QuLog: A flexibly typed logic and functional programming language with action rules - the *declarative* subset

Keith L. Clark and Peter J. Robinson

February 29, 2020

1 Introduction

QuLog is a declarative hybrid logic, functional, string processing language with an imperative action language sitting on top. Its declarative kernel was developed to complement our *guard* → *action* rule based robotic agent programming language *TeleoR*. The latter is a major extension of Nilsson's Teleo-Reactive robotic agent language. The primary use of declarative *QuLog* is to 'program' the agent's *BeliefStore* and to inference evaluate the *TeleoR* rule guards, which are restricted *QuLog* queries. Its action layer is used to extend *TeleoR* rule actions with atomic *BeliefStore* update actions and inter-agent communication actions routed via our Pedro communications server. The multi-threaded agent architecture that concurrently executes *TeleoR* programmed tasks is implemented in imperative *QuLog*. This draft paper, which currently lacks references, describes the declarative subset of *QuLog*. A companion draft paper describes the layer of action rules and illustrates their use for communicating agent applications.

QuLog relations are *typed* and *moded*, like Mercury. Its functions are just typed as all their arguments must be given as ground terms (terms with no unbound variables). 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 of relation or function calls, or due to arguments that *must* be given not having been computed before the call, particularly calls to *QuLog* primitives. They also enable us to guarantee at compile time that *TeleoR* rule actions will be fully ground and correctly typed when sent to a robotic resource. We need this to make *TeleoR* a serious 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. Functions always return ground term values.

To retain some of the flexility of use of Prolog we have *union* (aka *disjunctive*) types. These allow us to say that a relation accepts either integers or atoms in some input argument position, with run-time tests to allow different rules to be given depending on which type of value is given in a particular call, as in Prolog. However in Prolog such run-time tests are limited to a small range of sub-types of the generic *term* type, such as *integer* and *atomic*. In *QuLog* run-time type tests can be done for all primitive types, any program defined type, as well as higher order types.

In *QuLog* sub-type values are also considered as valid values of all their super-types. This means that a sub-type value may be given as an argument where a super-type is specified, and returned as a value in lieu of a super-type value. The type checker ensures that all function calls are given ground arguments of their declared argument types, and will return a ground value of the declared value type. It checks that each relation 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.

There is a sub-type lattice that includes every data value in *QuLog*, covered in Section 6.5. Relations and actions (defined by one or more action 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. The sub-type relationship for functions and relations is covered in Section 9. 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 also has support for *meta-level programming*. Compound terms representing relation calls can be constructed from and mapped into lists, as in Prolog. These call terms can be passed around as data terms and then evaluated as the relation calls that they denote. But unlike Prolog, this is done in a type and mode safe way. No meta-call is allowed in *QuLog* that does not satisfy the type and mode constraints of the relation being called.

This is discussed in Section 7.4.

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 and relation 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 introduction to declarative *QuLog* assumes familiarity with Prolog and higher order functional programming, as in languages such as Haskell or Scala. Sections 2, 3, 4 introduce its use as a typed relational programming, with its pattern match string processing support being covered in Section 4. Sections 5 and 8 deal with the functional subset and higher order programming. Sections 6, 9 and 10 deal with types, modes, the compile-time type/mode checking, and the need for auxiliary run-time type testing to filter out sub-type values. Section 7 introduces program defined types and illustrates their use. Type declarations for types with finite extensions double as both testers and *generators* for instances of the type. We exemplify *QuLog*'s debugging support in Section 3, and give a glimpse of its imperative action rules in Section 7.

2 *QuLog* Relational Subset

2.1 Python influenced 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 *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.

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. However if we want to use the atom name of a relation, function or actions as an atomic data value, it must be preceded by a single back-quote.

2.2 Moded type declarations

All relations (other than those with no argument), and all functions *should* have their types declared. If not declared, a default declaration is assumed where all arguments have *term* type, and for a function the value has *term* type. `term` is the top of the *QuLog* data type lattice. If an argument has type `term`, run-time type tests then have to be inserted in clauses and rules, before any *QuLog* primitive is used that requires a value of particular type, such as the arithmetic inequalities `<`, `>` etc. We exemplify this in Section 6.6.

The default mode for each argument of an undeclared relation is ground input, so an undeclared relation will be assumed to be a test only relation that needs to be called with ground arguments (terms not containing variables) but of unconstrained type. For Prolog relations, the default moded type of each argument relation would be specified using the *QuLog* moded type `??term`.

`term` says that the argument can be *any* Prolog term: a variable, `atomic` value - a `nat` (non-negative integer), an `int` (integer), a `num` (decimal number), an `atom` or a `string` (a double quoted sequence of any characters), or a `c(...)` compound term , which may contain variables. A `??` mode says a successful evaluation of the call *may not* result in a ground value for the argument term - may not generate any value, or may not fully instantiate a given compound term. Compound terms that may contain variables in argument terms we call template terms. A *QuLog* primitive `template: @term` will test for being a template term.

This contrasts with use of a `?` mode. This tells us that the argument, even if given as an unbound variable, or template term, will be fully instantiated to a ground term on success of the call. You should use the `?` mode whenever a ground value is guaranteed to be returned if not given as the argument value. Use of a variable in such an argument position, allows the variable to be later passed into a function call - for all function call arguments must be ground - or to be used as the value of an argument of a relation or action call that requires a ground value. If used as an argument of a robotic action of a *TeleoR* rule, it guarantees this action argument will be ground. If you want to emphasise that an argument for a call to a relation or action `must` be ground, you can use the prefix annotation `!`. If you mode an argument with a `?`, the *QuLog* type/mode checker will ensure that a ground value will be returned and gives an error message if this is not the case.

To summarise, a 3-argument relation `r` that does not have a type declaration will have an assumed, test only use, type declaration

```
rel r(!term,!term,!term)
```

The `rel` tells us that `r` is a relation. To indicate that a relation `r2` has arguments of type term where the first must be given as a ground term, the second can be a variable or any term, but will become ground on success of the call, and the third may likewise be a variable or any term and may not be ground on success of the call, you can use a declaration

```
rel r2(!term,?term,??term)
```

Finally, to indicate that a relation `r3`, had to be given a number as first argument, could be given but may also return an atomic value as second argument, and may be given a variable or list term containing only strings and variables and which may not become ground on success, use the declaration

```
rel r3(!num,?atomic,??list(string))
```

An unannotated type is implicitly `!` annotated - ground argument only. We could therefore drop the prefix `!` from the `num` type. Surrounding a type expression with `list` indicates a list of terms of that type.

A relation can have more than one type declaration. For example, suppose we know that when `r3` is called with an integer as first argument, and is given an integer second argument, it will always ground its last argument, we can use the disjunctive type declaration

```
rel r3(num,?atomic,??list(string)), r3(int,int,?list(string))
```

Here we have dropped the prefix `!` mode on `num` and `int` as this is the default for an un-moded argument.

2.3 Relation definitions

```
rel child_of(?atom,?atom)                    % Type and mode declaration
child_of(mary, peter)
child_of(bill,peter)
child_of(peter,john)
.
.
rel age_is(?atom,?atom)
age_is(peter,40)
age_is(mary,12)
.
.
rel male(?atom)
male(peter)
.
.
rel female(?atom)
female(mary)
.
.
rel descendant_is(!atom,?atom)    % First arg. of  a call must be a given
descendant_is(A,D) <= child_of(D,A)
descendant_is(A,D) <= child_of(C,A) & descendant_is(C,D)

rel ancestor_of(?atom,?atom)    % Test or generate use of both arguments
ancestor_of(A,D) <= child_of(D,A)
ancestor_of(A,D) <= child_of(D,P) & ancestor_of(A,P)
```

The *input* (`!`) mode annotation for the first argument of the type/mode declaration for `descendant_is` is because when the name of the ancestor `A` is given it is more efficient than the `ancestor_of` definition. It is worse than the other definition if `D` is given and `A` is to be found. The `descendant_is` recursive rule 'walks' down the `child_of` links starting at the given `A`. In contrast, the `ancestor_of` recursive rule 'walks' up the `child_of` links from a given descendant `D`. If `D` is not given then it finds some child/parent pair `D,P` and then walks up from that `P`. This is acceptable if `A` is not given as well, and all `ancestor_of` pairs need to be found, but if `A` is given the logically equivalent `descendant_is` relation should be queried.

If the programmer forgets and calls the less efficient `descendant_is` relation in a position where the first argument will not be given, the mode checker will give a mode use error so that the programmer can change the called relation. However, it will not object if `ancestor_of` is used where `A` will be given with `D` to be found, where `decendant_is` is more efficient. We are considering adding an optional `equiv` declaration to *QuLog*. Then, in the presence of

```
equiv ancestor_of, descendant_is
```

the mode checker would replace a call to `descendant_of`, where the first augment is not given, by a mode correct call to the equivalent `ancestor_of`, rather than raise a mode error. Conversely, it would replace a call to `ancestor_of` by a call to the more mode constrained equivalent relation `descendant_of`, when the mode analysis has determined that the first argument will be known at the time of the call. It would issue an advisory when either kind of replacement is made.

A simple list processing example is

```
rel doubleL(!list(num),?list(num))        % [num] is type of a list of numbers
doubleL([],[])
doubleL([N,..Nums], [DN,..DNums]) <=
     DN = 2*N & doubleL(Nums,DNums)
```

The `list(num)` type expression for both arguments indicates this is a relation over lists of floating point numbers. The `!` mode annotation on the first argument indicates that in any call this argument must be a ground (variable free) list of numbers or integers. It can contain integers, as `int` is a sub-type of `num`. The annotation on the second argument indicates this may be an unbound variable in the call, for generate use, or that it can be a ground list of numbers for test use, or even a list of numbers and variables for mixed test/generate use of the numbers on the list. For example, a call

The `,..` should be read as *followed by*. `[N,..Nums]` is read as `N` *followed by* list `Nums`, which may be the empty list `[]`. After the `,..` we usually have a variable which denotes the tail list of terms following the number of elements that precede the `,...` A list pattern that does not have a variable after the `,..` can be rewritten as one that does. `[X1,X2,..[X3,..L]]` can be written `[X1,X2,X3,..L]`. `[X1,X2,X3,..]` is equivalent to `[X1,X2,X3,.._]`. We can use the Prolog infix `|` instead of `,..` except that `[X1,X2,X3|]` cannot be used as shorthand for `[X1,X2,X3|_]`

It will be a match against a number value if `N` is a number. If it were not, or is an unbound variable, we would have a run-time error when the multiplication `2*N` is attempted. This is why we have type and mode of use declarations in *QuLog*, checked against the source program, so such a run-time error will not occur. All uses of `double` will be checked by the mode analyser to ensure that for every call the first argument will always be an ground list of numbers, or integers.

