`QuLog` User Guide

Keith L. Clark and Peter J. Robinson

September 17, 2021

Contents

1 Introduction

`QuLog` is a flexibly typed hybrid logic, functional, string processing language with an imperative action language sitting on top. It is higher order in that named relations, functions and actions can be passed as arguments and returned as values. The declarative kernel was developed to complement our robotic agent programming language `TeleoR`, to provide a `TeleoR` programmed robotic agent with its reasoning capability. `TeleoR` was inspired by and builds upon Nilsson's Teleo-Reactive Procedures language.

The relations, functions and actions are all defined by sequences of conditional and unconditional rules. A special class of *dynamic* relations are defined solely by facts. Only action rules can call primitive actions of the language, such as file I/O, thread creation, inter-process communication, and updates of the dynamic relations. More generally action rules can be used to program multi-threaded communicating agent *behaviour*. `QuLog`'s static relation and function definitions then comprise the agent's *knowledge*. The dynamic facts record its changing *beliefs*.

It is a fully integrated language in that function calls can appear as or inside arguments to relation and action calls, and relational queries can be used as guards of function and action rules rules. It has sets as a separate data type from lists with convertors for mapping between the two data types. Both can be created using `Term::Query` comprehension expressions.

Sets are manipulated using union, intersection and difference operators. Lists are manipulated as in Prolog but also using non-deterministic pattern matching. Similar pattern matching is used for string processing as a precursor to DCG parsing. An 'in' primitive can be used to access elements of sets, lists and characters in strings.

A `QuLog` agent application typically comprises a set of independent multi-threaded agent processes each of which has an its own knowledge and beliefs. The beliefs may be shared, but only if explicitly communicated or placed in a communal repository. Each agent thread has a name, which is unique within the agent, and executes some action call. It can atomically query the agent's current belief facts using its fixed knowledge. It can also atomically update these facts using `remember` and `forget` actions.

Changing the shared memory of dynamic facts is the main way that an agent's threads communicate. However a thread can also asynchronously communicate with other internal agent threads using send and receive actions, with receiver and sender identified by their unique thread names. Each thread has one message queue that only it can access. A sent message, which is a `QuLog` data term, is placed at the back of the destination thread's

message queue. A `receive` primitive allows a thread to search the message queue from front to back for a message satisfying some test.

The same message send action can be used to send a copy of a message to a thread in another agent process. In this case the destination thread must be identified by a term that gives not only the thread name, but the agent process name. If the agent is running on a host different from the one on which the sending agent process is running, the different host must also be identified using a destination address of the form `ThreadName:AgentName@HostName`. Such inter-agent communication routes the message via a `Pedro` communication server. `Pedro` is companion free software downloadable from:

`https://staff.itee.uq.edu.au/pjr/HomePages/PedroHome.html`.

When an agent is launched it typically connects and registers its name, which must be unique for the host on which it is running, with a `Pedro` server. The server may be located anywhere on the internet of hosts reachable from the agent's host machine. A message sent to a thread in another `QuLog` process, which has registered with the same `Pedro` server, is routed to the other `QuLog` process which puts it at the back of the message buffer for the named thread. It is put into a default message handling thread if the destination address for the message has the form `AgentName@HostName`, with no thread name.

`Pedro` also supports message routing of unaddressed notifications using lodged subscriptions. This allows multi-casting of a message to any agent that has lodged a subscription with `Pedro` that *covers* the message term. A `QuLog` application can also receive and send MQTT notifications routed via an MQTT publish/subscribe server.

Debugging is done by putting a `watch` on any number of relations, functions and actions. This invisibly transforms their code to display each call, the input and output bindings of the unification or match of the call with each rule that can be used, and optionally the instantiated body of the rule before it is used. An `unwatch` command reverses the code transformation.

2 `QuLog` Type system

`QuLog` relations and actions are both *typed* and *moded*. The modes determine which argumements must be given as *ground terms* (terms with no unbound variables) in calls, and which arguments will be ground if the call succeeds. Its functions are just typed, as all their arguments must be given as ground terms. Functions always return ground term values. The types and modes

are so that we can guarantee at compile time that there will be no runtime failures or errors due to wrongly typed arguments, or due to arguments that *must* be given not having been computed before the call, particularly calls to `QuLog` primitives. We need this to make `QuLog` (and `TeleoR`) a serious agent and robotic programming language. The result of these constraints is that all but a handful of primitives of `QuLog`, and *most* program defined relations, ground all their arguments if they succeed.

2.1 Type flexibility and runtime type checking

To retain some of the flexility of Prolog we have *union* (aka *disjunctive*) types. For example, using the union type:

```
def int_atom ::= integer || atom
```

we can specify that a relation `r` accepts either integers or atoms in some input argument position. We can then use the runtime type test primitive to determine which type of value has been given in the call. Typically we would have two rules for `r(int_atom,...)`, one with the test `type(A1,int)` and the other with the test `type(A1,atom)`, where `A1` is the variable in the first argument position of the rule head. In `QuLog` run-time type tests can be done for any primitive type, any program defined type, as well as any higher order type.

2.2 Sub-type relation and modes

All `QuLog` data types are partially ordered by a sub-type relation. For example, each of the system data types `nat` (the non-negative integers), `int` (all integers), `num` (all numbers) is a sub-type of all the types that follow it in the sequence. At the top of the data sub-type tree is `term`. Every value of a primitive type, or a program defined type, is a `term`, and every value of the `term` type is either a value of a system type or a defined type of the `QuLog` program being queried.

A sub-type value may be given as an argument where a super-type is specified. It may also be returned as a value in lieu of a super-type value.

2.3 Ground terms and modes of use

A `term` that contains no variable is said to be `ground`. `[1,a,-6,"hi"]` is a ground list term of type `list(integer || atom || string)`. The terms `[1,a, U,"hi"]` and `[1,a,-6,..L]` are not ground.

The type checker ensures that all function calls are given ground arguments of their declared argument types, and that they will return a ground value of the declared value type. It checks that each relation and action call will have each argument an unbound variable, or a term of the declared type. It also checks that the argument will be a ground term *before* the call if moded `!` (ground input), and will be ground *after* the successful evaluation of the call, if moded `?` (ground output). Only if moded with `??` (non-ground output) is there no check beyond the type check. A `@` moded argument is one where the argument will not be further instantiated by the call.

Relations and actions (defined by one or more rules) have a sub-type relationship based on this data sub-type relationships and their modes of use, specified by their moded argument types. For functions the sub-type relation just depends on argument and value types as there is only one mode of use - all arguments ground and ground value returned. As an example of the role of modes, suppose the higher order argument type specifies the relation given as an argument will only be used to check integer values. We can pass in a relation that checks *or generates number* values. A more flexible check or generate relation that handles numbers can be used where only integer values will be tested.

`QuLog` is a fully integrated LP/FP language in that function calls can appear as or inside arguments to relation calls, and relational queries can be used as tests in function rules, and in set expression arguments of function calls.

The compiler does type *checking* of function, relation and action definitions, assisted by type *inference* for data terms and variables. We believe that type declarations, linked with mode of use declarations for relations and action procedures, are very useful *active* documentation of the program. Also, because we have union types and sub-types, type inference on code could be very complex in some cases. Type inference on data terms will assign the term the minimum type in the sub-type lattice.

This user guide assumes familiarity with Prolog and higher order functional programming, as in languages such as Haskell or Scala.

In Section 3 we briefly discuss syntax and in Section 4 we discuss type declarations. In Section 5 we discuss the relation rule part of `QuLog` , in Section 6 we discuss the function rule part and in Section 7 we discuss the action rule part of `QuLog`. Lastly, in Section 8 we give examples of using the `QuLog` interpreter.

All `QuLog` and `TeleoR` program files use the `.qlg` extension. The `QuLog` examples below are nearly all from the file

`examples/introduction/qlexamples.qlg`

3 Syntax

`QuLog` does *not* have an operator precedence syntax and its syntax is not extensible. As in Prolog, a functor or predicate is written immediately next to its tuple of arguments as in `p(...)`. There is a collection of reserved words that are used for the builtin operators. There is *no need* to use a full stop followed by a white space character, as in Prolog, to separate the relation rules (aka clauses), function and action rules, type definitions and type declarations. Instead we borrowed an idea from Python and made what is a normal program layout format for a Prolog program - each clause starting on a new line - a syntax requirement.

