading

Version 5.0 offers **multi-threading** as a feature that totally differentiates it from the previous implementations of Qu-Prolog. Every Qu-prolog process has the ability to launch **threads** thus providing concurrency in the execution.

The dynamic database of Qu-Prolog is shared by all the threads hence providing a blackboard mechanism for inter-thread communication.

3 Inter-Thread Communication

To exploit the multi-threading more fully, the language has been enhanced with general Inter-Thread Communication (ITC) facilities, with the following features:

Communication between threads and machines is supported. The ITC system supports communication only between threads running under the same Qu-Prolog process, as well as between threads in different processes, and also between threads running on processes on different machines.

The diagram in Figure 1, shows communication amongst threads executing on several Qu-Prolog 5.0 processes on two machines. It is the later aspect, communications amongst several machines, that really exploits the power of a computer network.

The ITC programming interface is simple and generic. The predicates and programming idioms used in communication are identical wherever the intended recipient of a message happens to be executing.

Applications programs can use abstract concepts. The operations and objects that the system presents to the programmer aren't tied to the lower level operations and data structures used to implement them. [1]

[1] Direct TCP level predicates are provided by Qu-Prolog 5.0.

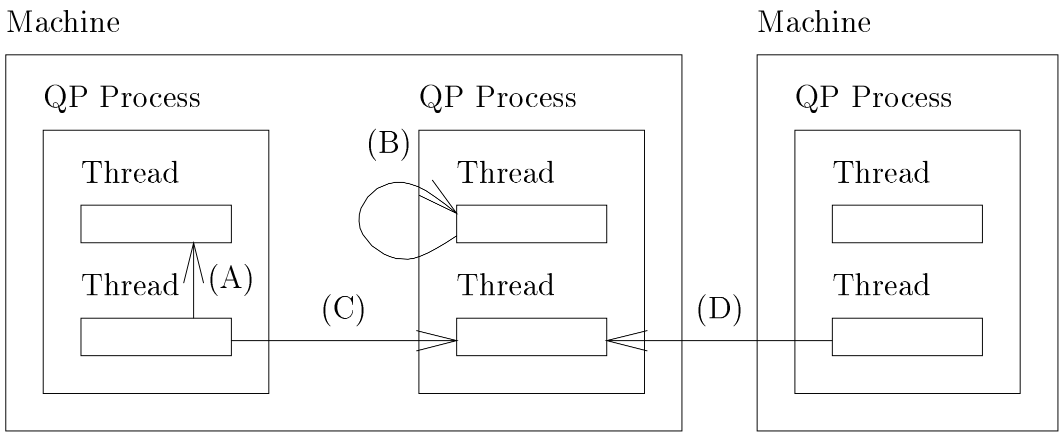

Figure 1: Messages (A–D), threads, processes and machines.

Messages are arbitrary Qu-Prolog 5.0 terms. No restrictions on the form of the messages have been made. In keeping with Qu-Prolog 5.0 input/output (I/O) predicates, the ability to remember the names of variables used in message terms is also supported.

Message access may be nondeterministic. Threads aren't constrained by fixed "first-in-first-out" access to incoming messages.

Address information is integrated into the messaging system. In the Qu-Prolog ITC system, the sending of messages entails the passing of information about where the message originated and where a reply should be sent.

Minimal "garbage" is created. Message passing doesn't leave unused or unneeded data in any of the Qu-Prolog 5.0 global data areas (e.g. the code area or the record database).

Messages are private. Messages intended for a thread are only visible to that thread.

3.1 Example: A Stack Implementation

The figure 2 shows a stack implemented as a thread, using message choice to handle the 'methods' of the stack. An initial handshake between the main thread and the (newly created) stack thread allows real address structures to be used. Message sending is handled by the meta-level operator `<<` while reception of messages is handled by `<<=` (asynchronously) and `<<` (synchronously).