2.4 General form of relation rules

The 'if' operator for a *QuLog* conditional rule is `<=`. The 'and' operator is `&`. We choose to use `<=` as it is often used as the 'if' of predicate logic, and we believe that comma is overloaded in Prolog. Its use as the associative 'and' for conjunction also means that tuples are not fully supported as a data term in Prolog, as explained in Section 6.1.

Typically, a conditional rule for a relation is written

Head `<=` $Cond_1$ `&` $Cond_2$ `&` `...` `&` $Cond_n$

where `Head` is a predication of the form *rel(*Arg_1*,...,*Arg_k*)*, $k \geq 0$, and each $Cond_i$ is a predication, or a negated predication using the negation-as-failure operator `not`.

In *QuLog* `not` is restricted to a *single* positive condition, as in Datalog. In a negated condition all but the variables starting with an `_` *must* or are existentially quantified have ground values by the time it is evaluated. The relation modes are used to check this will be the case at compile time. If not, a mode error is signalled. If the programmer wants a negated test that requires a conjunction, an auxiliary relation must be defined with body that conjunction. The negation is then applied to a call to the auxiliary relation.

2.5 General form of moded type declarations

QuLog's relation that have arguments must all be mode and type declared.

A type+mode declaration for a k argument declarative relation [`r` has the form

`rel r(`mt_1`,...,`mt_k`)`

Here each mt_i is a type expression t_i (often just a type name) with an optional `!`, `?`, `??` or `@` mode. The absence of a mode annotation is equivalent to having given a prefix `!` annotation. Clauses defining a no argument relation `r` are be declared as `rel r()`.

2.6 Non-grounding mode

A `??` mode is a relaxation of the `?` mode, most useful when the argument type covers non-atomic terms. As for a `?` moded argument, an unbound variable, or a non-ground term, may be passed as an argument value for a `??` moded argument, but the call may not ground that argument. It can return the same template term argument, or another template that is a partial instantiation of the given template argument, or a ground instantiation. If an unbound variable is given as the argument in the call, it may still be an unbound variable (perhaps a different one) after the call.

If for some particular call `C`, of a `rel`, with a `??` moded argument `A`, it is important that the call returns a ground value because `A` will be passed as a `!` moded argument to a later call, then the primitive `ground(A)` test should be used after `C` to force backtracking until `C` returns a ground value for `A`. Without such a `ground(A)` test, the mode checker will raise an error. If it is just required that `A` be bound to a term of the form `con(...)`, then the primitive `template(A)` can be used after the call `C`, to force backtracking to generate such a partial structure.

The `??` mode is not often used for user programs. We have an example in Section 4.2. The unification primitive has moded type `??term = ??term`. `=` takes any pair of terms *trm1* and *trm2*, and succeeds if it can instantiate each to single term *trm* by binding variables, *providing* no variable is bound to a term that contains that variable (the *occurs* check of unification). The *trm* result of unification may not be ground, it will be *trm1* or *trm2* if the other is a variable. *trm* is always a least instantiated common instance of *trm1* and *trm2*.

As an example, `f(X,g(Y,a))=f(h(U),g(V,U))` instantiates both to the non-ground `f(h(a),g(V,a))`, or the non-ground `f(h(a),g(Y,a))`.

The ability to return and pass to a later call a template term, that the later call either grounds or just further instantiates leaving as a non-ground term, is a powerful programming technique unique to logic programming

which is preserved in *QuLog*. It comes at the cost of having to support unification rather than just pattern matching, and having to treat variables as first class values. However, because of the mode information in *QuLog* we can, for the most part, safely avoid the costly occurs check. We discuss such implementation issues further in the companion *QuLog* paper on its action rules. The retention of unification, and the consequent need to have variables as first class values, is a major difference between *QuLog* and Mercury.

2.7 Iteration over all solutions to a query

Rule bodies can contain *forall* conditions which have *explicitly* quantified variables. Their simple form is

`forall` $\mathtt{V}_1,\ldots,\mathtt{V}_j$ `(exists` $\mathtt{EVars}_1$ `(`*Conditions*$_1$`) =>`
`exists` $\mathtt{EVars}_2$ `(`*Conditions*$_2$`))`

with $\mathtt{j} \geq 1$. The (...) brackets around the implication are essential. The variables $\mathtt{V}_1,\ldots,\mathtt{V}_j$ are said to be *universally* quantified.

Each variable of $\mathtt{V}_1,\ldots,\mathtt{V}_j$ *must* appear in *Conditions*$_1$ *and Conditions*$_2$. They must be variables *not* used anywhere else in the query or rule body in which the `forall` appears, and all other variables of the *forall*, except the explicitly existentially quantified variables $\mathtt{EVars}_1$ and $\mathtt{EVars}_1$, and _ variables, must have ground values when it is evaluated. If there are no existentially quantified variable for either the antecedent or consequent, the `exists` is dropped.

Each existentially quantified variable must have a name different from that of any other variable in the query or clause body in which the `forall` appears. Nested existentially quantifications are *not* allowed. They are not necessary as all the inner existential quantified variables can be put into a single outermost existential quantification.

`(exists X (g(X) & exists Y h(X,Y,Y)) =>` ...`)`

is equivalent to

`(exists X,Y (g(X) & h(X,Y,Y)) =>` ...`)`

The *forall* holds if for all bindings for its *UVars* quantified variables, given by a solution θ to *Conditions*$_1$, in which the explicit existential quantifica-

tions are ignored, then $Conditions_2\theta$ has at least one inferable answer for its existentially quantified variables[1].

2.7.1 Example uses of `forall`

```
rel only_has_adult_sons(?atom)      % arg. can be a var or given as an atom
only_has_adult_sons(P) <=
    person(P,_,_) &    % This checks P is a male or female, or finds one
    forall C (child_of(C,P) =>
                  exists A (male(C) & age_is(C,A) & A>20))

rel person(?atom,?atom,?nat)
person(P,male,A) <= male(P) & age_is(P,A)
person(P,female,A) <= female(P) & age_is(P,A)
```

We need two rules for `person` as *QuLog* has no disjunction.

2.8 Commitment to a rule

The infamous Prolog `!` *cut* primitive is *not* allowed in *QuLog* rules or queries, but commitment to a rule can be expressed using a `::` commitment test *before* the `<=`, as in:

- *Head* ::*Commit* `<=` *Body* - do not use any later rule if *Commit* holds even if *Body* fails.
- *Head*::*Commit* - ditto, rule has no preconditions except *Commit*.
- *Head*::true - do not use any later rule if the *Call* matches *Head*

Commit is a conjunction of predications and negated predications. It often only has one solution, and as such it is a query test extension of the head unification with a call. If it happens to have more than one solution only the first solution will be found. It may generate values for variables in non-input arguments of the *Head*, which may be used in *Body*

The recommended style in defining a relation *Rel* is to have mutually exclusive *Commit* conditions or rule *Head*s with perhaps a last default rule

[1] If there is an explicitly existentially quantified consequent `exists` $EVars_2$ $Conditions_2$ this will only have the variables in $EVars_2$ unbound by the time $Conditions_2$ is checked, that is the existential quantification is being used as a test. The compiler always transforms the consequent to `once` $Conditions_2$ (see Section 2.10)

that implicitly has an initial negated call `not` *above_Rel(...)* to an auxiliary (but undefined) relation *above_Rel*. The *above_Rel* definition has a rule $above\_Rell_R$ constructed from the head term arguments and *Commit* tests of each rule R above the last rule, such that the use of the $above\_Rell_R$ rule would succeed exactly when the head unification and *Commit* test of `R` would succeed. The implicit negated condition `not` *above_Rel(...)* tests that none of the head term match/commit tests of the rules above can be inferred. We give an example below.

`::` can be read as *such that* and the *Commit* test following it is a 'case' condition for use of the rule augmenting the term matching tests given by any non-variable terms in the *input* and *in/out* argument positions in the rule head. Ideally the 'case' conditions are non-overlapping, but in practice they often are not.

Disjunction and `if ... then ... else ...` conditionals are *not* supported. Their purpose must be achieved by defining auxiliary relations with commit tests.

2.9 Example use of commit

```
rel max_of(num,num,?num)                    % First 2 args implicitly ! moded
max_of(M,N,Max)::M=<N <= Max=N
max_of(M,N,Max)::M>N <= Max=M
```

The rules treat disjoint cases. As in Prolog the commit test of the second rule can be dropped and we would get the same behaviour. Implicitly the second rule then has the test `not above_max_of(M,N,Max)` where

```
above_max_of(M,N,Max) <= M =< N
```

i.e. in this case the implicit negated test is the equivalent of the given test `M>N`.

Note that the modes mean that the `max_of` relation will only be called if the first two arguments are given, thus preventing a runtime error on use of the arithmetic comparison tests. An attempt to use it when they will not both be number values - checked by the type and mode analyser - results in a compile time error.

2.10 Restricting a query to a single solution

QuLog has a `once` operator that can be used as a prefix to a single call or a bracketed conjunction of predications or negated predications (so no nested `once` calls or `foralls`), as in

`once (`*Conditions*`)`

When it is used only the first successful inference of *Conditions* is found. After is has been found, there is no backtracking inside the *Conditions* conjunction to find alternative inferences/solutions. Brackets are not needed if there is only one condition.

It is particularly useful if *Conditions* are being used to test and not to generate values. For example, the `once` in the body of a rule of the test only relation `has_son`.

```
rel has_son(?atom)
has_son(P) <= person(P,_,_) & once (child_of(C,P) & male(C))
```

If a call to the relation succeeds backtracking does not look for alternative proofs that the found `P` has a male child.

2.11 Generating lists and sets from query conjunctions

QuLog has a list comprehension expressions that generate lists of terms, each one being the instantiation of a template term corresponding to a solution to a conjunction of conditions. It also has a set comprehension expression. Sets contain no duplicate terms. Sets and lists are distinct types in *QuLog*.

To generate a list of instances of some `Term` corresponding to solutions of some conjunction of conditions we can use an expression

`[`*Term* `:: exists` *Vars Conditions*`]`

Vars are all the variables in *Conditions* that do not appear in `Term`, or elsewhere in the clause or query in which the list expression appears. They are the local variables of *Conditions*. If there are no such variables, `exists` *Vars* is dropped.

All variables in *Conditions*, not in *Vars* or *Term*, must appear earlier in the clause or query and be such that they will have a ground binding by the time the expression is evaluated. This also applies to variables in *Term* not appearing in *Conditions*.

Conditions is a conjunction of predications and negated predications with the usual contraints on the negated predications. The predications may have list comprehension expressions and function call arguments. Finally, any variable in `Term` that will not have a ground value when the set expression is evaluated *must* appear in a grounding condition in *Conditions*. This ensures the generated list will only contain ground values.

The list comprehension expression denotes the list of all instances of *Term* such that *Conditions*, *as found*. That is, the instances of *Term* are generated by finding all the different inferences of *Conditions*, ordered as found by the backtracking search for all the different inferences. It can contain duplicates. It is like the Prolog `findall`.

