Qulog Reference Manual

Keith Clark and Peter J. Robinson

March 3, 2015

Abstract

QuLog is a higher-order logic/functional/string processing language with an imperative rule language sitting on top, defining actions. QuLog's action rules are used to program multi-threaded communicating agent behaviour. Its declarative subset is used for the agent's belief store. The language is flexibly typed and allows a combination of compile time and run-time type checking.

Contents

0 Overview of QuLog

QuLog is a higher-order logic/functional/string processing language with an imperative rule language sitting on top, defining actions. QuLog's action rules are used to program multi-threaded communicating agent behaviour. Its declarative subset is used for the agent's belief store. The language is flexibly typed and allows a combination of compile time and run-time type checking.

It is a fully integrated language in that function calls can appear as or inside arguments to relation calls, and relational queries can be used as guards of function rules. It has sets as a separate data type from lists with set ¡-¿ list convertors. Both can be created using Trm::Query comprehension expressions.

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

QuLog supports type safe meta-level programming to complement its type safe higher order programming. However there are no lambda expressions in QuLog. All code has to be named and defined in the top level sequence of type definitions, type declarations and relation, function and action defining rules. At this time QuLog has no module system, so each consulted file must use different type and code names for its definitions.

As mentioned above, using its action rules and action primitives multi-threaded message communicating agent applications can be created with the agents communicating using the companion Pedro publish/subscribe and destination addressed communications server. Such an agent application can also receive and send MQTT notifications routed via an MQTT publish/subscribe server.

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

This manual assumes familiarity with logic programming and with higher functional programming in a typed language. A tutorial introduction to the QuLog declarative subset is given in,

`doc/tutorial/QuLog.pdf`.

A formal systax using extended BNF grammar rules is given as an Appendix of this manual. `examples/introduction/qlexamples.qlg` is an example QuLog program that can be consulted and queried.

The `teleor` extension of the QuLog interpreter allows program files to be consulted containing TeleoR procedures as well as QuLog rules. This exten-

sion includes a generic agent shell that can be launched to execute calls to TeleoR procedures as tasks. It can be configured by including specially named QuLog action procedures and relations in your program file, as explained in `doc/tutorial/toolsRM.pdf`.

To support use of TeleoR robotic agent programs with robots and simulations that use ROS, there is an example Python program in the ROS directory that will act as an interface between our Pedro inter-agent and inter-process communications server and an invocation of ROS. This program and information on how to modify it for a particular ROS architecture, are in the `examples/ROS` directory of the QuLog distribution.

QuLog has some predefined constructor types. These usually have a name such as `write_type__` and their constructors and atom values often end with underscore, for example the constructor `q_` and the atom `nl_` of the `write_type__` system type. We recommend you do not use a trailing underscore in any of your own type definitions. If you do use a system reserved name the compiler will reject your definition giving an error message.

Every data type of QuLog has a place in a lattice of types. At the bottom of the lattice is the system type `bottom` with one data value `botton_`, meaning *undefined*. At the top is the system type `top`. There are no values that just belong to `top` but it includes all data and code types. Just below `top` on the data side of the lattice is the type `term`, which will be familiar to Prolog programmers. All other data types are sub-types of `term`.

The type `code` is the other immediate sub-type of top. All relation, function, action and TeleoR procedure types are sub-types of `code`.

For a particular program the lattice of system and program associated types is finite.

1 Getting Started

This section describes how to set up the required environment variables and briefly describes how to run the interpreter. At the moment it is not possible to generate an executable QuLog application that can be launched independently of the interpreter. This will be possible. However, the intepreter can be used to launch a multi-threaded message communicating application that can be left to its own devices.

1.1 Environment Variables

The root directory of the QuLog tree contains the files `PROFILE_CMDS` that can be used to define the required environment variables.

1.2 Data Areas

Because Qulog is currently implemented in QuProlog it has the same data areas as QuProlog and the sizes of these areas can be modified in the same way as for QuProlog - see the QuProlog manual.

1.3 Running the Interpreter

`qulog` is the name of the Qulog interpreter. From a `Unix` shell, Qulog is started by typing:

```
qulog
```

or

```
qulog -A name
```

where `name` is a name for this QuLog process that will be registered with a Pedro server running on the same host. You need to use this option if you want to be able to receive and/or send messages to other processes that have similarly registered a different name with this Pedro server.