Note that multiple stack instances can be created and used simultaneously (as shown by the Qu-Prolog program of Figure 3); they are referenced by their address structures. Addresses have three components: thread, process and machine identifiers, as stated by the syntax <*thread-id*>:<*process-id*>@<*machine-id*>.

```
?- [itc_package].

init_stack_object(Address) :-
    thread_new(ThID),
    thread_set_goal(ThID, stack_object),
    thread_resume(ThID),
    start >> ThID,           % Handshake from parent
    ready <<= from(_,Address).     %

stack_object :-
    start <<= from(_,Address),     % Handshake from stack object
    ready >> Address,              %
    stack_methods([]).

stack_methods(Stack) :-
    msg_choice
    ( push(Item) << _ ->           % push an item
          stack_methods([Item|Stack])
    ; pop << from(_,ReplyTo) ->    % pop an item
          Stack = [Item|Stack1],
          pop(Item) >> ReplyTo,
          stack_methods(Stack1)
    ; empty << from(_,ReplyTo) ->  % test if stack is empty
      ( Stack = [] -> empty(true) >> ReplyTo
      ; empty(fail) >> ReplyTo
      ),
      stack_methods(Stack)
    ; kill << _ -> thread_exit     % terminate stack object
    ).
```

Figure 2: A simple stack object.

```
rev(L1, L2) :-
    init_stack_object(StackAddress),
    push_all(L1, StackAddress),
    pop_all(L2, StackAddress),
    kill >> StackAddress.

push_all([], _).
push_all([H|T], StackAddress) :-
    push(H) >> StackAddress,
    push_all(T, StackAddress).
pop_all([], StackAddress) :-
    empty  >> StackAddress,
    empty(Val) <<= _,
    Val = true, !.
pop_all([H|T], StackAddress) :-
    pop >> StackAddress,
    pop(H) <<= _,
    pop_all(T, StackAddress).
```

Figure 3: Example use of stack object.

4 Object Oriented Programming

The most recent addition to Qu-Prolog 5.0 is a layer (similar to a DCG) to allow OOP; this has the expected behaviour regarding inheritance and method over-ridding.

Figure 4 shows the declaration of an object **person**, its attributes declared in the **state** and its methods defined under **clauses**; figures 5 and 6 show two sub-classes derived form **person** while figure 7 shows another sub-class inheriting attributes and methods from various classes (note that attribute **likes/1** is purposely not inherited here).

The *= operator allows to retrieve the value of a specific attribute, **new** creates a new instance of an object (i.e. a new thread), ? extracts the attribute of an object instance, ∧∧ is for message sending and **self_assert** (together with **self_retract**) is a meta-level predicate that manipulates part of the process database as private for this object instance.

5 Contact Points

SVRC, University of Queensland

- Peter Robinson
  email: pjr@csee.uq.edu.au
  http://www.svrc.it.uq.edu.au/pages/Peter_Robinson.html

- Richard Hagen
  email: richard@it.iq.edu.au,
  http://svrc.it.uq.edu.au/~richard/home.html

Department of Computing, Imperial College

- Keith Clark
  email: klc@doc.ic.ac.uk
  http://www.doc.ic.ac.uk/~klc

- Silvana Zappacosta
  email: sza@doc.ic.ac.uk
  http://www.doc.ic.ac.uk/~sza

```
class person
state [ surname, age := 0, sex, child/1, parent/1 ]
clauses {
new_child( C, Sx ) :-
    surname *= Sn,
    myid( Me ),
    new( person,
         [ surname := Sn, sex := Sx, { parent( Me ).} ],
         C, O ),
    self_assert( child( O ) ).
get_married( Sp ) :-
    mystate( St ),
    become( married_person,
            [ spouse := Sp | St ].
likes( O ) :-
    child( O ).
}.
```

Figure 4: Declaration of a person object.

```
class married_person isa person
state [spouse]
clauses {
init :-
    spouse *= Sp,
    myid( Me ),
    Sp?spouse *= Me -> true;
        Sp ^^ get_married( Me ).
likes( T ) :-
    spouse *= T.
get_divorced :-
    state( St ),
    remove( spouse, St, NSt ),
    become( person, NSt ).
}.
```

Figure 5: Declaration of a married-person object.

```
class worker isa person
state [ occupation, salary, place ]
clauses {
init :-
    myid( Me ),
    place *= Pl,
    Pl ^^ new_worker( Me ).
likes( O ) :-
    place *= Pl,
    Pl?worker( O ).
}.
```

Figure 6: Declaration of a worker object.

```
class house_wife isa [ worker - [likes/1], married_person ]
state [ salary := 0, occupation := master_of_all_trades ].
```

Figure 7: Example of multi-inheritance.