2.11.1 Set comprehension

The set comprehension expression is very similar in structure but uses {..} braces instead of `[..]` brackets. The generated set will not contain any duplicates. However its internal representation is such that operations of `union`, `diiff` and `inter` (set uniion, difference and intersection) are efficiently implemented. Also a set will be displayed surrounded with {..} braces, with its elements ordered using the standard *QuLog* term ordering relation `@<`, which is the same as Prolog's.

A set expression has the form

{*Term* `::` `exists` *Vars Conditions*}

and is subject to the same constraints regarding variable bindings as list comprehension expressions.

2.11.2 Conversion between lists and sets

To covert a list to a set we use the built in function `toset`. `toset([3,2,-6,3,-4])` is the set {`-6,-4,2,3`}. To convert a set to a list, that can be processed using list patterns, we use the built in function `tolist`. So, `tolist({2,9,2,-2,-4})` is the list `[-4,-2,2,9]`.

We can use `in` to access elements of both lists and sets. Used to access elements of a set the elements will be returned starting at the least element in size order. So, `E in {2,-3,7,2,0}` will successively bind `E` to -3, 0, 2 and 7. So we can think of a set as being ordered.

The set comprehension expression can be viewed as shorthand for

`toset([`*Term* `::` `exists` *Vars Conditions*`])`

2.12 Example uses of comprehension expressions

```
rel children_are(?atom,?set(atom))        % {atom} is type for a set of atoms
children_are(P,L) <=  person(P,_,_) & L = {C :: child_of(C,P)}
                                          % L will be a set of children of P
```

The queries

```
| ?? Children = toset([C1 :: child_of(C1,peter)]
                    <> [C2 :: child_of(C2,mary)]).
| ?? Children = {C1 :: child_of(C1,peter)}
                    union {C2 :: child_of(C2,mary)}.
```

will both instantiate `Children` to the set of the children of either `peter` or `mary`. The set will be displayed with its elements ordered using the term order relation `@<`. Also, when the elements of a set `S` are accessed using `E in S` then `E` is bound to successive elements in `@<` term order.

For this example we could also have used the query

```
| ?? Children = {C :: child_of_either(C,peter,mary)}.
```

where `child_of_either` has been defined as

```
rel child_of_either(?atom,?atom,?atom)
child_of_either(C,P,_) <= child_of(C,P)
child_of_either(C,_,P) <= child_of(C,P)
```

`<>` is the list concatenation function of *QuLog*. It can be used for 'glueing' ground lists together, no matter how constructed. It can be used to concatenate lists of different types `list(T1), list(T2)`. The result is a list of the union type of `list(T1), list(T2)`, i.e. it is list of type `list(T1 || T2)`. `||` is the type union operator. We will say more about union types in Sections 6.4, 6.9.

`<>` can also be used for splitting a given ground list subject to constraints. This is when used in the right hand side pattern of the `=?` operator, as illustrated in Section 4.2.

There is also an `append` relation primitive in *QuLog*. Its use covers both the appending and splitting use of `<>`, and more. It *must* be used, not `<>`, to concatenate or split lists containing variables or template terms. We shall illustrate its use in Section 4.2.

Here is an example that uses a necessary existential quantification.

```
rel adult_children_are(?atom,?list(atom)
adult_children_are(P,L) <=
    person(P,_,_) &
    L = {C :: exists A (child_of(C,P) & age_is(C,A) & A>17)}
              % L will be a set of name ordered adult children of P
```

```
rel children_in_age_order(?atom,?set((int,atom)))
children_in_age_order(P,L) <=
   person(P,_,_) &
   L = {(A,C) :: child_of(C,P) & age_is(C,A)}
               % L will be a set of pairs ordered by increasing age


rel children_as_recorded(?atom, ?list(atom))
children_as_recorded(P,L) <=
   person(P,_,_) & L = [C :: child_of(C,P)]
```

Note the change from {..} brackets to `[..]` brackets. `L` will be a list of names of children of P in the order that they are found by the `findall` Prolog primitive, i.e. in the fact order.

2.12.1 Ordering lists

The simplest way to order a list is to use the converter functions `toset`, `tolist` in succession. The function

```
order: list(T) -> list(T)
order(L) -> tolist(toset(L))
```

will convert `L` in to list ordered by increasing value with all duplicates removed.

2.13 Relational queries

You have probably inferred by now that `| ??` is the *QuLog* interpreter prompt to enter a query or command. In Section 3 we shall illustrate the entering of commands such as those needed to *consult* a program file, to *show* particular definitions, and to *watch* calls to a particular relation. Here we say something about *QuLog*'s relation queries. These are quite different in *QuLog* from Prolog.

```
| ?? person(P,male,_) & child_of(C,P)
             % <fullstop><return> is query end

P=bill:atom                          % type of each answer binding is given
C=mary:atom
...                  % the ... indicates backtracking to find next solution
.                    % more answers
```

```
.
...
P=cheryl:atom                                    % The 5th answer
C=jake:atom
```

If the response is to enter `..<return>` after the 5 answers have been displayed, up to 5 more will be displayed. If `..3` is entered, up to 3 more answers are displayed and you must respond again to get more. In general, if the response is `..n`, where `n` is an integer, `n` more answers are displayed if there are `n` more.

For the above query we could use _ (underscore) for the age argument of `person` to suppress seeing its bindings. However, if we had a minimum age condition we could not do this. The query

```
| ?? person(P,male,A) & A>40 & child_of(C,P)
```

will display the `A` answer bindings as well as those for `P` and `C`. To see just the `P`, `C` bindings, we can prefix the query with these variables.

```
| ?? P,C :: person(P,male,A) & A>40 & child_of(C,P)
```

This will give us up to 5 pairs of bindings for `P` and `C` in the first instance. Notice that this is very like a list comprehension. If we did put a pair of the list comprehension brackets around the query, and wrapped `P`, `C` into a single term such as `(P,C)`, we would get *all* the answers displayed as a single list.

```
| ?? X = [(P,C):: exists A person(P,male,A) & A>40 & child_of(C,P)].
```

will display a list of pairs `[(bill,mary),...,(cheryl,jake),...]` as the instantiation of `X`. We have had to explicitly existentially quantify `A` which is implicitly existentially quantified in the relational query.

If we only wanted to see 3 answers of a query in the first instance we can prefix the query with that number. We can also use an existential quantification for the variables for which we do not want to see answer bindings instead of enumerating the ones for which we want to see answer bindings.

```
| ?? 3 ::  exists A (person(P,male,A) & A>40 & child_of(C,P)).
```

After the first 3 answers have been displayed we must enter a `..` to see the next 3, or a `..  n`, with `n` an integer, to see the next `n`. We can also prefix

a query with an integer greater than the default up to 5 answers displayed. We can also change this default using a command, as given in Section 3.

We can just give the number of solutions bindings we want to see in the first instance without giving a sequence of variables. The query

```
| ?? 1 :: person(P,male,A) & A>40 & child_of(C,P).
```

behaves exactly like a Prolog query. You will need to enter `..` to see each successive answer.

With all the query forms, if you enter `<return>` when able to enter `..` to see more answers (if any), the query evaluation will terminate. You will again get the top level prompt `| ??` for a new expression, relation query or command to be entered. Expression queries and commands are discussed further in Sections 5.2 and 3, respectively.

```
| ?? child_of(_,P). % <fullstop><return> used to mark query end

C=peter:atom
...
C=peter:atom
...
C=john:atom
...
.            % More answers
.
```

Notice that we get the answer `C=peter:atom` twice as there are two ways of inferring that `peter` is a child of someone using the facts of Section 2.3. *QuLog* also attaches its minimal type of each answer binding.

3 Query line commands and debugging

```
| ??  consult file.        % Double return used to mark end of command

success
```

will read in and type and mode check the *QuLog* program in *file* giving useful syntax, mode and type error messages, and perhaps suggestions for re-programming.

```
| ?? show  BeginName.    % For example  show chi.
```

will complete the *BeginName* to names of defined functions and relations that start with *BeginName* and display them with their type declarations. `show` on its own will display them all. Unlike most Prologs they will have their programmer given variable names of the source file.

`| ??  types.`

will display all the type definitions and type declarations. You can also display a specific type definition using `type` *TypeNm*.

`| ??  set_num_answers` *n*`.`

will change the default number of answers displayed before a *continue* input `..` is required from 5 to *n*. *n* must be a positive integer.

`| ??  watch` *relation*`.`

will display every call to *relation*, say which rule it unifies with, and give the bindings for the variables of the call and the rule head. The rule head variables with be of the form *var_i* where *var* is the variable name used in the relation rule and `i` is an integer. If the call is retried this is also displayed along with the same information for the next rule with which the call unifies, if any.

A variant `watchR` *relation* will also display the partially instantiated body of the rule being tried. `unwatch` *relation* will turn the watching of *relation* off. Any number of relations can be watched at the same time. Watching a relation in *QuLog* is a little like the spying feature of Prolog except Prolog's spy invokes the interactive tracer at the first call of a spied relation. The *QuLog* watch is not interactive. It is a substitute for inserting write statements in Prolog rules that most Prolog programmers actually use for debugging. This is what `watch` and `watchR` do under the covers. It cannot be done at the source level as writing terms is an action and can only be done inside an action rule of *QuLog*.