All the above program statements *must* begin at the left end of a new line. Each can be continued over several lines where all but the first line is *indented* by at least one space or tab. Starting rules at the left end of a newline, and indenting a continuation of a rule by several spaces, or one tab, is normal Prolog program layout. Our indentation indicator for continuation of a rule encourages this, making programs more readable. However, as a gesture towards Prolog programmers, including ourselves, a full stop followed by a newline, or spaces and a newline, *may* also be used at the end of a statement. It is *ignored* by the `QuLog` parser. Even if you use fullstops as terminators, you *cannot* put two clauses or rules on the same line of the program file.

Apart from requiring a predicate or functor to be adjacent to its `(...)` bracketed arguments, and treating a space or tab at the beginning of a new line as a continuation marker, `QuLog` is tolerant of spaces. They should be used to aid readability of the program. To aid in editing `QuLog` programs we have supplied an emacs mode and a simple Python/Tkinter based editor (quled) that uses tab to provide readable code layout and uses syntax highlighting.

As in Prolog, alphanumeric names beginning with an upper case letter or underscore _, or underscore on its own, are variables. To make such a name an `atom` (aka symbol), or the name of a relation or function, it can be singly quoted as in `'Peter'`, `'Father_of'`, `'Fact'`.

In contrast, an alphanumeric name that begins with a lower case letter, which can contain under-scores, is an `atom` and is used to denote individual things or names of relations, functions or actions. Surrounding such a name with single quotes has no effect at all, and they will be dropped when the atom is displayed.

One difference in syntax between `QuLog` and Prolog is that `QuLog` allows zero arity compound terms such as `empty()`.

The syntax for sets is a sequence of ground terms surrounded by parentheses such as `{1,2,3}`.

Lists and list patterns are as in Prolog but we also allow `[H,..T]` as equivalent to `[H|T]` and `[H,..]` as equivalent to `[H|_]`.

4 Type Declarations

As discussed in the reference manual `QuLog` comes with a collection of builtin types that include the following:

```
nat, int, num, atom, string, atomic, term,
list(T), set(T)
```

where the `T` is a variable used as a type variable providing polymorphic types. These types can be used in the construction of user defined types.

There are two kinds of type declarations: enumerated, union and macro types; and code types.

4.1 Enumerated Types

Enumerated type declarations take the form
`def` *LHS* `::=` *RHS*
where *LHS* is either an atom, the name of the type, or a compound term with distinct type variables as arguments and *RHS* is an enumeration or a range of integers.

So, for example,

```
def man ::= roger | tom | bill | alan | graham | keith |
    sam | fred
def woman ::= june | mary | rose | penny | sue | helen |
    veronica
def noun ::= "boy" | "fox" | "girl" | "ball" | "man" |
    "woman" | "lady"
```

declares `man`, `woman` and `noun` to be the names of types. The right hand sides enumerate the possible values of these types.

It's also possible to have enumerations of numbers as above but also another common enumeration for integers is to use a range as in

```
def age ::= 0 .. 110
def digit ::= 0 .. 9
```

In all the above enumerations, all the values are atomic. It it also possible to have compound terms (constructors) in enumerations as in

```
def tree(T) ::= empty() | tr(tree(T),T,tree(T))
def noun_phrase_tree::= np(article,noun_exp_tree)
```

The first declaration above is a recursive polymorphic type declaration. All the arguments in the constructors on the right hand side must be types (or type variables listed as arguments on the left hand side).

4.2 Macro and Union Types

Macro type declarations are similar to the above and take the form
`def` *LHS* `==` *RHS*
but in this case *RHS* is either a type or a union of types.

A type union is written as
T1 `||` ... `||` *Tn*.

So, for example, following from the above examples

```
def human == man || woman
def int_tree == tree(int)
```

In the first example we are declaring `human` as a macro for the union of the types `man` and `woman` and so `man` and `woman` are both subtypes of `human`.

4.3 Code Types

There are two different uses of code types (for relations, actions and functions): for the declaration of the code; and for specifying an argument of a relation, fuction or action as a code type.

Instead of using the key word `def` we use `rel`, `fun` and `act` as in the examples below.

```
rel descendant_is(!human,?human)
fun num_children(human) -> nat
act do_parse(!string, ?parse_tree)
```

The arguments of relation and action declarations are moded types. If the mode is missing it defaults to `!`. The arguments of a function declaration are types - modes are not required as the arguments are always ground.

By listing multiple type declarations for a single relation, function or action we are actually declaring an intersection type as below (for a user declared version of `append`).

```
rel app(!list(!T), !list(!T), ?list(?T)),
    app(?list(?T), ?list(?T), !list(!T)),
    app(!list(??T), !list(??T), ?list(??T)),
    app(?list(??T), ?list(??T), !list(??T)),
    app(??list(??T), ??list(??T), ??list(??T))
```

Below are some examples of using code types as arguments of declarations of other code types.

```
fun mapF((fun(T1) -> T2), list(T1)) -> list(T2)
rel mapR(!rel(!T1,?T2), !list(T1), ?list(T2))
fun curry(fun(T1,T2) -> T3) -> fun(T1) -> fun(T2) -> T3
```

We can also declare a special case of `rel` for dynamic relations using `dyn` as below.

```
dyn child_of(human,human)
```

These dynamic relations simply store ground facts in the *Belief Store* and the intention is to be able to query thse facts and so, for these declarations we have types (not moded types) as the implicit mode is `?` as opposed to `!` for functions.

A special case of dynamic relation is what we call "global values" - they can either be of type `int` or `num`. The declaration also initializes the value. The current value can be accessed using a `$` prefix and the value can be modified by using the operators `:=`, `+:=` and `-:=` as in the examples below.

```
int a:=0
num b:=0.0

act inc_a(?int)
"Increment the global value a and return the incremented value"
inc_a(N) :: N = $a+1 ~> a +:= 1
```

In declaring relations and functions (typically that depend on the belief store) we can use `mrel` and `mfun` instead. The `m` stands for memoization. Please refer to the section entitled "Automatic Memoization of Functions and Relations" in the reference manual for details.

4.4 Default Arguments for Code Types

Sometimes it is the case, particularly for actions, that some arguments take default values. In making such declarations we can use the keyword `default` followed by a value of the required type. In such cases all the arguments with default values must appear at the end and, in use, if an argument is given a value other than its default then all previous arguments with defaults have to be also given a value (usually its default value). Below are examples from the builtin (system) declarations.

```
act write_list(TermList : !list(@term),
               Stream : !stream_type default stdout)
"Write TermList to Stream."
act connect_to_pedro(Host : !atom default localhost,
                     Port : !int default 4550)
"Connect to the pedro server on Host using Port."
```

In the second example if the Pedro server on this machine was started using, say port 5000, then we would need to use

```
connect_to_pedro(localhost, 5000)
```

4.5 Doc Strings

Again, we borrow the idea of doc strings from Python where triple quoted strings can accompany definitions. For us we use single quoted strings that immediately follow a type declaration as below (and in the examples above).

```
dyn age_of(H:human,A:age)
"H is a human, A is an age"

fun fact(N:nat) -> nat
"Returns the factorial of N"

def man ::= roger | tom | bill | alan | graham | keith |
            sam | fred
"The allowed men in the application"
```

It is quite common in doc strings to refer to the arguments. We support this by allowing the type declaration to be preceeded by a variable with that variable being used in the doc string (as in the examples above).

This is purely to aid doc strings and has no semantic implications for the declarations.

In a similar way that doc string in Python are shown when the user uses `help` we use `types` for displaying user declarations or `stypes` for displaying system declarations.

4.6 Constraints on Type Declarations

The constraints on type declarations are discussed in detail in the reference manual. From the point of view of these constraints we consider the names of code types as though they are type names. Here we simply list them for convenience.

- If enumeration of atomics overlap then one must be completely contained in the other.
- Overlaps between constructor enumerated types are not allowed.
- The union of parameterized (polymorphic) types are not allowed.
- Enumerator values (or their functors in the case of constructor enumerations) can not be used as the names of types and visa a versa.

The following example, from the reference manual, shows how union types can be used when we want overlapping types.

```
def digit ::= 0..9
def range12 ::= 10..20
def range21 ::= -10..-1
def range1 == digit || range12
def range2 == range21 || digit
```

Now `range1` and `range2` are essentially the same as the enumerated types `0..20` and `-10..9` without breaking the above constraints.