If the Pedro server is running on a different host identified by domain or IP address `Host`, launch QuLog using

```
qulog -N Host -A name
```

For example

```
qulog -N leo.itee.uq.edu.au  -A keith_agent
```

When the interpreter is ready it will prompt you with

```
| ??
```

At this point, an expression query, a relation query or an action command, followed by a FULLSTOP NEWLINE, or NEWLINE NEWLINE, can be entered. The interpreter will check that the query or command is syntactically and type correct and that modes of use are correct. It will either display an error message or print out a response to the query or command.

A CONTROL-D will exit the interpreter whenever you get the prompt.

CONTROL-C will interrupt an evaluation and allow you to abort the interpreter (enter e in response to the interrupt prompt), or to terminate the current query and any forked action threads (enter r in response to the interrupt prompt), giving you the `| ??` query prompt again. There are other response options, displayed if you enter ? in response to the interrupt prompt.

If the query is an expression then the result of the expression evaluation will be displayed followed by its minimal type.

Example:

```
| ?? 2+sin(pi_()/4).

2.70711 : num

| ?? cos.

cos : (num)->num
```

The second expression is just the name of a primitive function and the value is that function. However its type is usefully displayed.

If you enter a relation query then either 'no' will be displayed to indicate there are no solutions to the query or bindings for variables of the query with their minimal types will be displayed separated by lines of fullstops. If you entered a command any output from the command will be displayed followed by 'success' or 'fail' depending upon whether the command ultimately succeeded or failed.

When there are multiple solutions to a relation query the first five (if there are that many) are displayed separated by lines containing `...`

Example:

```
| ?? X in [4,0,3,4].

X = 4 : nat
...
X = 0 : nat
...
X = 3 : nat
...
X = 4 : nat
| ??  % New prompt indicates all sols have been given
```

If there are five or more solutions the interpreter waits for input from the user before displaying more. The two usual responses are:

NEWLINE - no more solutions are required; or

..NEWLINE - asking for up to 5 more solutions.

Example, showing a second use of 'in':

```
| ?? S in "Apple".

S = "A" : string
```

```
...
S = "p" : string
...
S = "p" : string
...
S = "l" : string
...
S = "e" : string
..      % Request for more answers if there are any
no more solutions
% Above displayed only after .. input and there are no more answers

| ?? X in {6,2,3,0,3,7,4}.
% {6,2,3,0,3,7,4} is a set so second 3 ignored, third use of 'in'

X = 0 : nat    % Answers displayed in value order
...
X = 2 : nat
...
X = 3 : nat
...
X = 4 : nat
...
X = 6 : nat
..        % Request to display up to 5 more answers
X = 7 : nat
| ??      % Prompt for next query indicating no more answers
```

If you feel that the interpreter is giving back too many, or too few answers for a particular problem you can control this in two ways. The first is to prefix the query by the number of solutions you would like displayed at a time, followed by a `?`, followed by the query. Also, instead of simply using a `..` to ask for more solutions you can change the number of solutions to be displayed for this query to positive integer `k` by entering `..k`.

Example:

```
| ?? 1 ? X in [1,2,1,4,2].  % Answers 1 at a time

X = 1 : nat
.. 2                % Switch to sols 2 at a time
X = 2 : nat
...
X = 1 : nat
..                  % Request for the next 2 sols
```

```
X = 4 : nat
...
X = 2 : nat
..                  % Request for the next 2 sols
no more solutions
| ??
```

You can also change the default number of solutions that are displayed for any query to a positive number `n`, say 3, using the command:

```
| ?? set_num_answers 3.

success
```

Sometimes you might have a relation query that contains many variables but you might only be interested in the bindings for some of the variables. This can be accomplished by listing the variables for which you want to see the answer bindings, separated from the query by a `?`.

Example:

```
| ?? L1, L2 ? append(L1, L3, [1,2,3,4,5,6]) & append(L4, L2, L3)
     & 2 = #L4.
     % Expressions such as  #L4, length of L4, can be used in = tests

L1 = [] : [Ty1]
     % A type variable Ty1 as [] is empty list of any type
L2 = [3, 4, 5, 6] : [nat]
     % [nat] is type expression for a list of nats (non-neg ints)
...
L1 = [1] : [nat]
L2 = [4, 5, 6] : [nat]
...
L1 = [1, 2] : [nat]
L2 = [5, 6] : [nat]
...
L1 = [1, 2, 3] : [nat]
L2 = [6] : [nat]
...
L1 = [1, 2, 3, 4] : [nat]
L2 = [] : [Ty1]
```

The two ideas above can be combined as in the following example.

```
| ?? 2 L1, L2 ? append(L1, L3, [1,2,3,4,5,6]) & append(L4, L2, L3)
     & 2 = #L4.

L1 = [] : [Ty1]
L2 = [3, 4, 5, 6] : [nat]
...
L1 = [1] : [nat]
L2 = [4, 5, 6] : [nat]
```