We give an example use of `watch` and `watchR`. For this we assume we have a program defined relation

```
rel app(??list(T),??list(T),??list(T)),
    app(!list(T),!list(T),?list(T)),
    app(?list(T),?list(T),!list(T))
app([],L,L)
```

```
app([U|L1],L2,[U|L3]) <= app(L1,L2,L3)
```

that has the same rules as the builtin `append`, which cannot be watched. This is a polymorphic relation that can be used to append or split lists of any type `T`. The most general type declaration is the first. It tells us the relation will handle list terms containing variables and that it may not instantiate all these variables. The second two give extra information to the mode checker. The first says that if the first two arguments are ground in a call, the third will be ground after a successful evaluation. The last says that the first two arguments will be ground after a successful call, if the third argument is given as a ground list.

```
| ?? watch app.
success                                  % app is now being watched

| ?? app([1,2,3],[4],L).


1:app([1, 2, 3], [4], L)
  Call 1 unifies rule 2
      output L = [1 |L3_0]
  Rule body is:
      app([2, 3], [4], L3_0)
2:app([2, 3], [4], L3_0)
  Call 2 unifies rule 2
      output L3_0 = [2 |L3_1]
  Rule body is:
      app([3], [4], L3_1)
3:app([3], [4], L3_1)
  Call 3 unifies rule 2
      output L3_1 = [3 |L3_2]
  Rule body is:
      app([], [4], L3_2)
4:app([], [4], L3_2)
  Call 4 unifies rule 1
      output L3_2 = [4]
  No rule body
4:app([], [4], [4]) succeeded
3:app([3], [4], [3, 4]) succeeded
2:app([2, 3], [4], [2, 3, 4]) succeeded
1:app([1, 2, 3], [4], [1, 2, 3, 4]) succeeded
```

```
L = [1, 2, 3, 4] : list(digit)

1:app([1, 2, 3], [4], L) seeking another proof
2:app([2, 3], [4], L3_0) seeking another proof
3:app([3], [4], L3_1) seeking another proof
4:app([], [4], L3_2) seeking another proof
  no (more) proofs using rule 1 trying next rule for call 4
4:app([], [4], L3_2) no (more) proofs
  no (more) proofs using rule 2 trying next rule for call 3
3:app([3], [4], L3_1) no (more) proofs
  no (more) proofs using rule 2 trying next rule for call 2
2:app([2, 3], [4], L3_0) no (more) proofs
  no (more) proofs using rule 2 trying next rule for call 1
1:app([1, 2, 3], [4], L) no (more) proofs

| ?? watchR app.
success                    % Next level of watching turned on for app

| ?? app([U,2,V],[9],[1,W,3,..L]).

1:app([U, 2, V], [9], [1, W, 3,..L])
  Call 1 unifies rule 2
      input  U_0 = 1  L1_0 = [2, V]  L2_0 = [9]  L3_0 = [W, 3 |L]
      output U = 1
  Rule body is:
      app([2, V], [9], [W, 3,..L])
2:app([2, V], [9], [W, 3,..L])
  Call 2 unifies rule 2
      input  U_1 = 2  L1_1 = [V]  L2_1 = [9]  L3_1 = [3 |L]
      output W = 2
  Rule body is:
      app([V], [9], [3,..L])
3:app([V], [9], [3,..L])
  Call 3 unifies rule 2
      input  U_2 = 3  L1_2 = []  L2_2 = [9]  L3_2 = L
      output V = 3
  Rule body is:
      app([], [9], L)
4:app([], [9], L)
  Call 4 unifies rule 1
```

```
      input  L2_3 = [9]
      output L = [9]
  No rule body
4:app([], [9], [9]) succeeded
3:app([3], [9], [3, 9]) succeeded
2:app([2, 3], [9], [2, 3, 9]) succeeded
1:app([1, 2, 3], [9], [1, 2, 3, 9]) succeeded
U = 1 : digit
V = 3 : digit
W = 2 : digit
L = [9] : list(digit)

1:app([U, 2, V], [9], [1, W, 3,..L]) seeking another proof
2:app([2, V], [9], [W, 3,..L]) seeking another proof
3:app([V], [9], [3,..L]) seeking another proof
4:app([], [9], L) seeking another proof
  no (more) proofs using rule 1 trying next rule for call 4
4:app([], [9], L) no (more) proofs
  no (more) proofs using rule 2 trying next rule for call 3
3:app([V], [9], [3,..L]) no (more) proofs
  no (more) proofs using rule 2 trying next rule for call 2
2:app([2, V], [9], [W, 3,..L]) no (more) proofs
  no (more) proofs using rule 2 trying next rule for call 1
1:app([U, 2, V], [9], [1, W, 3,..L]) no (more) proofs


| ?? unwatch app.
success                                  % Watching of app is turned off
```

A watched relation does not need to be called at the query level. It could be called deep inside some query evaluation.

3.1 Fact updating commands

QuLog action rules can update facts for relations defined solely by a sequence of facts for relations declared to be `dyn` (dynamic) relations. This is similar to the Prolog `dynamic` except that **dyn** relations can only be defined by a sequence of ground facts.

Suppose we had type declared the `child_of`, `male`, `female` and `age` relations using

```
dyn child_of(atom,atom), female(atom), male(atom)
```

The `dyn` keyword tells the *QuLog* system that these three relations can be updated after the program file has been consulted. They can be updated both in action rules and command queries to the *QuLog* interpreter. Since dynaimc relations always comprise a sequence of fact, the mode for each argument is `?` so does not have to be given in the type declaration.

The basic dynamic updaters are

`forget` *Facts* `remember` *Facts*

`forget` *Facts*

`remember` *Facts*

All these are executed atomically - i.e. no other thread will get a timeslice until the dynamic updates have been done.

We can enter query commands such as

```
| ??  remember child_of(olive,sammy); remember female(sammy);
      remember female(olive)
```

to add three new dynamic facts. We have used `;` to separate the actions of the command as this is what must be used in *QuLog* action rules. A better approach, which will do all of these updates atomically, is simply to use the query

```
| ??  remember child_of(olive,sammy), female(sammy), female(olive)
```

Suppose we also had facts for the relation `dyn spouse(atom,atom)`. We could do an update recording a divorce and re-marriage using

```
| ?? forget married(bill,_) remember married(bill,mary)
```

4 Non-deterministic String and List Pattern Matching

Strings in *QuLog* are the same as C strings - packed sequences of bytes. They are not lists of byte codes as in most Prologs.

`=?` can also be used as a string or a list pattern match operator to split strings into substrings, or lists into and sublists, satisfying certain conditions. Its uses are

StringExpression =? *StringPattern*

ListExpression =? *ListPattern*

4.1 String matching

For string matching we use ++ for string concatenation and splitting. We use `<>` for list appending and splitting. When one or more `++` or `<>` operators appear at the top level of the right hand side of `=?` it acts as a *non-deterministic* or multi-valued match operator. There may be several bindings of variables appearing in the left hand side pattern that make both the pattern and the expression have identical string or list values. As a simple example,

```
"hello" =? S1 ++ S2
```

has seven different possible solutions for `S1` and `S2` starting with `S1=""`, `S2="hello"`, then `S1="h"`, `S2="ello"`, and ending with `S1="hello"`, `S2=""`.

In the pattern we can have one of

```
Var :: Test(..,Var,..)
Var / REString :: Test(..,Var,..)
```

(where the test part is optional) that restrict the values assigned to `Var` to those that satisfy `Test` or that match the regular expression. For the first, failure of the test will cause backtracking to find an alternative value for `Var`. When there are no more alternatives to the first or the regular expression doesn't match, it causes an attempt to find alternative values for variables appearing to the left of this condition in the pattern, ultimately resulting in a failure of the entire `=@` non-deterministic pattern match. For list matching the variable `Var` can be replaced by a list pattern such as `[E]` or `[E|L]` where the associated `Test` can contain all the variables of the pattern. As an example, the pattern match

```
"hello" =? S1::S1\="" ++ S2::S2\=""
```

excludes the first and last solutions we gave above.

Here are some uses of string patterns to 'parse' strings representing sentences into lists of string words. Such string processing can be a precursor to Definite Clause Grammar (DCG) parsing of lists of string words.

```
rel sepchar(?string)
"Word separator"
sepchar(" ")
sepchar(",")
sepchar(";")

rel endchar(C : ?string)
"C is a sentence terminator"
endchar(".")
endchar("?")
endchar("!")

rel symbolchar(?string)
symbolchar(C) <= sepchar(C)
symbolchar(C) <= endchar(C)

rel spaces(!string), wordchar(!string)

spaces(S) <=
    #S>0 & forall Ch (Ch in S => Ch=" ")

wordchar(S) <=  #S = 1 & not symbolchar(S)

rel word(!string), seps(!string)

word(S) <=
    #S>0 & forall Ch (Ch in S => wordchar(Ch))

seps(Seps) <=
    #Seps>0 & forall Ch (Ch in Seps => sepchar(Ch))

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)

| ?? words("Hello    Keith,   how are you?", Ws).

Ws = ["Hello","Keith","how","are","you"]:list(string)
```

4.2 List pattern matching

```
| ?? [1,2,3] =? L1<>L2.

L1 = [] : list(Ty1)
L2 = [1, 2, 3] : list(nat)
...
L1 = [1] : list(nat)
L2 = [2, 3] : list(nat)
...
L1 = [1, 2] : list(nat)
L2 = [3] : list(nat)
...
L1 = [1, 2, 3] : list(nat)
L2 = [] : list(Ty1)

% or we can use
| ?? [1,2,3] =? [U1,..L1]<>[U2,..L2].

U1 = 1 : nat
L1 = [] : list(Ty1)
U2 = 2 : nat
L2 = [3] : list(nat)
...
U1 = 1 : nat
L1 = [2] : list(nat)
U2 = 3 : nat
L2 = [] : list(Ty1)

| ?? [1,2,3,4] =? [1,U]<>[3|L].

U = 2 : nat
```

```
L = [4] : list(nat)

rel splitsAroundElement(!list(T), !list(T), !T,
                        ?list(T), ?list(T))
splitsAroundElement(L1,L2,E,OL1,OL2) <=
    L1<>L2 =?  OL1 <> [E] <> OL2


| ?? splitsAroundElement([1,5,3],[8,5,7],5,OL1,OL2).

OL1 = [1] : list(nat)
OL2 = [3, 8, 5, 7] : list(nat)
...
OL1 = [1, 5, 3, 8] : list(nat)
OL2 = [7] : list(nat)
```

The left hand side of a list match can use set expressions to denote the component lists, e.g. `[E|cond(E)]<>L` =? *list_pattern*.

We could also code the above examples using the built in `append` relation to split lists as in Prolog. We think the above are more transparent. Here is the rule for `splitsAroundE` using the *QuLog* builtin concatenation relation `append`.

```
splitsAroundE(L1,L2,E,OL1,OL2) <=
   append(L1,L2,L12)  &
   append(OL1,[E|OL2],L12)
```

In this case it is not too difficult to decipher but it is less clear than the use of the split pattern using `<>`. You have to look at the use of the variables quite closely to determine what it is doing. It does not wear its 'semantics' on its sleeve unless you are well practiced at the use of `append` for list splitting.

`append`'s library definition is the same as the earlier `app`.

Here is a query we cannot express using `<>` as it uses a template term for the appending result.

```
append([1,U],[3],[V,2|L]).

U=2:nat
V=1:nat
L=[3]:[nat]
```

5 *QuLog* Functional Subset

5.1 Example function definitions

```
fun fact(nat) -> nat          % The factorial function
fact(0) -> 1
fact(N)::N1 =! N-1 & type(N1, nat) -> N * fact(N1)
```

Note we have the rather indirect way of testing `N>0` in the second rule. This is because use of the test `N>0` would not enable the type checker to infer that the value of the argument of the recursive call `N-1` was a natural number as required. However, the presence of the run-time test `type(N1,nat)` gives it this assurance. Input arguments of functions do not need to be moded. They are all implicitly `!` mode. That is, they must always be ground terms when the function is called.