The more strict constraint on constructor enumerations seems to be excessively strong but we believe that it will not come up often in practice. If it does we can define two constructor enumerations with different constructors and then write a converter relation as in the following example.

```
def tree(T) ::= empty() | tr(tree(T),T,tree(T))
def tree2(N,L) ::=
leaf(L) | none() | node(tree2(N,L), N, tree2(N,L))
```

```
rel tree_to_tree2(!tree(T), ?tree2(T, T)),
    tree_to_tree2(?tree(T), !tree2(T, T))
tree_to_tree2(empty(), none())
tree_to_tree2(tr(empty(), V, empty()), leaf(V)) :: true
tree_to_tree2(tr(L, V, R), node(L2, V, R2)) <=
    tree_to_tree2(L, L2) &
    tree_to_tree2(R, R2)
```

5 QuLog Relation Rule Subset

The definition of a given relation is made up of a contiguous sequence of relation rules. Relation rules take one of the following forms.

RuleHead
RuleHead `<=` *RuleBody*
RuleHead `::` *RuleGuard*
RuleHead `::` *RuleGuard* `<=` *RuleBody*

Normally *RuleHead* is a simple compound term whose functor is the declared name of the relation and with arity that matches the arity of the corresponding type declaration for the relation. The exception is when we have a higher-order function whose value is a relation. In this case the function is defined via relation rules as in the following example.

```
fun curryR(rel(T1,??T2)) -> fun(T1) -> rel(??T2),
  curryR(rel(T1,?T2)) -> fun(T1) -> rel(?T2),
  curryR(rel(T1,!T2)) -> fun(T1) -> rel(!T2)

curryR(Rel)(X)(Y) <= Rel(X,Y)
```

In either case the head arguments are not allowed to include function calls.

Both *RuleGuard* and *RuleBody* are conjunctions of relation calls but that can also include the `exists` and `forall` quantifiers. The EBNF in the appendix of the reference manual gives a more exact description of the allowed syntax.

`QuLog` does not include cut and so we use the guards to commit to rules and so a rule like

```
p(X) :: q(X) <= r(X)
```

is semantically the same as the equivalent Prolog rule

```
p(X) :- q(X), !, r(X)
```

If we wanted to commit to rule with no body we need to write something like

```
p(X) :: true
```

which is semantically the same as the equivalent Prolog rule

```
p(X) :- !
```

The examples file `examples/introduction/qlexamples.qlg` contains many rules for relations (and functions and actions). We discuss a cross section of relation definitions in this file below. The definitions of auxilary relation used below can be found in this file.

First consider

```
rel person(?human,?gender,?age)
person(H,male,A) <= isa(H,man) & age_of(H,A)
person(H,female,A) <= isa(H,woman) & age_of(H,A)
```

This is a straightforward definition whose rule bodies are simple conjuctions. The point of interest here is the call on `isa` in these rules. This, along with `type`, is a way of doing run time type checking. The main difference is that `type` is purely a type checker while `isa`, only used on enumerated or union of enumerated types, can be used as a generator. Notice the the first argument of `person` is of mode `?` and so it is expected that the relation should be able to generate instances of `H`. If `H` is a variable at the time of call then `isa(H,man)` will, on backtracking, bind `H` to each value in the enumerated type. On the other hand, if `H` is given it will simply check the type of `H`.

The next example uses both `forall` and `exists`.

```
rel only_has_adult_children(?human)
only_has_adult_children(P) <=
    child_of(_,P) &
    forall C (
            child_of(C,P) =>
                exists A (
                        person(C,_,A) &
                        A>20
                        )
            )
```

The point of this example is to understand the reasons why there is an initial call on `child_of` and then another call in left hand side of the body of `forall` and why there is an `exists` in the right hand side of the body. Both of these have to do with the groundedness of variables.

For both `forall` and `exists` we insist that any global variables are ground at the time of call. Further any variables on the right hand side of the body of `forall` that are not global variables (other than _) are ground by the left hand side. Also, the set of global and quantified variables in a rule are required to be disjoint.

In this example, if we omit the initial call `child_of(_,P)` then `P` in `child_of(C,P)` is a global variable that is not guaranteed to be ground. Secondly, if we omit the `exists` then `A` would be treated as a global variable.

The call to `forall` in this case can be read as "for a given human `P` then for every child `C` of `P`, `C` has an age `A` that is greater than 20".

Since every occurrence of _ is a unique variable it can be though of as implicitly existentially quantified. Indeed, we can modify the above code to replace the inner _ by an explicitely quantified variable as below.

```
rel only_has_adult_children2(?human)
only_has_adult_children2(P) <=
    child_of(_,P) &
    forall C (
            child_of(C,P) =>
                exists G, A  (
                        person(C,G,A) &
                        A>20
                        )
            )
```

The above examples were used to illustrate the constraints on global and quantified variables. The variant below is simpler and slightly more efficient and avoids the need to an underscore or extract quantified variable.

```
rel only_has_adult_children3(?human)
only_has_adult_children3(P) <=
    child_of(_,P) &
    forall C (
            child_of(C,P) =>
                exists A  (
                        age_of(C, A) &
                        A>20
```

```
                        )
            )
```

Negation has similar constraints - global variables inside `not` need to be ground at call time. The following examples show the use of underscore and `exists` inside `not` in order to satisfy the constraint.

```
rel childless(?human)
childless(P) <=
    isa(P, human) &
    not child_of(_,P)


rel has_no_siblings(?human)
has_no_siblings(P) <=
    isa(P, human) &
    not exists Parent, Sibling (child_of(P, Parent) &
                                child_of(Sibling, Parent) &
                                P \= Sibling)
```

The next example gives an example of set comprehension.

```
rel children_are(?human,?set((age,human)))
children_are(P, Cs) <=
    isa(P,human) & Cs = {(A,C) :: child_of(C,P) & age_of(C,A)}
```

This set comprehension produces a set of age, human tuples. If we replace the parentheses by square brackets we would get a list comprehension (of type `list((age,human))`.

Set and list comprehension have the same groundedness constraints as `forall` and `exists` and explains why the call `isa(P,human)` is before the comprehension. In this case `A` and `C` are quantified variables.

We could use set comprehension to produce a slightly less inefficient version of `childless` as below.

```
  rel childless(?human)
  childless(P) <=
      isa(P, human) &
       {} =  {(A,C) :: child_of(C,P) & age_of(C,A)}
```

Prolog programmers will see set and list comprehension as a variant of findall, and indeed it is. Python programmers might see this is as similar to a list comprehension - we chose this syntax to make it look more like Python so as to be more readable than using something like findall and also to make it more clear what the quantified variables are.

Python programmers who are not familiar with Prolog might now be wondering about iterators and generators. We don't need to do anything special as backtracking naturally produces one solution at a time as can be seen in the comparative examples below.

We start with a simple Python example - turning a list into an iterator. One way to do this is as follows

```
def list2gen(lst) :
  for x in lst:
      yield x
```

Then we could use this to produce a more complex iterator, for example, with the following expression.

```
map(lambda x: x**2,
    filter(lambda x: x < 8, list2gen([1,6,2,8,5])))
```

We could then use `next` to get one value from this iterator at a time with a `StopIteration` exception raised when there are no more values.

In `QuLog` we could simply write

```
X in [1,6,2,8,5] & X < 8 & Y = X**2
```

and `Y` would be instantiated to the first solution. On backtracking `Y` would be instantiated to the next solution. When there are no more solutions we would simply get failure.

Below is a simple example using `range` to compare an iterator approach in Python with a backtracking approach in `QuLog`.

In Python we might write

```
((x,y) for x in range(5) for y in range (5) if x < y)
```

whereas in `QuLog` we might write

```
range(X, 0, 5) & range(Y, 0, 5) & X < Y & Z = (X, Y)
```

For those not familiar with Python, the above Python expression is a generator and we can ask for the next value as in the following example in the Python interpreter.

```
>>> g = ((x,y) for x in range(5) for y in range (5) if x < y)
>>> next(g)
(0, 1)
>>> next(g)
(0, 2)
>>> next(g)
(0, 3)
```

Using `next` is very similar to backtracking to get another solution in the corresponding `QuLog` call.

In Python, replacing the brackets with square brackets will produce the list while in `QuLog` we would write the equivalent list comprehension:

```
[(X, Y) ::range(X, 0, 5) & range(Y, 0, 5) & X < Y]
```

We could use list comprehension for list processing as in the following example.

```
[X**2 :: X in [0,1,2,3,4] & p(X)]
```

where `p` is a relation and so `p(X)` is a test on `X`. However, because this uses `findall`, it is less efficient than using straightforward list processing.

With this in mind we have supplied the system defined functions `map`, `filter` and `filter_map` and so the above example is more efficiently written as

```
filter_map(p, square, [0,1,2,3,4])
```

where `square` has been defined as the square function.