Equivalently you can express the above query using an existential quantification on `L3, L3`.

```
| ?? 2 ? exists L3, L4 append(L1, L3, [1,2,3,4,5,6]) &
    append(L4, L2, L3) & 2 = #L4.
```

The existential quantification is needed if you want all the answers as a list.

```
| ?? [(L1,L2) :: exists L3, L4 append(L1, L3, [1,2,3,4,5,6]) &
      append(L4, L2, L3) & 2 = #L4].

[([],[3,4,5,6]), ([1],[4,5,6]), ([1,2],[5,6]), ([1,2,3],[6]),
       ([1,2,3,4],[])]:[([nat],[nat])]
% Value type is a list of pairs of lists of nats
```

We can re-express the last list expression query more succinctly using the list concatenation operator `<>` for splitting of a list using the special non-determinsitic match operator `=?` that requires its left hand side to be, or to evlaluate to a ground term. `<>` may also be used for concatenating complete lists or ground or non-ground terms.

```
| ?? [(L1,L2) :: exists L4 [1,2,3,4,5,6] =? L1 <> L4?2=#L4 <> L2]
```

Using this non-deterministic list pattern matching we do not need the `L3` variable, and the constraint that `L4` must contain two elements becomes a constraint

```
2=#L4
```

expressed inside the `<>` pattern expression attached to the variable `L4`, preceded by a `?`.

If you have constructed a program file `prog1.qlg` of QuLog type definitions, type declarations for relations, function and actions and their rules, you can bring all those into the interpreter using the command

```
| ?? consult prog1.

success
```

You may get syntax and mode errors signalled in which case none of the program file is consulted. There will be at least one QuLog examples file with the QuLog distribution that you installed. You can consult and query one of these files. For example, there may be a file `qlexamples.qlg` in `qulog/examples/introduction`. If you launch QuLog from inside this directory you can load all its definitions using:

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

You can see all the relation and function rules you currently have in the interpreter using:

```
| ?? show.
```

or specific ones using:

```
| ?? show child_of, person, fact, new_child.
```

Notice the variable names of the consulted file will be used.

You can see all the type definitions and declarations using:

```
| ?? types.
```

You can see all the system type definitions and the type declarations for the primitive relations, functions and actions using:

```
| ?? stypes.
```

A displayed type declaration may be accompanied by a brief description of the primitive. You can also show the type declarations for specific relations by giving their names, separated by commas, after either the `types` or `stypes` command.

2 Syntax

This section informally describes the concrete syntax of Qulog. There is a formal extended BNF syntax in the HTML file ¡a href=”grammar.html”¿EBNF Grammar for Qulog¡/a¿ and as an Appendx of this Manual.

The basic building block is an *expression*. An expression is a: *data constant* (aka *atomic* value), *variable*, *compound term*, *list*, *set*, *code name*, *function call*, *list comprehension*, *set comprehension*.

We define each of these categories below.

A reader unfamiliar with logic programming might find it odd that a *variable* is considered a data value. However in both QuLog and Prolog variables are first class values and can be passed between calls and embedded in lists and other compound terms, but not in sets. An answer to a query that contains a variable denotes the set of instantiates of that answer where the variable is replaced by any value of its type. The ability to pass around terms that are or which contain variables is a powerful programming feature of QuLog and Prolog. It is not a feature of Datalog or Answer Set Logic Programming.

The last three are evaluable expressions that denote a *ground term*.

A term is a: *data constant*, *code name*, a *simple compound term* (see below) all arguments of which are terms, a *list of terms*, a *set of ground terms*.

A *ground term* is a term containing no variables. QuLog function call evaluation is strict. A function call argument is completely evaluated just before the function call in which it appears is evaluated.

A *code name* is a value of system type `code`. For example `append` and `*`

Both `term` and `code` are sub-types of system type `top`

A `ground expression` is an expression that contains no variables, or is such that all its variables are bound to ground values at the point that the expression is evaluated.

2.1 Data constants - the atomic term values

These are atoms (`atom` type), natural numbers (`nat` type), integers (`int` type), floating point numbers (`num` type) and strings (`string` type).

QuLog strings are not lists of byte codes as in Prolog. They are packed sequences of byte codes as in Python an are stored on the heap and are garbage collected when no longer referenced. Identity of strings is determined by character by character matching if they have the same length. Manipulation of strings - concatenation and sub-string extraction involves copying but is quite fast.