`type` is a primitive of *QuLog* that allows the run-time type checking of any *QuLog* data value, or defined relation, function or action. It tests if the first argument belongs to a moded type. If the mode is not present it is assumed to be `!`.

```
fun max(num,num)->num
max(M,N)::M>=N -> M
max(M,N) -> N              % implicitly when not M>=N, i.e. when M<N

rel appBetween(list(E),E,list(E)) -> list(E)  % E is a variable, a poly-
morphic function
appBetween(L1,X,L2) -> L1<>[X]<>L

fun father_of(atom) -> atom
father_of(C)::child_of(C,P) & male(P) -> P
father_of(C)::not child_of(C,_) -> no_father_

fun number_of_children(atom) -> nat  % nat is type of non-negative integers
number_of_children(P) -> #{C :: child_of(C,P)}
```

`#` is a *QuLog* operator for finding the length of a list or string.

Auxiliary functions such as `father_of` can be defined for any use pattern of a relation that returns a single value, or a single tuple of values for some of its arguments, when certain other arguments are given.

`fun now() -> num` is a *QuLog* function primitive that accesses the host's clock and returns the current time as a float. The `now` function allows this varying time value to be found using a function call. This may make functional programmers aghast, but is useful for robotic applications where time is important and the result from an expression evaluation pertinent to a robot's behaviour may well be time dependent. Similarly,
`fun random_num() -> num` and
`fun random_int(int, int) -> int`
are functions that return random numbers and so get a different value on each call.

At the other extreme, *QuLog* has two primitive fixed value functions, `fun e()->num` and `fun pi()->num`. The first returns a numeric value for the constant `e`, the second a numeric value for π.

5.2 Expression evaluation as queries

The results of expression evaulation can be checked by creating a unification query in the interpreter. As we will see shortly, expression evaluation is done on arguments to relations (and actions) before the relation is called.

```
| ?? X = 1+2.
X = 3:nat

| ?? X = {1,-5,6} union {apple,1,4}.          % union is the set union
X = {-5,1,4,6,apple}:{atom||int}             % A set of atoms and ints
```

We have sets as a separate type from list with `union`, `diff` (first set minus the elements in the second) and `inter` (set intersection) for combining sets, and the `in` membership test for accessing their elements. Because of the way they are stored, testing whether a given term is in a set is more efficient than testing if its in a list of the same elements.

5.3 General form of a function rule

Functions are defined by sequences of condition/expression rules of the form

Head `::` *Commit* `->` *Expression*

The `::Commit` is optional. As with relation definitions it is a simple conjunctive query that augments the head/call pattern match to determine if that rule should be used. Unlike in most functional languages *Commit* may generate values for variables in *Expression* making the returned function

value partly depend on current facts. Even without a *Commit* test, there is an implicit *commit* at the `->` of each function rule.

Note the similarity between the form of conditional rules for relations and the form of a function rule. The difference is just whether `<=` or `->` is used followed by a conjunction or an expression.

The list of function rules make up the definition of the function except that the compiler adds a catchall rule that throws an exception if chosen. This means that if the function is called with an element not in its domain (as defined by head unifications and commit tests) then an exception will be thrown.

So, for example, if we take the `fact` function above but change its type to

```
fun fact(int) -> nat
```

and call `fact(-1)` then we will get an exception as this does not match either rule.

5.4 General form of function type declaration

Every function must have its type declared. A function declaration for `f` has the form

`fun f(`t_1`,...,`t_k`) ->` t

where t is the value type. The arguments are always unannotated. Unlike relations, function arguments must always be given as ground values.

Zero argument functions have a type `()->t` and must be defined and called using `f()`.

There is a comprehensive set of primitive functions: the usual arithmetic operators of Prolog, trigonometric functions, integer division, integer division remainder, `sqrt`, `exp` etc. All these can be called inside or as arguments to relation calls.

5.5 Expression use, implicit and explicit evaluation

Set expressions, list and string concatenation expressions, arithmetic expressions and any function calls can appear as or inside the term arguments of relation calls in queries and rule bodies, but *not* in rule heads. They are fully evaluated before the relation call is made. There is no lazy evaluation in *QuLog*.

As an example, we could have a call of the form

```
r([X::q(X,fact(Y)+2)])
```

where `Y` is a variable that must have been given an numeric value before the *rel* call. `q` will have a moded type such as `rel q(?int,int)` and `r` a moded type `rel r(list(int))`. Suppose `Y=3` at the time the `r` call is made and `fact(3)` will return `6`. `r` is called with argument the integer list value of `[X::q(X,8)]`.

We can also explicitly evaluate expressions. Two expressions can be evaluated and checked for identical values using `Exp1=:=Exp2`, *providing* they have the same type. The complement test `Exp1 =\= Exp2` is subject to the same type constraint. It can only be used to test non identity of values for primitive and program defined types. The argument expressions for `=:=` and `=\=` must both be ground before the comparison test will be evaluated, a constraint checked by the compiler. After evaluation of all function calls `=:=` and `=\=` respectively reduce to identity and non identity tests for the computed ground term values.

6 Primitive and Defined Types

6.1 Primitive Types

The primitive *QuLog* types are:

```
atom nat int num string atomic
list(T)  set(T)  (T1,..,Tk)  term
atom_naming(CodeType)  term_naming(CodeType)
```

`nat` is the natural number (non-negative) sub-type of `int`, the integer type which is in turn a sub-type of `num`, the type of all numbers. The `T`s are any type expression or a variable (for parameterised types). `list(T)` is the list of terms of type `T`. `set(T)` is the set of terms of type T, i.e. non duplicate terms. `(T1,..,Tk), k>1` is a tuple of `k` terms of the given, possibly mixed types. `term` covers every value that can be passed as an argument or returned as a value in *QuLog*. It includes every data term and every higher order value - functions, relations and actions. `atom_naming` and `term_naming` are the types of atoms and higher-order terms that name code. For example if we make the declaration

```
fun curry((T1, T2) -> T3) -> (T1) -> (T2) -> T3
```

then `curry` is of type
`atom_naming(((T1, T2) -> T3) -> (T1) -> (T2) -> T3)`
and `curry(/)` is of type
`term_naming((num) -> (num) -> num)`

As a shorthand we typically leave off `atom_naming` and `term_naming` and say
`curry` is of type `((T1, T2) -> T3) -> (T1) -> (T2) -> T3` and
`curry(/)` is of type `(num) -> (num) -> num`

There are problems using tuples in Prolog because of associativity of ','. This is needed because it is also used as the 'and' connective in rules. Prolog converts what should be a three element tuple `(2,big,(4.7,"hello"))`, containing a last element that is a pair, into `(2,big,4.7,"hello")`, a four element tuple. But `(2,big,(4.7,"hello"))` remains a three element tuple in *QuLog*, with a two element tuple as last component.

If we have a relation or function that has one argument, that is a tuple, in its type declaration we use `((T1,..,Tk))`. `rel((atom,int))` is the type expression for a unary relation with argument a pair that can be given or generated by a call to the relation. The type expression `((atom,int))->int` is for a function of one argument that is a pair.

`nat < int < num < atomic < term`
`atom_naming(`*CodeType*`) < atom < atomic`
`string < atomic`
`list(T)` $\leq$ `list(T')` `if T` $\leq$ `T'`
`set(T)` $\leq$ `set(T')` `if T` $\leq$ `T'`
`list(T) < term`
`set(T) < term`
`atom_naming(`*CodeType*`) < term_naming(`*CodeType*`) < term`
$(T_1,..,T_k)$ `< term`
$(T_1,..,T_k) \leq (T'_1,..,T'_k)$ `if T1` $\leq$ T'_1 `& ... &` $T_k \leq T'_k$

If `term` is used as the only data type, the *QuLog* program is essentially untyped like Prolog. In fact, as mentioned in Section 2.2, a mode annotation `m` with no type is an abbreviation for `m term`. The programmer must then do run-time type checking using the `type` primitive, as described in Section 6.5, before calling any builtin relation or function such as an arithmetic primitive that requires a more specific type.

6.2 Program defined types

The primitive types can be extended with program defined types including parameterised and recursive *constructor* types, *enumerated* types (sets of atoms), *range* types (sets of successive integers), *union* or *disjunctive* types (the values can be from any one of a set of alternative types). All defined types are sub-types of `term`. Examples are given below.

Different enumerated types must either be disjoint, or one must be a subset, hence a sub-type, of the other. Different range types must either be disjoint, or one must be a subset, hence a sub-type, of the other. The constraint regarding not being partially overlapping is checked by the type checker and it issues an error message if flaunted. This is because in *QuLog* every term must belong to just one minimal type in the sub-type lattice.

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

This defines a new type parameterised constructor type `tree(T)`, with labels which are any primitive or program defined data type `T`. `empty` is a 0 argument constructor of `tree` terms, and `tr` is a 3 argument constructor.

```
def gender ::= male | female
          % enumerated type gender of two atoms, is a sub-type of atom
             % it is not a constructor type as the alternatives are atoms
def age ::= 0..110
        % defines age type as a range of nat values, is a sub-type of nat
def digit ::= 0..9
        % is a sub-type of age, hence of nat, int, atomic, term
def genage == gender || age
        % defines genage as a macro for a union type of age nats and
        % gender atoms
        % [male,67,female,89] has type list(genage), a list of genage terms
        % whereas [(male,57),(female,89)] has type list((gender,age))
```

We use `|` to separate the alternatives of an enumerated type and a constructor type as both are alternatives between data *values*. We use `||` for separating the alternatives of a type union as they are alternative data *types*.

6.3 Data constructors do not have a function type

We do not view `tr` as a function of type `(tree(T),T,tree(T))->tree(T)` which Haskell does, nor do we view `empty` as a function of type `()->tree(T)`. `tr` is not a function in the normal sense of returning a value *different from*

the call term. `tr` is what is termed a *free* function. Its role is just to wrap its name around its arguments. We do not want `tr` to be passed as an argument value where what is required is a non-wrapper function of type `(tree(T),T,tree(T))->tree(T)`. In *QuLog* `tr` has type `atom`.

There is another more important reason for not assigning any type to names of data constructors. In order to do meta-programming in *QuLog* we need to have data terms 'naming' relation and action procedure calls. We can do this in a type safe way by having a double role for the program given names of the relations and action procedures as constructors of two special meta-types `relcall` and `actcall`. If constructors had function types, this simple way of 'naming' calls with template data terms that use the relation and action procedure names as data constructors would not be possible. Section 7.4, shows a use of the `relcall` type to program a very simple query evaluator that evaluates a list of `relcall` terms. It is example of what might be used inside an agent, with its behaviour programmed using *QuLog* action rules, to answer queries from other agents, as will be shown in the companion paper.

As an example, since `children_are` is the name of a relation of type `rel(?atom,?set(atom))`, there will a constructor `chid_is(atom,{atom})` of the special meta-type `relcall`. If, because of this, `children_are` had to have the type `(atom,set(atom))->relcall` we would have a clash with its already declared program type as a relation of moded type `rel(?atom,?set(atom))`. By not assigning a function type to each of a constructor type's constructor functors we avoid this clash. Since relation and action names automatically become constructors of the compiler generated `relcall`, `actcall` types respectively, their names *must not* be used in any programmer defined types. If used, there will be a type error.