The next example shows a use of a run-time type check. The intention is to produce the sum of all the numbers that are in the suplied list of terms.

```
rel add_nums_of_list_of_any_term(!list(term), ?num)
add_nums_of_list_of_any_term([],0)
add_nums_of_list_of_any_term([N | Rest], Total)  ::
    type(N,num) <=
        add_nums_of_list_of_any_term(Rest, RTotal) &
        Total = RTotal+N
add_nums_of_list_of_any_term([ _Any | Rest], Total) <=
    add_nums_of_list_of_any_term(Rest, Total)
```

Since the first argument is a list of terms then any individual element of the list may or may not be a number. The type check `type(N,num)` as a guard in the second rule guarantees `N` will be a (ground) number for the body of the rule. Note that the type check is in the guard and not part of the body. If it was in the body then, on backtracking the third rule would also be chosen when `N` is a number.

The `QuLog` unification `Total = RTotal+N` will automatically evaluate the expression argument `RTotal+N`. This is because, unlike Prolog, `QuLog` evaluates any function calls in arguments of relation calls before evaluating the relation call.

The next example shows the power of string processing in `QuLog`. Normally `++` is the builtin string concatination function (`fun string ++ string -> string`) but when used on the right hand side of `=?` it turns into a (backtracking) string splitter. We present two versions of a "tokenizer" for English sentences with a sentence terminator. The first version uses backtracking to find words and the second uses regular expressions.

```
rel words(!string,?list(string))
words(Str,[WStr]) :: Str =? WStr::word(WStr) ++ E::endchar(E)
words(Str,[W|Words]) ::
    Str =? W::word(W)++Seps::seps(Seps)++RStr::words(RStr,Words)

rel words2(string, ?list(string))
words2(Str,[WStr]) :: Str  =? WStr/"\\w*" ++ _End/"[.?!]"
words2(Str,[W|Words]) ::
    Str =?
        W/"\\w*" ++
        _Seps/"([,;:]?\\s+)|(\\s+)" ++
        RStr::words2(RStr,Words)
```

In the first rule for `words` we are asking to find a way of splitting the given string `Str` into two strings `WStr` and `E` such that `WStr` is a word and `E` is a sentence terminator. If we can find such a split then `WStr` is the last word of the sentence. In the second rule we see if we can split the sentence into a word, followed by a string of space characters, followed by the remaining sentence (which is recursively processed).

As an example of how this backtracking string matching works consider how the sentence `"Hello  world!"` is processed by this relation.

The first rule is tried but there is no way of splitting up the string into a word followed by a sentence terminator. The second rule is therefore tried.

First `W` is instantiated to `"H"` (which is a word and so satisfies the constraint `word(W)`). However no initial part of `"ello  world!"` is a separator (whitespace). This causes backtracking and `W` is now instantiated to `"He"`. This backtracking continues until `W` is instantiated to `"Hello"`. Now `Sep` is instantiated to `" "` and we get backingtracking until `Seps` is instantiated to `"  "`. At this point the remainder of the string, i.e. `"world!"`, is recursively processed. The first rule (after some internal backtracking) will match this string.

The second version uses regular expressions. Instead of using `::` constraints we are using a slash followed by a string representing a regular expression. Note the need to double up backslashes. Regular expression matches are deterministic and so they do not create choicepoints.

In general each argument of `++` is one of

VarOrString
VarOrString `::` *Call*
VarOrString `/` *REString*
VarOrString `/` *REString* `:: Call`

Often *Call* is simply a test on *Var* (which is instantiated to a substring) as in the first rule of `words` but can be any arbitrary valid call as in the second rule of `words` where it is a recursive call.

As well as backtracking string processing, `=?` also works on lists where normally `<>` is the list appending function but, as with `++`, on the right hand side of `=?` it turns into a backtracking list splitter. The following example, on backtracking, splits a list of numbers at pairs that are in order.

```
rel split_on_ordered_pair(!list(num), ?list(num), ?num, ?num,
                          ?list(num))
split_on_ordered_pair(Lst, LeftLst, V1, V2, RightLst) <=
    Lst =? LeftLst <> [V1, V2] :: (V1 < V2) <> RightLst
```

In general each argument of `<>` is one of

VarOrListPattern
VarOrStringListPattern `::` *Call*

6 `QuLog` Function Rule Subset

The definition of a given function is made up of a contiguous sequence of function rules. Function rules take one of the following forms.

RuleHead `->` *Result*
RuleHead `::` *RuleGuard* `->` *Result*

The first of the above forms is a special case of the second where *RuleGuard* is `true`. A consequence of this is that all the rules are committed rules, i.e. function calls are deterministic. This is one of the main differences between function rules and relation rules - relations are nondeterministic. Another difference is that relation calls can fail whereas functions are not allowed to fail. Instead, if no rules match the function call then a `no_matching_function_rule` exception it raised. This is flagging that we are outside of the domain of the function.

Consider the following definition of the factorial function.

```
fun fact(N:nat) -> nat
"Returns the factorial of N"
fact(0) -> 1
fact(N) :: N1 = N-1 & type(N1,nat) -> N*fact(N1)
```

The type check call is required as the type checker deduces the type of `N1` is `int` after the subtraction. Can we avoid this type check call? We could if we change to the code below.

```
fun fact2(N:int) -> int
fact2(0) -> 1
fact2(N) -> N*fact2(N - 1)
```

However, we can give `fact2` a negative number and this will lead to infinite computation. In an attempt to fix this we could add a guard to the second rule as follows.

```
fun fact3(N:int) -> int
fact3(0) -> 1
fact3(N) :: N > 0 -> N*fact3(N - 1)
```

Now the problem is that if we give this function a negative number we will raise a `no_matching_function_rule` exception. This is because `fact3` is not a total function on its domain (`int`).

For the next example we consider converting between lists and ordered trees. Note that this code does not produce balanced trees.

```
def tree(T) ::= empty() | tr(tree(T),T,tree(T))
```

```
fun tree2list(tree(T)) -> list(T)
tree2list(empty()) -> []
tree2list(tr(LT, V, RT)) -> tree2list(LT) <> [V] <> tree2list(RT)

fun list2tree(list(int)) -> tree(int)
list2tree([]) -> empty()
list2tree([H|T]) -> add2tree(H, list2tree(T))

fun add2tree(TT, tree(TT)) -> tree(TT)
add2tree(T, empty()) -> tr(empty(), T, empty())
add2tree(T, tr(L, V, R)) :: T @< V -> tr(add2tree(T, L), V, R)
add2tree(T, tr(L, V, R)) -> tr(L, V, add2tree(T, R))
```

Recall that `<>` is the list concatenation function. We use the generic term ordering `@<`.

We finish this section by giving three examples of higher order functions - i.e. functions that take fuctions or relations as arguments. The example file contains several more such examples.

```
fun mapF((fun(T1) -> T2), list(T1)) -> list(T2)
mapF(_, []) -> []
mapF(_F, [H|T]) -> [F(H)|mapF(F, T)]

fun curry(fun(T1,T2) -> T3) -> fun(T1) -> fun(T2) -> T3
curry(F)(X)(Y) -> F(X,Y)

fun curryR(rel(T1,??T2)) -> fun(T1) -> rel(??T2),
    curryR(rel(T1,?T2)) -> fun(T1) -> rel(?T2),
    curryR(rel(T1,!T2)) -> fun(T1) -> rel(!T2)
curryR(Rel)(X)(Y) <= Rel(X,Y)
```

You will note that in each example we have a compound term with a variable functor. The type check allows this if the variable functor is known to be a ground code type as is the case in these examples.

In the second and third examples the head of the rule is a compound term whose function is itself a compound term. The third function definition is unusual - it is declared as a function but the rule is a relation rule. This is because `curryR(Rel)(X)` is a one argument relation and so when we apply this to its argument we get a relation call and so we need a relation rule.

Please note the complex type declaration for `curryR`. This intersection type allows different modes for the second argument that is reflected in the modes of the resulting relation. This maximizes the flexibility of use of this function.

7 `QuLog` Action Rule Subset

Action form the procedural component of `QuLog`. The definition of a given action is made up of a contiguous sequence of action rules. Action rules take one of the following forms.

RuleHead `~>` *RuleBody*
RuleHead `::` *RuleGuard* `~>` *RuleBody*

In some cases we might want to write a rule that has no action in the rule body. In this case `{}` represents the null action.

As with function rules, actions are deterministic and cannot fail - in this case a `no_matching_action_rule` exception will be raised. As discussed in the reference manual, we constrain the use of `?` and `??` moded arguments for actions as follows.