As in Prolog, QuLog atoms are stored in an atom table and are replaced by

a pointer to its entry in the atom table. Identity of atoms is then identity of atoms table address, there is no character by character matching at runtime. Atoms are a suitable alternative to strings for character sequence values that will not be manipulated and are not transient values. For example use them for names of things in facts. The atom table entries are not garbage collected.

All data contants are sub-types of the system type `atomic`.

`nat` is a sub-type of `int`, which is a sub-type of `num`

2.1.1 Atoms

There are four syntactic forms for atoms.

1. A lower case letter followed by any sequence consisting of "_" and alphanumeric characters.
   For example:

   ```
   percy_smith_2
   semester_1
   ```

2. Any combination of the following set of graphic characters.

   ```
   |-/+*<=>#@$\^&~:.?
   ```

   For example:

   ```
   @<=
   ```

3. An atom of the above two forms preceded by the back quote charater "‘". This form of atom is used when the sequence of characters without the back-quote has been used as the name of a relation, function or action. Such a name cannot be used as an atom unless preceded by a backquote. For example:

   ```
   ‘<>
   ‘append
   ```

4. Any sequence of characters enclosed by "’" (single quote). Single quote can be included in the sequence by writing the quote twice. "\" indicates an escape sequence, where the escape characters are case insensitive. The possible escape characters are:

`newline`	Meaning: Continuation.
`^`	Meaning: Same as `d`.
`^character`	Meaning: Control character.
`dd`	Meaning: A two digit octal number.
`a`	Meaning: Alarm (`ASCII = 7`).
`b`	Meaning: Backspace (`ASCII = 8`).
`c`	Meaning: Continuation.
`d`	Meaning: Delete (`ASCII = 127`).
`e`	Meaning: Escape (`ASCII = 27`).
`f`	Meaning: Formfeed (`ASCII = 12`).
`n`	Meaning: Newline (`ASCII = 10`).
`odd`	Meaning: A two digits octal number.
`r`	Meaning: Return (`ASCII = 13`)
`s`	Meaning: Space (`ASCII = 32`).
`t`	Meaning: Horizontal tab (`ASCII = 9`).
`v`	Meaning: Vertical tab (`ASCII = 11`).
`xdd`	Meaning: A two digit hexadecimal number.

Here are a few examples of quoted atoms.

```
'hi!'
'they''re'
'\n'
```

2.1.2 Numbers

The available range of integers is `-(2^31-1)` to `2^31-1` on a 32 bit machine and `-(2^63-1)` to `2^63-1` on a 64 bit machine. Integers can be represented in any of the following ways.

1. Any sequence of numeric characters. This method denotes the number in decimal, or base 10.
   For example:

   ```
   123
   ```

2. `Base'Number`, where `Base` ranges from 2 to 36 and `Number` can have any sequence of alphanumeric characters. Both upper and lower case alphabetic characters in `Number` are used to represent the appropriate digit when `Base` is greater than 10.
   For example, integer value 10 can be written as:

   ```
   2'1010
   16'A
   16'a
   ```

3. Binary numbers can also be represented in the form `0b` followed by binary digits. Similarly octal and hexadecimal numbers can be represented by `0o`

or `0x` followed by digits.
For example

```
0b1011
0o3170
0x3afd
```

4. 0'`Character` gives the character code of `Character`.
For example,

```
0'A
```

gives the ASCII character code 65.

A natural number is a non-negative integer. `num` type numbers include double precision floating point numbers. They are represented using either a decimal point or scientific e notation. Examples:

```
27.8
1.896e4
```

2.1.3 Strings

Any sequence of characters enclosed by '`"`' is considered as a string.

Note: Strings in Qulog are the same as Python strings and NOT Prolog strings.

Example:

```
| ?? "Hello"++"there".
"Hello there" : string
```

2.2 Code names

Syntactically these are the same as the first two forms of atom - alphanumeric and graphic - but they are the names of primitive or program defined relations, functions or actions.
For example:

```
<>
append
```

These were names we have to precede with a back-quote if we want to use them as atoms. Without a preceding back-quote they denote code values.

2.3 Variables

Variables are available in three syntactic forms.

1. An upper case letter followed by any sequence consisting of "_" and alphanumeric characters.
   For example:

   ```
   List_nums  Head
   ```

2. "_", followed by an upper case letter, and then any sequence consisting of "_" and alphanumeric characters.
   For example:

   ```
   _Dictionary   _X_1
   ```

3. "_" alone denotes an anonymous variable. Repeated occurences of underscore in a query or rule denote different unnamed variables.