6.4 Sub-type relation for defined types

For a parameterised type such as `tree` we have

`tree(T) < term` $\qquad$ `tree(T) < tree(T')` *if* `T < T'`

The general sub-type relationships for defined types *Type*, *Type'* is

$$Type(\mathtt{T}_1,\ldots,\mathtt{T}_n) \leq Type(\mathtt{T}'_1,\ldots,\mathtt{T}'_n) \text{ if } \mathtt{T}_1 \leq \mathtt{T}'_1,\ldots,\mathtt{T}_n \leq \mathtt{T}'_n$$

$$\begin{aligned} Type \leq Type' \text{ if } & Type\mathtt{::=}\ atom_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ atom_m\ \mathtt{\&} \\ & Type'\mathtt{::=}\ atom'_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ atom'_n\ \mathtt{\&} \\ & \{atom_1,\ldots,atom_m\} \subseteq \{atom'_1,\ldots,atom'_n\} \end{aligned}$$

$Type$ `<` `atom` *if* $Type$`::=` $atom_1$ `|` ... `|` $atom_m$

$Type \leq Type'$ *if* $Type$`::=` $I..J$ `&` $Type'$`::=` $M..N$ `&` $M \leq I$ `&` $J \leq N$

$Type$ `<` `int` *if* $Type$`::=` $M..N$

$Type \leq Type'$ *if* $Type$`==` T_1`||...||`T_m `&`
$Type'$`==` T'_1`||...||`T'_n `&`
$\forall$`i(1≤i≤m)`$\exists$`j(1≤j≤n)` $T_i \leq T'_j$

In the first rule *Type* is the name of a parameterised constructor type with `n` argument types.

6.5 Run-time type checking

A run-time type checking primitive can be used to check if an expression, which could be a variable, has a value of a specific type - either primitive or programmer defined. It is useful for checking if a value is a particular sub-type of an expected argument type.

`type(`*Exp*`,`*!Type*`)`

in a program clause or function rule will check that the *Exp* has a ground value of type *Type*, where *Type* is a ground type expression - so no type variables.

Similarly, `type(`*Exp*`,`*?Type*`)` will check that the *Exp* has a value of type *Type*, but may be a variable or include variables.

One extreme of *QuLog* programming has no user defined types and makes much use of the `term` type and run-time type checks. This style is not recommended as it trades run-time failures of type testing conditions for compile time indications of a type error. If `term` is used as a catch all type, the programmer *must* insert type tests before the use of any *QuLog* primitive that requires a specific type, such as a `num`. For example, any call to an arithmetic function or arithmetic comparison relation must be preceded by runtime tests that ensure that the arguments will be numbers.

The type checker will take into account the occurrence of the run-time type tests acting as filters of all values that are not of the tested type to make sure that *QuLog* primitives can only be passed arguments of their required types. If they are not present it will give a type error. This trades the compile time checking of programs that use more constrained type declarations,

for run-time checks that will fail if an incorrect type of value would have been passed into a call to a primitive. The programmer then has to have extra rules that handle these run-time failures of clauses due to type test failures. Far better to let the compiler do all the work of type validation. The best approach is to declare as constrained a type as possible for each relation and function argument, and to avoid uses of run-time type tests except to select out sub-types of declared argument types, either before a call that requires a sub-type, or after a call that might generate a super-type of the type that must be returned from a clause. These type filtering uses of run-time type tests are discussed in Section 10. A test

`type(`*Exp*`,`*Type*`)`

is the same as having the prefix `!` on *Type*.

If `type` is used in a single condition top level query in the interpreter, `Type` may be a variable which will be bound to the inferred type of `Exp`. This is useful for learning about the type system.

There is also a three argument `typeC`, which can only be used as a single condition interpreter query. This will tell us the types that any variables in `Exp` must have for `Exp` to have type `T`. The third argument of a `typeC` call *must* be a variable. It will be bound to a list of type constraints for any variables in `Exp`, as exemplified below.

6.6 Some example uses of run-time type checks

As an example, this is the program for the factorial function defined as a relation `factr`, where we do not use the `nat` type in its declaration. By giving no type the default type `term` for both arguments us assumed.

```
rel factr(!, ?)
factr(0,1):: true
factr(N,FN)::type(N,nat) & N1 =! N-1 & type(N1,nat) <=
                factr(N1,FN1) & FN =! FN1*N
factr(N,0)                % The rule for N not a non-negative integer
```

Without the run-time `type(N,nat)` test, the type checker will reject this definition as not being type safe. Note the last rule returns the value `0` for any non-nat first argument. If we had used the type declaration `rel factr(!nat,?nat)` the relation would never be called with a non-nat value as that would be a type error picked up at compile time. We then only need the `type(N1,nat)` type test as for the `fact` function definition.

If a relation has no type declaration at all, it is assumed to have been declared with `term` as each argument type with implicit mode `?`. Typically the programmer must then also use the `type` primitive with a `!` prefixed type that checks that the argument is ground as well as of the given type.

Using run-time type tests is similar to what one would do in Prolog before using a primitive to do addition, if the argument might not be a number. However, in the case of *QuLog* the use of the type check is *not optional*. It has to be in the rule else a type error will be flagged. The type checker must know that a variable holds a number value before arithmetic can be done using its value. The presence of the explicit `type(N,num)` test in the second rule below enables it to infer that `N` will have the `num` sub-type required for addition operation of this second rule.

```
rel add_nums_of_list_of_any_term(!list(term),?num)
        % term is any data term so first arg can be a list of very mixed types
add_nums_of_list_of_any_term([],0)
add_nums_of_list_of_any_term([N|Rest],Total) ::
   type(N,num) <=                % run-time check that N is a number
     add_nums_of_list_of_any_term(Rest,RTotal) &
     Total =! RTotal+N
add_nums_of_list_of_any_term([NonN|Rest],Total) <=
      not type(N,num) &
      add_nums_of_list_of_any_term(Rest,Total)
```

The use of the `::` test in the second rule does not affect the logical correctness of the last rule because it has the complement to the commit test of the second rule as a precondition. Of course, as in Prolog, one is tempted to drop complement tests in rules following a commit rule if these would require significant search to show they are not inferable.

An example query is

```
| ?? add_nums_of_list_of_any_term(
                    [3,bill,"sally",[2,-19],0.6,empty()], Sum).
Sum = -16.6:num
```

Here the given list will have inferred type

```
list(atomic||list(int)||tree(T))
```

as a list of any of the above types of terms. This is a sub-type of `list(term)` so the call is type correct.

6.7 Type expressions as first class values

As *QuLog* has the built-in meta-type `typeE`, we can pass in the type expression to be used for a run-time `type` test as an argument. Here is an example of a filter function that removes all but values of some given type from a list of terms.

```
filter: (typeE(T),list(term)) -> list(T)
filter(_TypeE,[]) -> []
filter(Type,[Trm,..Trms]) :: type(Trm,Type) ->
                                  [Trm,..filter(Type,Trms)]
filter(Type,[_,..Trms] -> filter(Type,Trms)

| ?? X = filter(int,[3.4,hello,-6,"happy",2]).

X = [-6,2]:list(int)
```

The role of the parameter `T` of the type expression `typeE(T)` is to allow us to inform the type checker that the value returned by the function will be a list of the type given as the first argument of a call to the function.

We give more examples of the use of run-time type checks in Section 10.1.

6.8 Generating instances of finite types

`digit` and `gender` are special program defined data types in that they each comprise a finite set of data values. In the case of `gender` it is the set of atoms {`male, female`} and in the case of `digit` it is the set {`0,1,...,9`}. For enumerated and range types, and any other types that have a finite set of values, for example unions of enumerated types, we can use another primitive `isa(E,T)`. It will generate, one at a time as successive bindings for an unbound variable `E`, all the values of the type `T`, in the order that they appear in the enumerated or range type definition of `T`. If `E` is given, it will also test that it is a value of type `T`.

For example,

`[D | isa(D,digit)]` evaluates to `[0,...,9]`
`[G | isa(G,gender]` evaluates to `[male,female]`.
`[G | isa(G,genage]` evaluates to `[male,female,0,...,110]`.

If `L=[23,8,-4,5,61]`,

`[D | D in L & isa(D,digit)]` evaluates to `[8,5]`.

In the last expression, `isa(D,digit)` could be replaced by `type(D,digit)`. `isa` can be used for testing or generating instances of finite types. `type` can only be used for testing but it can test for any type, including higher order types.

6.9 Example type queries

```
| ?? type((tr(empty(),-3,empty()),!tree(int)).
yes

| ?? type(tr(empty(),-3,empty()),T).
T = tree(int)

| ?? typeC(tr(L,a,R),T,Cs).
T = tree(atom),
Cs = [L:tree(atom),R:tree(atom)]

| ?? type([1,2,male,11],T).
T = list(genage)
```

We get `genage` rather than `list(int||atom)` as all the given integers are in the `age` range of values, `male` is an atom of the `gender` type and `genage` is defined as union type `gender||age`.

However, a type checking query will *confirm* that `[1,2,male,11]` has type `list(age||atom)`, `list(int||atom)`, `list(num||atom)`, `list(atomic)`, `list(term)`. A sub-type can always be passed into a function or relation requiring a more covering type.

7 Relations and functions over defined types

7.1 Relations defined by type validated fact sequences

Range types and enumerated types are useful for data checking. They allow us to declare a tight schema for data facts that is checked as the facts are read in from a consulted file.

Suppose that instead of defining the `person` relation in terms of `male`, `female` and `age_of` relations, it was instead an updateable belief relation. We can use the enumerated type `gender` and the range type `age` to have these facts data validated.

```
dyn person(atom,gender,age)
person(bill,male,58)
person(johnny,male,25)
person(cheryl,female,23)
...
```

A fact `person(bill,mle,34)` or `person(john,male,120)` will be rejected when the *QuLog* program file is consulted. `mle` is not a `gender` atom and `120` is outside the defined range for `age`.

If we query this relation the answers will be annotated with these new types.

```
| ?? 2 of person(P,G,A) & A>=25.
P=bill:atom
G=male:gender
A=58:age
...
P=johnny:atom
G=male:gender
A=25:age
```

Suppose we also try to update a fact for this `belief` relation to record a birthday.

```
| ?? forget person(bill,male,Age) remember person(bill,male,Age+1)
```

On the surface this looks fine but it will produce a type error. The reason is that the last argument of the `person` relation must be of type `age` which is a range type with maximum value 110. The type/mode checker cannot know the value of `Age` stored in the `person` fact for `bill`, and this could be the maximum age 110. If it were then `Age+1` is not a valid `age` integer.