1. Non-variables are not allowed in `?` and `??` moded arguments.
2. Variables that are `!` moded are not allowed in `?` and `??` moded arguments.
3. Variables that have already occurred in the body of a rule are not allowed in `?` and `??` moded arguments.

For example, the call

```
read_term([X])
```

violates the first constraint. The following rule with the given declaration violates the second constraint.

```
act a(!term)
a(X) ~> read_term(X)
```

Given the declaration

```
act read2(??term, ??term)
```

the call

```
read2(X, X)
```

violates the third constraint as the second `X` occurs earlier (in the first argument).

Typical uses of actions in `QuLog` are to send and receive messages, update the *Belief Store*, manage threads and read and write to streams.

We start by giving examples of updating the *Belief Store*.

```
act new_child(!human, !age, !human, !human)
new_child(C, A, M, F) ~>
    remember([child_of(C,M), child_of(C,F), age_of(C,A)])

act birthday(!human)
birthday(P) :: person(P, _, A) & Z = A+1 & type(Z, age) ~>
        forget_remember([age_of(P, A)], [age_of(P, Z)])
        birthday(P) ~> write_list([P,
              ' is not a person or would have an invalid age'])
```

In the first example we use `remember` to add facts to the *Belief Store* (similar to `assert` in Prolog). In the second we replace an old fact by an updated fact using `forget_remember`. This is similar to doing a combination of `retract` and `assert` in Prolog.

Each call to `remember`, `forget` and `forget_remember` is done atomically, no other thread has either read or write access to the *Belief Store* until the update is complete. Also when such a call completes a single update to the timestamp on the *Belief Store* is done and this is used in the `TeleoR` system to trigger re-evaluation. Furthermore, before the atomic update is complete, any memoized relation or function (see the reference manual) is checked to see if their memoized data needs to be cleared and if so, clears the data.

The next example is an action variant of the relational parser in the examples file. The problem with the relational version is that if the sentence did not parse then the relational parser would simply fail. For the action version below we give feedback output. Because output is an action then this version of the parser needs to be an action.

```
act do_parse(!string, ?parse_tree)
do_parse(Str,PT) ~>
    to_words(Str,Wrds);
    write_list(["Word list: ",Wrds,nl_]);
```

```
    check_dict(Wrds);
    write_list(["All words in dictionary",nl_]);
    to_parse_tree(Wrds,PT)

act to_words(!string, ?list(string))
to_words(Str,Wrds) :: words(Str,Wrds) ~> {}
to_words(Str,[]) ~>
     write_list(['Cannot split into words: ',Str, nl_])

act check_dict(!list(string))
check_dict(Wrds) :: all_dict_words(Wrds) ~> {}
check_dict(Wrds) ~>
write_list(["Unknown words in: ",Wrds, nl_])

act to_parse_tree(!list(string), ?parse_tree)
to_parse_tree(Wrds,PT) :: a_parse_tree(PT,Wrds,[])  ~> {}
to_parse_tree(Wrds,parse_error()) ~>
    write_list(['Cannot parse word list: ',Wrds,nl_])
```

First note that the declaration says that the action must produce a ground term of type `parse_tree`. Because actions are not allowed to fail, we have added `parse_error()` to the type enumeration. The problem with this implementation is that if either we cannot tokenize the input or some of the tokenized words are not allowed we continue on to `to_parse_tree`.

Another option is to declare one or more user declared exceptions and have the action raise one of these exceptions when a problem occurs as in the variant below. This also allows us to terminate early (by raising an exception) as soon as a problem is discovered.

```
  def user_exception ::= cannot_tokenize() |
        unknown_words(list(string)) |
        cannot_parse(list(string))

  act do_parse2(!string, ?parse_tree)
  do_parse2(Str,PT) ~>
    try {
        to_words2(Str,Wrds);
        write_list(["Word list: ",Wrds,nl_]);
        check_dict2(Wrds);
        write_list(["All words in dictionary",nl_]);
```

```
        to_parse_tree2(Wrds,PT)
        }
    except {
        cannot_tokenize() :: PT = parse_error() ~>
             write_list(["Cannot split into words: ",Str, nl_])

        unknown_words(Wrds) :: PT = parse_error() ~>
             write_list(["Unknown words in: ",Wrds, nl_])

        cannot_parse(Wrds) :: PT = parse_error() ~>
             write_list(["Cannot parse word list: ",Wrds,nl_])
        }
```

In uses in relation rules `forall` is used as a test but in action rules it is used for iteration as in the following example.

```
act remove_child(!human)
remove_child(C) ~>
   forall P {child_of(C,P) ~> forget([child_of(C,P)])}
```

In this example, for a given input child `C`, we find each parent `P` of `C` and forget the `child_of` fact.

We complete this section by giving a brief explanation of the code for a simpler fact data server that can be updated and queried by any number of QuLog client processes. The code is included near the end of the `qlexamples.qlg` file.

Clients may update the server's facts by adding and removing facts. They may also query the facts and some of its rule defined relations. We implement the server using a "repeat/fail" approach. The examples file also contains a recursive version.

```
def message_t ::= tell(dyn_term) | deny(dyn_term)

rel may_update(??dyn_term,!agent_handle)
may_update(age_of(_,_),_)
may_update(child_of(_,_),_)
/* We use the may_update definition to restrict updates
 to certain agents. In this case all clients are
 allowed to update any age_of or child_of fact.
 Such rules may be used to allow only certain clients to update
 certain relations, even to restrict updates of certain to facts
```

```
 having certain argument values. For example we might want
 to restrict updates of child_of(_,P)  facts to an agent with
 handle P@_ - the agent for the parent P. */

% The following relation has a system type declaration
% rel allowed_remote_query_from(??rel_term,!agent_handle)
allowed_remote_query_from(age_of(_,_),_)
allowed_remote_query_from(child_of(_,_),_)
allowed_remote_query_from(person(_,_,_),_)
allowed_remote_query_from(descendant_is(_,_),_)
allowed_remote_query_from(ancestor_is(_,_),_)
/* Any agent is allowed to query without restriction all the
 above relations, but only these relations.
 We can be more restrictive by partially instantiating the
 relation call templates and/or the agent handle arguments. */


  act rf_handle_messages()
  rf_handle_messages() ~>
    fork(rf_handle_message(), Name, messages);
    set_default_message_thread(Name)

  act rf_handle_message()
  rf_handle_message() ~>
    repeat {
        try {
            receive {
                tell(Bel) from Ag ::
                   ground(Bel) & may_update(Bel,Ag) ~>
                       write_list(["Remembering: ", Bel, nl_]);
                       remember([Bel])

                deny(Bel) from _ ::
                   nonvar(Bel) & may_update(Bel,Ag) ~>
                       write_list(["Forgetting:",Bel, nl_]);
                       forget([Bel])

                %% special message pattern for query_at calls
                %% from a client
                remote_query(ID, QueryStr) from_thread AgTh ::
```

```
            nonvar(ID) & nonvar(QueryStr)  ~>
                write_list(["Agent thread ", AgTh,"
                            asked:", nl_,QueryStr, nl_]);
                %% builtin action that parses, type
                %% checks, evaluates QueryStr and
                %% returns answers to Client
                respond_remote_query(ID, QueryStr, AgTh)

        M from_thread Addr ~>
            write_list(["Invalid message ", M,
                            " from ", Addr, nl_])
        }
    }
except {
    %% All messages that are received are type checked
    %% as a term If the test fails the message is
    %% consummed and this exception is raised
    input_term_type_error(_, Err) ~>
        write_list(["Message type error: ", Err, nl_])
    }
}
```

First we declare a `message_t` type so that `tell` and `deny` terms will be accepted as valid messages. Note that the type says that both of these messages take an argument of type `dyn_term` that is suitable for remembering and forgetting.

At the top-level, calling `rf_handle_messages` will fork a thread using the root name of the thread as `messages`. Assuming no other thread is named `messages` then `Name` will be instantiated to `messages`. We set this thread to be the thread that will receive agent messages - i.e. messages sent using `to` where the sender doesn't specifier the receiver thread. The created thread will call `rf_handle_message`.

The top-level of `rf_handle_message` uses a repeat that causes the thread to repeatedly call the inner action which is a `try-except` action. We use this in case a client sends a message that is not of type term, which causes an `input_term_type_error` exception to be raised.

Inside the `try` we call the `receive` action which contains a collection of message/address patterns (with an optional guard) and an action. The semantics of the use of `receive` in this example is as follows.