Variables beginning with an underscore should be used when there is just a single occurence of a variable in a rule. It suppresses the "single occurrence of variable" warning which is given otherwise, which is useful for picking up mis-typed variable names.

2.4 Compound Terms

A *simple compound term* is composed of an atom of the first two forms (an alphanumeric or graphic atom), called the functor, immediately followed (no spaces) by a sequence of zero or more expressions separated by commas, enclosed in a pair of "(".")" parenthesis. For example:

```
data(jack, 35)
tr(emp(),X/9,tr(L,7,R))
$$(5)
```

Simple compound terms are typically instances of a structured data type declared in the program where the functor is a constructor for the type. If not, the compound term has default type `term`, and a warning that it is not a constructor of a defined type is issued in case there has been a spelling error.

A *compound term* is a simple compound term, or a compound term immediately followed by a sequence of zero or more expressions separated by commas, enclosed in a pair of "(".")" parenthesis. For example:

```
curry(*)(4)
curryR(child_of)(mary)(P)
```

Compound terms that are not simple determine the functor of the compound term by a function call which is itself a compound term.

2.5 Function calls

A *function call* is a either simple compound term where the functor is the name of a primitive or program defined function, or it is a non-simple compound term where the compound term that denotes the functor is a function call that returns a function value.

For certain binary primitive functions the functor name may be used as an infix operator and placed between the two arguments. This holds for the usual binary arithmetic operators `+`, `*` etc. for which function applications are written as expressions such as `6+9*X`.

The special zero argument functions `e_` and `pi_`, invoked as in expressions `e_()` and `pi_()`, evaluate to the numbers 'e' and 'pi'. More details are given in Section 3.7

In the QuLog interpreter a function call, indeed any expression, can be given as an entry to be evaluated.

Examples:

```
| ?? 67.7/2.3.
29.4348 : num

| ?? curryR(child_of)(peter).

curryR(child_of)(peter) : (atom)<=
% The denoted value is a relation over atom names
```

Function calls denote expressions that contain no function calls. That is they denote non-variable terms: atomic values, code names, simple compound terms all the arguments of which are non-variable terms, lists or sets of non-variable terms. The exceptions are certain code returning function calls which are only evaluated when the code value they denote is itself called. The above `curryR(child_of)(peter)` is an example. It denotes an unary relation but that relation is only used when the unary relation is called in a query such as:

```
| ?? P ? Rel=curryR(child_of)(peter) & Rel(P).

P = harry : atom
P = mary : atom
```

2.6 Lists

A list is a comma separated sequence of expressions enclosed in "[".."]" brackets. This is a complete list. Or it is a comma separated sequence of terms ending with `,..` optionally followed by a variable, or ending with `|`) always followed by a variable, enclosed in "[".."]" brackets.

Example:

```
[3,2.7,X*Y,"hello"]
[3,4,..Tail]
[Head,..Tail]
[Head,..]    % shorthand for [Head,.. _]  with _ the anonymous variable
[Head|Tail]
```

The first example is a complete enumeration and the remaining examples are list patterns in which both `,..` and `|` can be read as "followed by". The fourth example is equivalent to `[Head,.. _]`. The last example is using the Prolog syntax for a list.

2.6.1 List compehension expressions

Lists of ground terms can also be denoted by a list compehension expression. Examples are:

```
[X**2::X in L & not X in [0,1]]
% Squares of numbers other than 0,1 in nums list L

[C :: exists A child_of(C,peter) & age_of(C,A) & A<18]
% Non-adult children of peter in order found
```

The general form of a list comprehension is:

```
[Expression :: exists VarSequence SimpleConjunction]
```

where the `exists VarSequence` is optional.

There are constraints on the variables that can be used in such a comprehension. Each variable in `Expression` must either appear in `SimpleConjunction` and be such that it will be given a ground value by some call in the conjunction, or it must appear before the comprehension expression in a query or rule and will have been given a ground value. Every variable in `SimpleConjunction` must either be underscore, appear in `Expression` or in `VarSequence`, or must appear before the comprehension expression in a query or rule and will have been given a ground value. This ensures that the value of a list comprehension is always a list of ground terms.

`VarSequence` is a single variable or a comma separated sequence of variables such as `X,Y,Z`

The syntax for `SimpleConjunction` is given below.

2.7 Sets

A set is a comma separated sequence of ground expressions surrounded with `{` and `}` braces. If there are any duplcate ground terms when all the expressions have been evaluated all but one of the duplicates will be removed. If returned as the value of a expression entry to the interpreter, or as a binding of a variable in a relation query, it will be diplayed with its elements in term order as determined by the `@` primitive.