We must use a runtime type test and enter the update command in a more verbose way.

```
| ?? ?(person(bill,male,Age) & NewAge = Age+1 & type(NewAge,age));
      forget person(bill, male, Age) remember person(bill, male, NewAge).
```

Since the query contains the forget/remember action then we need to turn the test into an action. That can be done by wrapping the test in `?`. This throws an exception if the test fails.

To give you a taste of the use of action rules to define new actions, here are two action rules that will define a new action for updating the age of a person as recorded by a `person` fact.

```
update_age: atom
update_age(P) :: person(P,G,A) & NewA = A+1 & type(NewAge,age)  ~>
     forget person(P,G,_) remember person(P,G,NewA)
update_age(P) ~>
     writeLine([P,' has no recorded age or has maximum age'])
```

~> can be read as *do*. The commit test of the first rule checks if the given P has a recorded age that can be incremented to a valid age. Only if that test fails will the second rule be used.

7.2 Recursion over a defined recursive data type

The label membership relation for the recursive `tree(T)` data type is the parameterised recursive relation

```
rel on_tree(?T,tree(T))
on_tree(E,tr(_,E,_))
on_tree(E,tr(Left,_,_)) <= on_tree(E,Left)
on_tree(E,tr(_,_,Right)) <= on_tree(E,Right)
```

Note that the variable `E` of the first rule that appears in an `?` argument position also appears in the term `tr(_,E,_)` in the ground input argument position. This enables the mode checker to confirm that any variable given as the first argument of a call will be bound to a ground value - the root label of some sub-tree of the ground tree argument - if the call succeeds.

```
fun flatten(tree(T)) -> list(T)
flatten(empty()) -> []
flatten(tr(Left,E,Right)) -> flatten(Left)<>[E]<>flatten(Right)
```

Remember <> is the list concatenation function.

7.3 Complete skeleton structures containing variables

In Section 2.3 we defined the `doubleL` relation with type `rel(list(num),?list(num))` and gave the example use

```
doubleL([1,3.5,-2.1,4], [2,N2,N3,..R])
```

producing the bindings `N2=7.0`, `N3=-4`, `R=[8]`.

If we wanted to insist that the second argument was always a complete list of variables or numbers, such as `[2,N2,N3,8]`, we can type the relation using

```
rel doubleL(list(num),list(?num))
```

`list(?num)` (implicitly `!list(?num)`) is the moded type expression for a complete list of numbers, *or variables*, of type `num`. We can have this sort of qualified grounded mode for any parameterised type. Where the type is a recursive type as here, the implicit outer `!` tells us that the recursive structure - the skeleton of the term - is fully given. If the type of its components - elements on a list or labels on a tree - is unannotated then they must all be given as ground values. The element type is also implicitly `!` moded. However, the element type may be `?` prefixed, or, if it is also a structure type, it can be have a `??` mode. So, `list(?tree(int))` allows the argument to be a complete list of variables or partially given `tree` terms, all of which will become ground by a call to the relation. `list(??tree(int))` allows the call argument to be the same, and says it may remain a list of partial integer tree terms even after a successful evaluation of the call.

7.4 Meta-programming

Meta-programming is the idea of treating programs as data and conversly "executing data". In Qulog we can construct terms that represent programs and then being able to execute those programs.

To support this we have a small language for representing programs - particularly the treatement of quantifiers and negation. We can then call these programs using `call` (for executing relational queries) and `do` (for executing action sequences). Both take two arguments: a term in this meta-programming language and a variable that will be instantiated to a string that is either the empty string if type and mode checking and other tests succeeds or a string representing the failure of these tests.

For example

```
| ?? call(all([X], in(X, [1,2]), in(X, [1,2,3])), Err).

Err = "" : string

| ?? call(all([X], in(X, [1,2,3]), in(X, [1,2])), Err).

no

| ?? call(in(X, [1,2]), Err).

X = 1 : nat
```

```
Err = "" : string
...
X = 2 : nat
Err = "" : string

| ?? call(sort(a, Y, @<), Err).

Err = "Type Error: a has type atom but is required to be of type list(@T)
            in condition sort(a, Y, @<)
            ( with sort : rel(!list(@T), ?list(??T), !rel(@T, @T)),
                                rel(!list(!T), ?list(T), !rel(!T, !T)))" : string
```

The first query is equivalent to

```
forall X (X in [1,2] => X in [1,2,3])
```

This query succeeds with no type/mode error.
The second query fails.
The third query succeeds with multiple solitions.
The forth query succeds but this is because there is a type error.

Why would we want to support meta-programming? The first reason is that we might write a program that, for example, constructs an action to execute based on the the state of the belief store. It might make sense for the particular application to first construct a term representing the action and then to call that.

Another possibility is we have a process running that manages a belief store and we might want clients to send messages to the process to update its belief store or to query its belief store. The messages will arrive as terms (in the meta-programming language) and then be executed in the process using `call` or `do`.

8 Higher order functions and relations

Here are the *QuLog* definitions of `mapF`, `curry` and `uncurry` for functional programming enthusiasts with example expression queries. As `->` is right associative not all the brackets are necessary in the type declarations, but using them resolves any ambiguity.

```
fun mapF((T1->T2), [T1]) -> [T2]
mapF(F,[]) -> []
mapF(F,[U|L]) -> [F(U)|mapF(F,L)]

fun curry((T1,T2)->T3) -> (T1->(T2->T3))
curry(F)(A)(B) -> F(A,B)

fun uncurry(T1->(T2->T3)) -> (T1,T2)->T3
uncurry(F)(A,B) -> F(A)(B)

| ?? X = mapF(curry(+)(2),[3,4,5]).
X = [5,6,7] : list(digit)
```

`curry(+)` is a function of type

```
nat -> (nat -> nat) | int -> (int -> int) | num -> (num -> num)
```

`curry(+)(2)` is a single argument function that adds 2 to any supplied number. `mapF` takes this function and apples it to each value on the list of numbers second argument, so producing the list in which each number is 2 greater than the corresponding number on the argument list.

`uncurry(curry(+))` gives back the original two argument function `+`. But `uncurry` will convert any function of type `T1->(T2->T3)` to a function of type `(T1,T2)->T3`, even if that function is not the result of currying.

For comparison, we also define `mapR`, `curryR` and `uncurryR`. Currying a relation is a little odd because it generates a function that returns a one argument relation. So its type is that of a higher order function, but its definition is a relation rule telling is how to evaluate a predication with a `curryR` function call value as its relation.

```
rel mapR(rel(?T1,?T2), ?[T1], ?[T2])
mapR(R,[],[])
mapR(R,[U|L], [RU|RL]) <= R(U,RU) & mapR(R,L,RL)

fun curryR(rel(T1,?T2)) -> (T1->(rel(T2)))
curryR(R)(A)(B) <= R(A,B)

fun uncurryR(T1->rel(?T2)) -> rel(T1,?T2)
uncurryR(F2R)(A,B) <= F2R(A)(B)
```

`mapR`, as typed, can be used either to generate or test a pair of lists of the

same length such that corresponding elements are in the `R` relation. Either list may be given, or neither list.

An example use of `mapR` is

```
mapR(child_of,[bill,C],[P,peter])
```

A call

```
mapR(child_of,[bill,mary],[june,peter])
```

will check that each person in the first list is a child of the corresponding person in the second.

`curryR(child_of)` is a function that we can apply to a person `P` that returns a unary relation `curryR(child_of)(P)` that can be used to test if another person is a child of `P`. For example `curryR(child_of)(mary)` is a unary relation for checking if some person is a child of `mary`. This unary relation can be passed around as a value. For the sample facts we gave in Section 2.3, `curryR(child_of)(mary)(peter)` is inferable.

`uncurryR(curryR(child_of))` denotes the binary relation `child_of`. `uncurryR(curryR(child_of))(mary,peter)` holds.

Note that `curryR` is defined as only requiring a relation argument of type `rel(T1,?T2)`. It is used with a relation of type `rel(?atom,?atom)`, which is not an instance of the polymorphic type expression `rel(T1,?T2)`. This is allowed as the `?` mode for the first argument allows the required `!` mode of use. The relation type `rel(?atom,?atom)` is a sub-type of the instance `rel(atom,?atom)` of the `rel(T1,?T2)` type pattern. Hence `child_of` may be given as the relation argument of `curryR`. The sub-type relationship for functions and relations is defined in Section 9.

8.1 Run-time handling of higher order values

Calls to functions that return a function or relation are special in that they do not get evaluated in the way that a call to a normal function gets evaluated. A call such as `fact(3)` gets replaced by its value `6`. However, an application of a function such as `curry` or `curryR` that return a higher order value is left as a wrapper term around its argument, in the same way that application of a data constructor such as `tree` is left as a wrapper around its arguments. In both cases the value is a functor term that is passed around. So, the function *value* of `curryR(child_of)` is actually passed around as the function application term `curryR(child_of)`.

When a curried function or relation gets called, as in

```
curryR(child_of)(mary)
```

this is still not 'evaluated'. What is then passed around, denoting the unary relation of type `atom<=`, is the above unevaluated complex term. It is only when this term denoting the one argument relation that tests if a person is a child of `mary` is called, say as in an `R(C)` condition of a rule in which `R`, typed has `(atom)<=` has be assigned the 'term' value `curryR(child_of)(mary)`, that the resulting higher order call expression

```
curryR(child_of)(mary)(C)
```

which `R(C)` becomes, actually call some code.

First the rule `curryR(R)(A)(B) <= R(A,B)` of the `curryR` definition is applied to reduce this complex call to the call `child_of(mary,C)`. Then the relation `child_of` is queried.

8.2 Example higher order expression queries

When the expression value is a function or relation its type is returned as the value.

```
| ?? X = curry.

X = curry : atom, ((Ty1, Ty2) -> Ty3) -> (Ty1) -> (Ty2) -> Ty3

| ?? X = curry(+).

X = curry(+) : (nat) -> (nat) -> nat, (int) -> (int) -> int,
               (num) -> (num) -> num

| ?? X = curry(+)(1).

X = curry(+)(1) : (nat) -> nat, (int) -> int, (num) -> num


| ?? X = curry(+)(1)(12).

X = 13:age                    % Again minimum inferable type for value is given
```

```
| ?? X = mapF(curry(+)(1), [1,2.3,3]).
X = [2,3.3,4]:list(num)

| ?? curryR(child_of).
curryR(child_of): atom->((atom)<=)
                    % A function from an atom to a relation over atoms
| ?? X = curry(<>).
 = curry(<>) : (list(Ty1)) -> (list(Ty2)) -> list(Ty1 || Ty2)

| ?? X = curry(<>)([1,2]).

X = curry(<>)([1, 2]) : (list(Ty1)) -> list(Ty1 || digit)

| ?? curry(<>)([1,2])([a,b]).
[1,2,a,b]:[digit||atom]

| ?? curry(<>)([1,2])([-3]).
[1,2,-3]:[int]
```

The first of the of the last four expression queries tells us that `curry(<>)` is a polymorphic function from a list of any type `A` to a function from a list of the same *or* different type `B`, to a list of type `A||B` elements. The second tells us that calling this function with a list of digit integers will return a function from a list of any type `A` to a list with elements of type the union of `digit` and `A`. This allows us to *later* append to `[1,2]` a list of atoms, as in the expression `[1,2]<>[a,b]`. We can do this because, although `list(digit)` is the minimal type of `[1,2]`, and `list(atom)` is the minimal type of `[a,b]`, both are sub-types `list(digit||atom)`. In the last query we append a list of integers so we have a list of integers value. `list(digit||int)` is simplified to the equivalent `list(int)` type.