The call first blocks waiting for a message to arrive. When it does it

checks that the message has type term and if not raises an exception. If it is a `tell(Bel)` message it checks if the message is ground and that the sender of the `tell` is allowed to update the `Bel` fact update by querying the `may_update` relation. If so it prints a message and remembers the sent belief. If instead it is a `deny(Bel)` message, it checks that `Bel` is a non-variable, and that the sender may do the update. If so it prints a message and forgets the belief. If the message is a `remote_query` message then a message is printed and the server parses, evaluates the query whilst checking that for each relation call `Call` in the query that the querying client is allowed to query the relation of `Call` in that way by using the `allowed_remote_query_from` rules in the server. If every call is allowed, it sends back the query answers to the client as a stream of strings. All this is done by the `respond_remote_query`. If any call in the remote query has no answers, or is not an allowed call for the client which sent the remote query, no answers are returned - a query failure.

Although not used in this case, a `receive` action can have an optional timeout as the last choice and has the form

`timeout` *Time* `~>` *Action*

If no message that matches any of the `receive` choices has arrived within *Time* seconds of the call start then *Action* will be called.

Note that the use of `ground` and `nonvar` in the first two choices are necessary as `remember` requires a ground `dyn_term` and `forget` requires a non-variable and hence a `dyn_term` pattern.

Because `receive` rejects messages that do not type check it is important that the client and server agree on types otherwise a message term that type checks on the client side might not type check on the servr side and so the message will be rejected. This suggests that the programmer should use a common ontology for both clients and server. One way to do this is to create a file that contains all declarations that are common to both clients and server and consult this file within both the client and server program files.

We now look at examples of sending client messages to the server using the `QuLog` interpreter - we look the general functionality of the interpreter in more detail in section 8. We assume both the client and server have consulted `qlexamples.qlg`, and that the server is running on the same machine as the client and has process name `server`.

```
| ?? (A :: age_of(june, A)) query_at server.
```

```
A = 23 : age

| ?? deny(age_of(june, _)) to server.

success

| ?? tell(age_of(june, 24)) to server.

success

| ?? (A :: age_of(june, A)) query_at server.

A = 24 : age
```

The first query is an example of remote querying. We are asking the server to find all solutions for the variable `A` in `age_of(june, A)`. The server returns the list of answers and the client binds `A` to the first answer and then, on backtracking, binds `A` to the next solution. The result is the same as the query `age_of(june, A)` in the server.

The second message causes the server to forget `age_of(june, 23)` while the third message causes the server to remember `age_of(june, 24)`. The final remote query confirms this change has been made.

Note that we could also use `to_thread` as below

```
| ?? tell(age_of(june, 24)) to_thread messages:server.
```

However, `query_at` is an agent message rather than a thread message - a special case of `to`.

A multi info_server example

The directory `qulog/examples/introduction/info_broker` there are program files for a simple multi sensor information server application. There are two base level sensor information servers, but there can be many more. Acting as an optional query interface to these sensor information servers is a broker agent. The broker agent and the information servers have a shared ontology of three relations describing rooms in a building. The relations give the room temperature, the open or closed status of the room's doors, and which people are in the room.

In the information servers, the rules for the common ontology relations just query dynamic relation facts. For example:

```
temperature(L,T) <=
    temp_info(L,T)
```

where `temp_info` is a dynamic relation.

The dynamic relation facts in each information server are being frequently updated by a Python sensor process. In the example you will interact with the Python sensor processes to provide the sense data, but in a real application this Python process would be harvesting readings from sensors in the rooms, sending each changed reading as a message to the information server for which it is the data provider. On receipt of the new reading message the dynamic fact recording the new reading is immediately updated by the information server. So repeated remote queries to the information server return different answers at different times, even though the queries are to 'static' rule defined relations.

The broker agent does not store sensor data. Its dynamic data comprises facts such as:

```
info_source(temperature, sensor_server1)
```

giving meta information about which sensor servers are active and may be queried about temperatures. The broker agent's rule defining `temperature` is:

```
temperature(Loc, Temp) <=
    info_source(temperature, Server) &
    temperature(Loc, Temp) query_at Server
```

This maps a query about `temperature` to remote queries to each of the sensor servers currently beleived to be an `info_source` for `temperature`, i.e. which will have temperature readings for rooms recorded as `temp_info` dynamic facts.

In the `info_broker` directory is a README file giving instructions of how to deploy and query this multi-server toy application.

Guarded actions

The action `receive` of the fact server message handling loop uses a form of guarded actions - in this case the guard is a message pattern together with a test. `QuLog` supports two other forms of guarded action actions: `case` and `wait_case`.

The first of these has the form

```
case {
     Guard1 ~>  Action1
    ...
     Guardk ~>  Actionk
    }
```

This is similar to a cascading if-then-else in Prolog. If *Guardi* is the first guard that is true then *Actioni* will be called. If no guards are true an `action_failure` exception will be raised.

The second of these has the same form and is a generalization of the `wait` action.

```
wait_case {
     Guard1 ~>  Action1
    ...
     Guardk ~>  Actionk
    }
```

The semantics is the same as for `case` except that if none of the guards are true then `wait_case` waits until the *Belief Store* has changed and then re-tests the guards. It only makes sense to use `wait_case` if the guards either directly or indirectly depend on the *Belief Store*.

As with `receive`, `wait_case` can have an optional timeout as the last choice.

8 General use of the QuLog Interpreter

The `QuLog` interpreter is essentially the same as a Prolog interpreter - queries are entered and the interpreter responds by calling the query and displaying answers. There are some differences, most notably is that the `QuLog` interpreter carries out type and mode checking on each query (in a similay way to when it checks the body of a rule).

8.1 Starting the interpreter

To start the interpreter, assuming paths have been set up, we simply need to use the command

```
qulog
```

In many uses of `QuLog` we want to use Pedro for communication and so the simplest way to do this is instead use the command

`qulog -A` *Name*

where *Name* is the name we want to give to this process. This will connect us to Pedro and tell Pedro that this is the name of this process. If that name, on this machine, is already in use interpreter will produce an error message and terminate.

In the examples below we will assume we are in the same folder as `qlexamples.qlg`

Once we have started the interpreter we will get the prompt `| ??` and we can consult the example file in the same way as in Prolog

```
| ?? [qlexamples].
```

or

```
| ?? consult qlexamples.
```

All queries, like in Prolog, are terminated by a fullstop, linefeed.

If the file contains syntax or type errors we can modify and save the file and consult again - this is really a re-consult. Note that files we consult can contain consults themselves. These files are consulted and checked before the main file is consulted. The reference manual gives details about how reconsulting works when either the main file or a sub-file is edited.

Assuming the file of interest has been successfully consulted we can then enter queries. In the `QuLog` interpreter queries are either a conjuction of relation calls or a sequence of action calls. The interpreter does not allow a mixture.

8.2 Controlling the number of answers given for a relation query

We start with a simple example (a comment has been added to the interpreter interaction to highlight a particular line).

```
| ?? age_of(P,A).

P = roger : man
A = 110 : age
...
P = tom : man
A = 26 : age
...
```

```
P = june : woman
A = 23 : age
...
P = bill : man
A = 40 : age
...
P = mary : woman
A = 40 : age
..                  %% Note .. was entered by user
P = rose : woman
A = 40 : age
...
P = penny : woman
A = 1 : digit
```

Unlike in Prolog where answers are printed one at a time and a semi-colon is used to get the next answer, in `qulog` interpreter the first five answers are printed and a `..` is used to get the next batch of answers. Notice that the interpreter displays the inferred type next to each answer term.