Example set expression entry:

```
| ?? {4,3,1,5-2,1}.
{1, 3, 4} : {nat}
```

2.7.1 Set compehension expressions

Sets can also be denoted by a set compehension expression. Examples are:

```
{X**2::X in L & not X in [0,1]
% Squares of numbers other than 0,1 in  L, duplicates removed.
{A :: exists C, P child_of(C,P) & age_of(C,A) & A<18}
% Set of all the ages of recorded children
```

The general form of a set comprehension is:

```
{Expression :: exists VarSequence SimpleConjunction}
```

where the `exists VarSequence` is optional.

The constraints on the variables that can be used in a set comprehension are the same as those for a list comprehnesion expression.

`VarSequence` and `SimpleConjunction` are as for list comprehensions.

2.8 Programs

A Qulog program comprises a sequence of:

type definitions,

type declarations,
relation rules (aka clauses),
action rules,
function rules.

A QuLog/TeleoR program also includes:

TeleoR procedures.

They may appear in any order except that all the rules for a particular relation, action or function must be contiguous. A type declaration for a relation, action, function or procedure does not need to be given immediately before its code. The rules of a TeleoR procedure are all included inside `{...}` braces following the procedure head.

An important constraint is that each type definition, type declaration, relation, action and function rule *must* begin at the left end of a new line. If one needs to be continued over more than one line all but the first line must be indented from the left end by at least one space or tab. Fullstop terminators *may* be given at the end of each definition, declaration or rule but is not needed. It is the text starting at the extreme left end of a line after one or more newlines that terminates the previous program statement. TeleoR procedures must also begin at the left end of a line but inside the `{...}` there are more relaxed layout constraints. Each TeleoR rule can start anywhere on a new line. The parser can use the rule syntax to determine that it is the start of a new rule. Again fullstop terminators may be given at the end of each rule but are ignored.

2.8.1 Type Definitions

A type definition is of the form

type-name `::=` *type-expression*

type-name is either an alphanumeric atom or a single argument compound term whose only argument is a variable (representing any type). A type definition with such a type name defines a parameterised type where the type variable stands for any type. That type variable then appears in one or more of a disjunction of compound terms with other arguments that are type names. We give examples below.

type-expression is another user or system defined type, in which case the type definition is essentially a type alias, for example

```
speed ::= num
```

More usually it is one of the following type expressions defining a new data type.

Integer range type expression

This is an expression of the form `M..N` where `M < N` and both are integers.

Examples:

```
digit::= 0..9
small_int::= -10..10
```

As in the examples different range types may overlap but only when one is completely contained inside the other. To have overlapping sets of integers corresponding to different types, type union must be used (see below).

Disjunction of constants type expression

This is an expression of the form `C1 | C2 | ... | Ck` where each `Ci` is the same kind of constant, except that we can mix different types of numbers.

Examples:

```
gender::= male | female
threeNums::= 20 | 6.7 | -50
article::= "a" | "an" | "one" | "the" | "that" | "those"
```

Different type definitions using overlapping disjunctions of constants are allowed providing one is completely contained inside the other. So, as well as the `article` type we could define

```
indef_article::= "a" | "an" | "one"
```

A disjunction of integers can also overlap with a range type providing it either comprises a subset or a superset of the integers of the range type. These constraints ensure that each constant belongs to a unique minimal type. For example `"a"` would belong to the types `indef_artcle`, `article`, `string`, `atomic`, `term`, `top`.

To have partially overlapping disjunctions of constants corresponding to different types, type union expressions must be used to define each partially overlapping type (see below).

Parameterised type expression

This is an expression of the form `CT1 | CT2 | ... | CTk` where each `CTi` is a compound term with arguments that are type names, or a single type variable `T`, or a parameterised type name with argument the same type variable `T`. Such a type expression can only appear as the right hand side of a parameterised type definition with left hand side a unary compound term containing the type variable `T`.

Examples:

```
tree(T) ::= empty() | tr(tree(T),T,tree(T))
an_indexed(T)::= rec(int,T)
```

Type union expression

This is an expression of the form `Ty1 || Ty2 || ... || Tyk` where each `Tyi` a simple type name or a ground parameterised type name or a code type expression.

Examples:

```
int_atom ::= int || atom
int2intOrstring::= int -> int || int -> string
```

Code type expressions

The last example above was the union of two function type expressions. There are four code type expressions in `Qulog/TeleoR`. These are: a function type, a relation type, and action type and a `TeleoR` procedure type.

Function type expression

This has the form `(TE1,TE2,...,TEk) -> TE` where each `TEi` and `TE` is any simple, or compound type name, or type union expression, or a code type expression.