9 Sub-type relationships for functions and relations

When we have a higher order relation or function which has arguments that are themselves relations or functions, the type and mode checker makes sure that a relation or function with a type compatible with the required type is always passed in as an argument, or returned as a value. The higher order value will be type compatible if its type is in the following higher order $\leq$

relation to the required type.

$(t_1,\ldots,t_k)\ \texttt{->}t \leq (t'_1,\ldots,t'_k)\ \texttt{->}t'\ \textit{if}\ t_1 \geq t'_1\ \texttt{\&...\&}\ t_k \geq t'_k\ \texttt{\&}\ t \leq t'$

That is, the compatible function must be able to be given the same or a greater type for each of its arguments, and return a value of the same or lesser type.

For relation types we give the sub-type relation where every type is explicitly moded with a prefix mode.

$(mt_1,\ldots,mt_k)\texttt{<=} \leq (mt'_1,\ldots,mt'_k)\texttt{<=}\ \textit{if}\ mt_1 \leq mt'_1\ \texttt{\&...\&}\ mt_k \leq mt'_k$

$\texttt{!}t \leq \texttt{!}t'$ *if* $t \geq t'$ *% test only use of super-type ok for test only use*
$\texttt{?}t \leq \texttt{!}t'$ *if* $t \geq t'$ *% test/gen, for super type, ok for test only use*
$\texttt{??}t \leq \texttt{!}t'$ *if* $t \geq t'$ *% even if a non-grounding generator*
$\texttt{?}t \leq \texttt{??}t$ *% grounding rel., same type, may be used if non-grounding is ok*
$\texttt{!}t(met) \leq \texttt{!}t(met')$ *if* $met \leq met'$ *% t a parameterised structure type*
% moded element type met of structure tester should cover all met′ uses
$\texttt{?}t(?et) \leq \texttt{?}t(\texttt{??}et)$ *% may use a grounding rel for same element type if only*
% non-grounding use for elements required, for test/gen of structure

The above tells us that the compatible relation or action procedure must have the same or a lesser *moded* type for each of its arguments. This *lesser* relation depends on both the type and the mode. The sub-type relationship between the argument types can flip from $=$ to $>$ to $<$, depending upon whether the higher order argument value may be used to both generate and test ($=$), or is only used to test ($>$), or is only used to generate ($<$), a particular argument.

Suppose a function argument is typed `(atom,num) -> num`. A function typed `(atomic,atomic)->int` has a compatible type, so could be passed as an argument. If a relation argument is typed `(!string,!int,!atom)`, a relation moded `(?string,?num,!atomic)` can be used in the way the relation argument will be used, and may be passed as an argument value.

The fourth inequality is for parameterised structure types like list and trees, where the outer structure must be complete, but values for structure's elements, such as the elements of a list or the labels of a tree, may be given as variables or non-ground terms. The moded element types must satisfy the moded sub-type relation.

As an example, if an argument of a higher order relation is declared as being of type `rel(list(int))` (implicitly `rel(!list(!int))`) then we can

pass a relation of type `rel(list(?num))`, (implicitly `rel(!list(?int))`), as the value of that argument. However, if the argument type is `rel(list(?int))`, we cannot pass in a relation of type `rel(list(?num))` in case the passed in relation is used to generate values for the list elements, which must be of type no greater than `int`.

The last rule is for relations which may be used to test, or generate, a complete structured term. It says that a relation typed `rel(?list(?tree(num)))` is an allowed value for an argument of type `rel(?list(tree(num)?))`, as it does not matter that the passed in relation will always generate a ground structure for the list elements - trees of numbers.

10 Moded type checks, declarative reading and sound inference

A variable *will have* a *ground* value if it appears in the head of the function rule, or in a `!` argument position in the head of the clause, or immediately after a call in which it appears in a `?` argument position.

It *may have* a ground if it only appears in the head of a clause in `?` or `??` argument terms, and only appears in earlier relation calls in `??` moded argument positions.

The *initial* type of a variable is the of the type its occurrences in a rule head, or, if not in a rule head, the type of its occurrences in the first relation call in which it appears. This `initial type` then becomes its *current* type for its next call use.

The current type of variable `V` changes to be a sub-type $\texttt{t}_V$ of its current type whenever it appears in a type test `type(V,`$\texttt{t}_V$`)`. A warning is issued that the type test will always fail if $\texttt{t}_V$ is not a sub-type of `V`'s current type immediately before the type test.

The type of the occurrences of a variable in a predication is the minimum of all its occurrence types. The minimum is only defined if they can be linearly ordered. For example, if the types are `atomic` and `atom` the minimum is `atom`. For the types `int` and `atom` there is no minimum type. When a minimum type for occurrences of a variable cannot be computed, it is a type error.

Using the above rules for when a variable is ground, or may be ground, and for determining its current type, the *QuLog* type checker does the following checks as it moves left to right along a conjunction of relation calls.

1. If a variable appears in `!` argument positions in a clause head, with

type $\mathtt{t}_I$ for these ground input occurrences, and it also occurs in other argument positions, the type $\mathtt{t}_{IO}$ of each non-input occurrence must be such that $\mathtt{t}_I \leq \mathtt{t}_{IO}$.

2. Each variable will have a ground value before it is used in a function call or a `!` argument of a relation call.

3. After it is determined a variable `V` has a ground value, all its later call uses have a type $\mathtt{t}_O$ such that $\mathtt{t}_O \leq \mathtt{t}_V$, where $\mathtt{t}_V$ is its current type just before the use.

4. After it is determined a variable `V` may have a ground value, all its later call uses have a type $\mathtt{t}_V$, where $\mathtt{t}_V$ is its current type just before the use.

5. Each variable `V` in a `?` argument position in the head of a clause will have a ground value at the end of the clause body.

6. Each `not C` condition will have ground values for all but anonymous `_` variables when evaluated

7. In each *forall*, all except the quantified variables, `_` variables and any `exists` explicitly quantified variables will have ground values when it is evaluated. In addition, the two conditions of the *forall* satisfy the normal type and mode constraints when viewed as queries.

8. In a comprehension containing `Term::exists Vars Conds`, all variables in `Conds`, except `_` variables and variables in `Term` and `Vars`, *and* any variables in `Term` not in `Conds`, will have ground values before the set expression is evaluated. In addition, `Conds` satisfies type and mode constraints when viewed as a query.

All these checks are made by a single pass algorithm over each clause, function rule and query. If any check fails, a type or mode use error is signalled giving the reason for the failure and the compiler terminates.

The last three checks ensure that *QuLog* rules for relations have a straightforward declarative reading as sentences of predicate logic enhanced with functions and set expressions. These rules are implicitly universally quantified with respect to all their variables except: the inner universally and existentially quantified variables of a *forall*, the local variables in the template terms of set comprehension expressions and any existentially quantified variables of its generator condition, and any `_` variables. The `_` variables are

implicitly existentially quantified immediately before the condition in which they appear.

A *QuLog* query evaluation of relational and expression queries is an inference that is sound with respect to that declarative reading, providing:

- A `::` commitment test in a rule is only used to prevent unnecessary attempts to use later rules because their use would definitely fail. (In Prolog terms, each `::` is a so called 'green' cut).
- Equality (=) of higher order values is understood as identity of the expressions denoting the higher order values *not* equality of their extensions as sets of tuples of data values. It is identifier unification not denotation equality.

10.1 Need for runtime type checks to filter out sub-types

Ideally, the programmer has designed his program so that all the above checks are passed without the need for the use of run-time `type` or `ground` tests. Sometimes this is not possible.

Here is an example of the need for a type test before a call. Suppose we have

```
rel produce(?atomic)
rel use(!num)
```

The conjunction of calls

```
produce(A) & use(A) ...
```

will generate a compile time type error message. However,

```
produce(A) & type(A,num) & use(A) ...
```

is acceptable, as is

```
produce(A) & type(A,int) & use(A) ...
```

The second conjunction would be used if a later call in the conjunction required `A` to have type `int`. This only makes sense if `produce` will generate some values of the sub-types `num`, `int` of `atomic`, which is a union of `nat`, `int`, `num`, `atom` and `string`. The type test then forces backtracking until such a value is generated, as this is what must be passed to the later call(s). The type test acts as a filter of the values of the super-type that might be generated by the `produce` call.

As an example of the need for a type test where the variable only may have a ground value, suppose we have

```
rel r(?num)
rel q(?int)
```

The clause

```
r(N) <= q(N)
```

must be written as

```
r(N) <=  type(N,int) & q(N)
```

else there will be a type error signalled for this clause identifying the `q` call as the culprit. We need the `type` test in case `N` is given in the call to check that it is an integer. If not given, the test will just succeed and its effect is simply to satisfy the type checker that only `int` values will be passed into the `q` call. The integer binding for `N` that will then be generated by `q` satisfies the promise that a `num` value will be retuned by a call to `r`.

An example related to check 4, is

```
rel p(..,?tree(int),..)
rel q(..,??tree(int),..)

p(..,Tr,..) <= q(..,Tr,..)
```

The clause is not mode correct as `q` may not generate a ground `tree`. We must write the clause as

```
p(..,Tr,..) <= q(..,Tr,..) & ground(Tr)
```

The `ground` test will cause backtracking to retry the `q` call until it returns a ground `tree` of integer values. This only makes sense of `q` may generate several answer bindings for `Tr` some of which will be ground. If `Tr` was given as a ground tree when `p` was called, the test does no harm.

Here is a final example where only `??` modes are involved.

```
rel p( ..,??tree(num),..)
rel q( ..,??tree(int),..)

... p(..,Tr,..) & type(Tr,?tree(int)) & q(..,Tr,..) ...
```

is type and mode correct as only a ground or partial tree of integers can be passed through to the call to `q`. There will be backtracking to generate an alternative `Tr` value using the `p` call, should the type test fail. Without the `type` test it is not type correct as `q` could be given a tree containing non-integer values. If the `tree` argument of `q` had `!` ground input mode we would have to use

```
... p(..,Tr,..) & type(Tr,!tree(int)) & q(..,Tr,..) ...
```