We can change the number of answers produced at a time as follows using the special interpreter query action `set_num_answers`. This, and several queries below, are only for use in the interpreter and is not available for use in rules.

```
| ?? set_num_answers(3).

success

| ?? age_of(P,A).

P = roger : man
A = 110 : age
...
P = tom : man
A = 26 : age
...
P = june : woman
A = 23 : age
```

We can set the number of solutions back to the default:

```
| ?? set_num_answers(5).

success
```

We can constrain the query in a couple of ways as illustrated below. Say we wanted to know the people who have ages over 39 then we could use the query

```
P = roger : man
A = 110 : age
...
P = bill : man
A = 40 : age
...
P = mary : woman
A = 40 : age
...
P = rose : woman
A = 40 : age
```

What if we didn't care about the age and didn't want to clutter the interpreter output with this information then we use either of the following queries.

```
| ?? P :: age_of(P,A) & A>39.

P = roger : man
...
P = bill : man
...
P = mary : woman
..
P = rose : woman

| ?? exists A age_of(P,A) & A>39.

P = roger : man
...
P = bill : man
...
P = mary : woman
```

```
...
P = rose : woman
```

What if we wanted the answers to this particular query to appear two at a time but didn't want to globally change the number of answers printed. The following query will do.

```
| ?? 2 of P :: age_of(P,A) & A>39.

P = roger : man
...
P = bill : man
..                          %% user input
P = mary : woman
...
P = rose : woman
..
no more solutions
```

8.3 Action calls and commands

One or more action calls (separated by ;), or a single interpreter command, may be entered in response to the query prompt. Remember the action calls are deterministic and so an entered action call sequence will give exactly one answer binding for its variables, or produce an exception.

```
| ?? do_parse2("the fat lady sings!", PT).

Word list: ["the", "fat", "lady", "sings"]
All words in dictionary
PT = s(np("the", ne("fat", n("lady"))), v("sings")) : parse_tree
success
```

The first two lines of output are produced by the action call. The last two lines are produced by the interpreter.

We can't call functions directly but we can turn them into a relational query by using an = query as below.

```
| ?? X = curry(+).

X = curry(+) : term_naming(fun(int) -> fun(int) -> int &&
                           fun(num) -> fun(num) -> num &&
```

```
                              fun(nat) -> fun(nat) -> nat)

| ?? X = curry(+)(2).

X = curry(+)(2) : term_naming(fun(int) -> int &&
                                fun(num) -> num &&
                                fun(nat) -> nat)

| ?? X = curry(+)(2)(3).

X = 5 : digit
```

Note the types in the first two examples. The answer terms are terms that name functions and the type of each of these functions are intersection types - i.e. can can be used in multiple situations.

8.4 Seeing code type declarations

We can list both system types and user types using, respectively, `stypes` and `types`. Without arguments these will list all types. We can follow these with a comma separated list of atoms. The interpreter will respond by listing all types whose name includes one of the supplied atoms. For example

```
| ?? stypes line, term.


act get_line(Line : ?string, Stream : !stream_type default stdin)
"Read Line from Stream"

act put_line(Line : !string, Stream : !stream_type default stdout)
"Write Line to Stream"

rel copy_term(Term : @term, Copy : ??term)
"Copy Term with all variables in Term replaced by fresh variables."

act read_term(??term, Stream : !stream_type default stdin)
"Unifies its argument with the next term denoted by the next
sequence of characters in the stream followed by fullstop, return."

rel string2term(String : !string, Term : ??term)
```

```
"String is a string comprising the character sequence of a QuLog
term.
Term is unified with that term."

rel term2string(Term : @term, String : ?string)
"Term is converted to String - the string representation of the term."
success

| ?? types curr.

fun curry(fun(T1, T2) -> T3) -> fun(T1) -> fun(T2) -> T3
"Curried form of F"
fun curryR(rel(T1, ??T2)) -> fun(T1) -> rel(??T2),
    curryR(rel(T1, ?T2)) -> fun(T1) -> rel(?T2),
    curryR(rel(T1, !T2)) -> fun(T1) -> rel(!T2)
fun uncurry(fun(T1) -> fun(T2) -> T3) -> fun(T1, T2) -> T3
fun uncurryR(fun(T1) -> rel(!T2)) -> rel(T1, !T2),
    uncurryR(fun(T1) -> rel(?T2)) -> rel(T1, ?T2),
    uncurryR(fun(T1) -> rel(?T2)) -> rel(T1, ?T2),
    uncurryR(fun(T1) -> rel(??T2)) -> rel(T1, ??T2)

    success

| ?? types tree2.

def tree2(N, L) ::= leaf(L) | none() |
                    node(tree2(N, L), N, tree2(N, L))
rel on_tree2(?tree_val(N, L), !tree2(N, L))
act to_parse_tree2(!list(string), ?parse_tree)
fun tree2list(tree(T)) -> list(T)
rel tree_to_tree2(!tree(T), ?tree2(T, T)),
    tree_to_tree2(?tree(T), !tree2(T, T))

success
```

Similar to `types`, we can also show user code definitions using `show` as follows. This is similar to `listing` in Prolog. Note that both the type declaration and any `"...."` quoted comment string are displayed, along with the defining rules.

```
| ?? show age_.  %  This will show defs for all code beginning with age_
```

```
dyn age_of(H : human, A : age)
"H is a human, A is an age"
age_of(roger, 110)
age_of(tom, 26)
age_of(june, 23)
age_of(bill, 40)
age_of(mary, 40)
age_of(rose, 40)
age_of(penny, 1)

success

| ?? show inc.

act inc_a(?int)
"Increment the global value a and return the incremented value"
inc_a(N) ::
    N = $a + 1 ~>
        a +:= 1


fun inc(nat) -> nat
inc(N) -> N + 1


success
```

8.5 Debugging using `watch`

We finish the section on the interpreter with a discussion on debugging. For debugging Prolog code we often use `trace`. This can be very frustrating as we will often sufer from information overload but, even worse, we might accidentally use `skip` when we should have continued to use `creep` causing us to start again.

In `QuLog` we have tried to make debugging simpler. Here is a relation definition in the `qlexamples.qlg` file.

```
rel only_has_adult_children(?human)
only_has_adult_children(P) <=
```

```
    exists C child_of(C,P) &
    forall C (
        child_of(C, P) =>
            exists A age_of(P, A) & A > 20)
```

It can be used for both finding the humans who have one of more children but all are above the age of 20, or for checking if a given human has that property.

We can watch its use by entering the command

```
| ?? watch only_has_adult_children.

success
```

If we do a `show only_has_adult_children` we will see the same single rule definition but when we query the relation we get information about its use.

```
| ?? only_has_adult_children(P).

1:only_has_adult_children(P)
  Call 1 unifies rule 1
      output none
  Rule body is:
      child_of(_, P) &
      forall C_0 (
          child_of(C_0, P) =>
              exists A_0 age_of(P, A_0) & A_0 > 20)
1:only_has_adult_children(roger) succeeded
P = roger : man

1:only_has_adult_children(P) seeking another proof
1:only_has_adult_children(roger) succeeded
...
P = roger : man  %  Answer roger given again

1:only_has_adult_children(P) seeking another proof
1:only_has_adult_children(mary) succeeded
...
P = mary : woman
```

```
1:only_has_adult_children(P) seeking another proof
1:only_has_adult_children(bill) succeeded
...
P = bill : man

1:only_has_adult_children(P) seeking another proof
1:only_has_adult_children(rose) succeeded
...
P = rose : woman
..  % ..  entered to request more answers if there are any

1:only_has_adult_children(P) seeking another proof
  no (more) proofs using rule 1 trying next rule for call 1
1:only_has_adult_children(P) no (more) proofs
no more solutions
```

Ideally we would not get `roger` given as an answer twice. The suspician is that it might be multiple solutions to the test or generate condition `child_of(_,P)` giving `P=roger` twice. The use of the anonymous variable `_` indicates we are not interested in knowing the child of `P`.

We can add a `watch` on the relation `child_of` and repeat the query:

```
watch child_of.
| ?? only_has_adult_children(P).

1:only_has_adult_children(P)
  Call 1 unifies rule 1
      output none
  Rule body is:
      child_of(_, P) &
      forall C_0 (
          child_of(C_0, P) =>
              exists A_0 age_of(P, A_0) & A_0 > 20)
2:child_of(A, P)
  Call 2 unifies rule 1
      output P = roger  A = tom
      %  First proof of child_of(A,P)  with P=roger
  No rule body
2:child_of(tom, roger) succeeded
3:child_of(C_0, roger)
  Call 3 unifies rule 1
```

```
      output C_0 = tom
  No rule body
3:child_of(tom, roger) succeeded
3:child_of(C_0, roger) seeking another proof
  no (more) proofs using rule 1 trying next rule for call 3
  Call 3 unifies rule 2
      output C_0 = june
  No rule body
3:child_of(june, roger) succeeded
3:child_of(C_0, roger) seeking another proof
  no (more) proofs using rule 2 trying next rule for call 3
3:child_of(C_0, roger) no (more) proofs
1:only_has_adult_children(roger) succeeded
P = roger : man

1:only_has_adult_children(P) seeking another proof
2:child_of(A, P) seeking another proof
  no (more) proofs using rule 1 trying next rule for call 2
  Call 3 unifies rule 2
      output P = roger  A = june
      % Second proof of child_of(A,P)  with P=roger  again
  No rule body
2:child_of(june, roger) succeeded
4:child_of(C_0, roger)
  Call 4 unifies rule 1
      output C_0 = tom
  No rule body
4:child_of(tom, roger) succeeded
4:child_of(C_0, roger) seeking another proof
  no (more) proofs using rule 1 trying next rule for call 4
  Call 4 unifies rule 2
      output C_0 = june
  No rule body
4:child_of(june, roger) succeeded
4:child_of(C_0, roger) seeking another proof
  no (more) proofs using rule 2 trying next rule for call 4
4:child_of(C_0, roger) no (more) proofs
1:only_has_adult_children(roger) succeeded
...
P = roger : man
```

.
.
.