Relation type expression

This has the form `(MTE1,MTE2,...,MTEk) <=` where each `MTEi` is a moded type where the type is any simple, or compound type name, or type union expression, or a code type expression.

The possible modes of a moded type are the prefixes `!`, `?` and `??` and the postfix `?`.

The moded type `!`*Type* used as an argument of a relation means, when called, the supplied argument must be ground and of type *Type*.

The moded type `?`*Type* used as an argument of a relation means, when called, the supplied argument must either be ground and of type *Type* or will be ground to a term of type *Type* by the call.

The moded type *Type*`?` used as an argument of a relation means, when called, the supplied argument must either be ground and of type *Type* or, if ground by the call, will be ground to a term of type *Type*.

The moded type `??`*Type* used as an argument of a relation means, when called, the supplied argument, if ground, must be of type *Type* and the call will not further instantiate the argument.

Modes can be used multiple times in structured types as long as inner modes are more liberal than outer modes. For example, the moded type
`![?int]`

means that the top-level list structure must be given (i.e. the number or elements are known at call time) but the elements of the list can be a mixture of integers and variables with the variables instantiated to integers by the call.
Action type expression

This has the form `(MTE1,MTE2,...,MTEk)` ∼`>>` where each `MTEi` is a moded type where the type is any simple, or compound type name, or type union expression, or a code type expression.

The modes are as described above. **TeleoR type expression** This has the form `(TE1,TE2,...,TEk)` ∼`>` where each `TEi` is any simple, or compound type name, or type union expression, or a code type expression.

2.8.2 Type Declarations

All functions, relations, actions and teleoR programs have type declarations of the form

Name : *Type*

Examples of declarations are given below where definitions are described.

If multiple functions, relations, actions or teleoR programs have the same type their names can all be listed on the left hand side of a declaration as follows.

Name1, ..., NameN : *Type*

As an example, below is the type declaration for the builtin `append` relation (with the same semantics as the standard Prolog append relation).

```
append : (![T], ![T], ?[T]) <=  |
         (?[T], ?[T], ![T]) <=  |
         (![T?], ![T?], ?[T?]) <=  |
         (?[T?], ?[T?], ![T?]) <=  |
         ([T]?, [T]?, [T]?) <=
```

The first type of `append` says that, if the first two arguments of the call on `append` are ground lists of a given type, then the third argument will be a ground list of the same type on exit from the call.

The second type says that, if the third argument is a ground list of a given type, then the first and second arguments will be ground lists of the same type on exit from the call.

The third type of `append` says that, if the first two arguments are lists of a known length (i.e. do not have a variable tail) but possibly containing non-ground elements, then the third argument will have a known length on exit from the call but that variables occurring in any of the arguments need not be ground.

The fourth type is the "append driven backwards" version of the third type.

The fourth type is the most general allowing variable length lists in all arguments. In this situation, nothing can be said about the modes on exit from the call.

Note that when we say, for example, the first two arguments are of the same type we mean that the type inference system can find a suitable type as in the example interpreter query below.

```
| ?? append([1,2], [a,b], X).
X = [1, 2, a, b] : [atom || nat]
```

Here, the suitable (minimal) type for `T` is the union of two types.

Beliefs are dymanic relations and are declared as follows.

`belief` *Name1* : *ArgTypes1* `,` `...,` *NameN* : *ArgTypesN*

Where each *ArgTypesI* is of the form

`(TE1,TE2,...,TEk)` in which each `TEi` is any simple, or compound type name, or type union expression, or a code type expression.

The declaration
`belief age_of: (human,age)`
is esentially the declaration
`age_of : (?human, ?age) <=`
together with the declaration that `age_of` is dynamic.

Percepts are similar to beliefs in that they are dynamic but are specifically used for storing percepts in teleoR programs. They are declared as follows.

`percept` *Name1* : *ArgTypes1* `,` `...,` *NameN* : *ArgTypesN*

Global variables are used to store either integer or number values and are declared as follows.

`int` *Name* `:=` *IntValue*
or
`num` *Name* `:=` *NumValue*

The declaration
`int count := 0`
is like a combination of the declaration
`belief count : (int)`
and the definition `count(0)`
with the restriction that the `count` belief always contains exactly one fact.

2.8.3 Simple Conditions for Rules and Relation Queries

These comprise *predications*, *negated predications* and *meta-calls*.