To avoid the repeated answers for any parent that has more than 1 child we can change the definition of `only_has_adult_children` replacing the condition `child_of(_,P)` by the condition `has_a_child(P)` where

```
rel has_a_child(?human)
has_a_child(P) <=
    isa(P,human) &
    % When P needs to be found, generate in turn names of humans
    once(child_of(_,P))
% Then check, once only, if they have a child
```

We now get the answer `P=roger` just once.

```
| ?? only_has_adult_children2(P).

P = roger : man
...
P = bill : man
...
P = mary : woman
...
P = rose : woman
```

As a final version of the definition for the relation `only_has_adult_children` we replace the use of the `child_of` condition in the `forall` check by a call to the function `age`.

```
fun age(human) -> age_val
age(P) :: age_of(P,A) -> A
age(_) -> 0
% 0  is used as the default age if none is recorded

rel only_has_adult_children3(?human)
only_has_adult_children3(P) <=
    has_a_child(P) &
    % This will generate one at a time all humans with at least one child
    forall C (
            child_of(C,P) => age(P) > 20
            )
```

We can also `watch` a function evaluation.

```
watch age.

| ??  only_has_adult_children3(P).

1 : age(roger) (matches rule 1)
    1 : age(roger)  -> 110
    1 : age(roger) <- 110
2 : age(roger) (matches rule 1)
    2 : age(roger)  -> 110
    2 : age(roger) <- 110
P = roger : man
3 : age(bill) (matches rule 1)
    3 : age(bill)  -> 40
    3 : age(bill) <- 40
...
P = bill : man
4 : age(mary) (matches rule 1)
    4 : age(mary)  -> 40
    4 : age(mary) <- 40
...
P = mary : woman
5 : age(rose) (matches rule 1)
    5 : age(rose)  -> 40
    5 : age(rose) <- 40
...
P = rose : woman
```

9 Advanced Topics

In this section we look at two advanced topics: extending `QuLog` by accessing the Qu-Prolog and C levels; and building a runtime application.

9.1 Language Extensions

Currently `QuLog` includes only a core subset of Qu-Prolog functionality. It is our intention to include more functionality as the need arises but, in the meantime, it is easy to "lift" Qu-Prolog functionality to `QuLog` for a specific application as describle below. Given Qu-Prolog has a foreign function

interface to the C-level, this means a `QuLog` application can also access the C-level.

The first question the programmer needs to ask is "Am I lifting functionality as a relation, function or action?". The answer to this question determines the required approach. Remember that any functionality that is stateful needs to be lifted as an action. Because `QuLog` cannot check the various requirements imposted on definitions, for example modes and types, it is the programmers responsibility to make sure these requirements are met. The extra requirements on actions are given later. The examples given below can be found in `lift_eg.qlg` and `lift_eg_support.ql` in the `examples/introduction` folder.

The easiest of these to lift are relations. As an example consider listing `gmtime` which is a predicate for converting between seconds since the Unix epoch and the GM time structure. We consider the simplest case first - when the programmer wants to use the same name with the arguments in the same order and having the same types and modes. In this case, because we have a `time` data structure we declare this type as well as the relation type as below. This is all that is required to lift this predicate.

```
def time_t ::= time(nat, nat, nat, nat, nat, nat)
rel gmtime(!nat, ?time_t), gmtime(?nat, !time_t)
```

and then calling `gmtime` at the `QuLog` level (after type and mode checking) will simply call the Qu-Prolog predicate.

For various technical reasons we do not support lifting of functions: we have compiled in declarations that allow user defined functions to be used in higher order functions and we also compile in an extra rule that raises an exception in case the function would otherwise fail. Instead the approach that should be taken is to lift a relational version of the function and then define the required function in terms of this relation. So, for example, we want the function to delete an element from a list. Qu-Prolog already has the `delete` predicate defined and so we can lift that to the `QuLog` level by simply declaring the type:

```
rel delete(!T, !list(T), ?list(T))
```

and then defining the required function in terms of this relation:

```
fun deleteF(T, list(T)) -> list(T)
deleteF(Item, List) :: delete(Item, List, Result) -> Result
```

We can't use `delete` as the name of the function as it will get compiled into a `delete/3` predicate which conflicts with the builtin predicate.

In fact any relation that has exactly one argument in `?` mode and all the other arguments in `!` mode can be turned into a function using the above approach.

Lifting actions require more effort. The main problem is that actions are deterministic and are not allowed to fail. A given Qu-Prolog predicate that we might want to lift might be non-deterministic and typically might fail under certain situations. It might also throw exceptions. We therefore need to consider how these situations need to be handled and write Qu-Prolog code to take this into account.

It is often the case that we might want to use the same name as the Qu-Prolog predicate and so we need to map the name at the `QuLog` level to the support code at the Qu-Prolog level.

For an example we consider the TCP support Qu-Prolog provides and look at the process of lifting this support. For this example we will only consider `tcp_server`.

The first thing to notice is that many of these predicates have a socket as an argument and that is an integer at the Qu-Prolog level. To provide better type checking we fist declare a socket type:

```
def socket_t ::= socket(nat)
```

We can then make the declaration for the action (noting that we can use default arguments).

```
act tcp_server(Socket:?socket_t, Port:!nat default 0,
               Host:!atom default localhost)
"Create a Socket for a tcp_server for the given Port and Host"
```

Since the TCP predicates can raise exceptions we should trap them and turn them into `QuLog` exceptions by declaring the user exceptions:

```
def user_exception ::= tcp_exception(string)
```

The support code at the Qu-Prolog level is then

```
?-assert(qulog2qp_map(tcp_server(Socket, Port, Host),
                      tcp_server_interface(Socket, Port, Host))).

tcp_server_interface(Socket, Port, Host) :-
    catch(tcp_server(Socket, Port, Host), Pattern,
```

```
        handle_exception(Pattern)), !.

%% convert the QuProlog exception to the corresponding Qulog exception
%% where the argument is a string representation of the QuProlog exception
handle_exception(Pattern) :-
    open_string(write, Stream),
    write_term_list(Stream, [Pattern]),
    stream_to_string(Stream, Str),
    throw(tcp_exception(Str)).
```

The asserted fact `qulog2qp_map` is used by the `QuLog` compiler to replace the `QuLog` call by the Qu-Prolog level version of the call.

To access the C-level we use the Qu-Prolog foreign function interface to define predicates that call the C-level. Once this is done we simply follow the approach above to lift that Qu-Prolog interface code to the `QuLog` level.

The folder `examples/ev3` contains a simple example using a Mindstorm EV3 robot. The C-code is specific to this application but is realatively easy to modify for other applications.

The folder `examples/ROS` contains a foreign-function interface to ROS: ROS messages are assumed to be strings representing Prolog terms and are placed in the message buffer of the thread that started the ROS interface.

9.2 Building a Runtime Application

Typically we develop an application using the interpreter for experimenting and testing. Often we subsequently want to deploy the application as a runtime application - i.e. it can be started from the command line and does not require the interpreter.

The `bin` folder contains the Python program `qulogc` that takes a `.qlg` file and translates the file to the corresponding `.ql` file and then, if requested, compile the program to a runtime application.

The default call is

`qulogc` *application*

that will, assuming the program type checks, translate the program *application*`.qlg` to *application*`.ql` and then compiles that to a runtime application.

The default is that the program is a `Teleor` program and is to be used as a runtime application and so needs a definition of `qmain` - the same as the

Qu-Prolog `main` definition. If the application is not a `Teleor` application but instead is a `QuLog` application then a `-Q` switch can be used. If the translated code is intended to be used as library code for a Qu-Prolog application then a `-L` switch should be used and `qmain` should not be defined.

So, for example, if we wanted to translate the `QuLog` program `qapp.qlg` and compile it as a runtime application we would use

```
qulogc -Q qapp
```

On the other hand, if we wanted to translate the `QuLog` program `lib.qlg` as a library file (but not compile it) we would use

```
qulogc -Q -L  lib
```