Predications

These are atoms that are names of no argument defined relations or compound terms with functors that are the names of primitive or program defined relations with argument types consistent with the relation's declared type. A compound term with a functor that is an expression of relation type consistent with the argument types of the compound term is also a predication, this includes the case where the functor is a variable. The compiler will also check that all varibles of any argument term of the predication that must be ground will have have been given ground values by the time the predication needs to be evaluated.

Negated Predications

These have the form `not Cond` where `Cond` is a predication. The compiler will check that any arguments of the predication `Cond` that need to be ground will have ground bindings for all their variables before the `not` is evaluated. All other arguments must also be ground values by the time the condition is evaluated, or they must be anonymous underscore variables.

`not Cond` is deemed to have been inferred if, and only if, a complete search of the tree of alternatve possible inferences of `Cond` fails to find a proof. It is the so-called *negation-as-failure*. It is a sound rule of inference on certain assumptions regarding the completeness of the relation definitions used in the exploration of the possible proofs of `Cond` and on the assumption that different data terms (after being normalised in the case os sets) denote different values.

Meta-calls

These have the form `call Var` where `Var` is of type `relcall`. `relcall` is the system type comprising all terms that denote type correct calls to primitive or program defined relations. The meta call `call Var` succeeds providing the relation call denoted by the `relcall` value of `Var` at the time of evaluation has all its input arguments ground and will succeed.

Complex Conditions

`not Predication`

> Negation. If `Predication` is inferable then fail else succeed. At the time of call all variables appearing in `Goal` must be ground or underscore variables.
> mode/type `not : (??relcall) <=`
>
> `relcall` is the system type comprising all terms that are type correct calls to primitive or program defined relations. The functor of the predication must either be given or be a variable with known moded type so that the compiler can check that the call is type correct and that any arguments that need to be ground will have ground bindings before the

`not` is evaluated. The only variables that need not already have ground bindings are underscore variables. None of these can be in `!` argument positions of the called relation.

`once (SimpleConj)`

Find first solution to `SimpleConj` and discard any alternative choices. If `SimpleConj` is just a predication the brackets are not needed.

`once` cannot be used inside a `SimpleConj`. It may only be used in an interpreter query or in the conjunctive body of a relation rule.

`call Predication`

call `Call`
mode/type `call : (relcall?) <=`

`relcall` is the system type comprising all terms that are type correct calls to primitive or program defined relations. The functor of the predication must either be given or be a variable with known moded type so that the compiler can check that the call is type correct and that any arguments that need to be ground will have ground bindings before the `call` is evaluated.

`forall UVars (SimpleConj1 => exists EVars SimpleConj2)`

If, for all bindings of variables in `UVars` that make `SimpleConj1` inferable, `exists EVars SimpleConj2)` is also inferable, then the `forall` succeeds, otherwise it fails.

`UVars` must be a collection of new variables and all variables occurring in `SimpleConj1` must be in `UVars`, have a ground value before the `forall` is evaluated, or be an underscore variable.

`EVars` must also be a collection of new variables and all variables occurring in `SimpleConj2` must be in `UVars` or `EVars`, have a ground value before the `forall` is evaluated, or be an underscore variable. `exists EVars` is optional.

`forall` cannot be used inside a `SimpleConj`. It may only be used in an interpreter query or in the conjunctive body of a relation rule.

Assuming we have a collection of `child_of(C, P)` beliefs that associate a child `C` with a parent `P`, and a collection of `person(Name, Gender, Age)` beliefs, giving the gender and age of people. The following query will return as an answer each parent who has at least one adult child.

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

Note that `P` will be given a value before the call on `forall` and so there are no free variables in the body of `forall` other than `C` and the underscore variable.

Also note that each quantified variable is not allowed to appear outside the quantification or in other quantifications.

2.8.4 Relation Definitions

A relation definition consists of a sequence of rules (clauses) of the form

Head `::` *Commit* `<=` *Body*

where *Head* is an atom or simple compound term, *Commit* is a conjunct of goals, and *Body* is a conjunct of goals. Conjuncts are separated by `&`. Both the `::` *Commit* and `<=` *Body* parts of the rule are optional. The heads of each rule of a relation have the same functor and arity.

When a goal *Goal* with the same functor and arity as *Head* is called, the rules of the relation are tried in order. If *Goal* unifies with the head of the rule then the *Commit* part is called. If that succeeds then this rule is committed to (i.e. no subsequent rules are tried on backtracking) and *Goal* succeeds if and only if *Body* succeeds. If *Body* is not present it is treated as being the goal `true`.

If *Commit* is not present then *Goal* succeeds if *Body* succeeds but, on backtracking, subsequent rules will be tried.

The rule has the same semantics as the Prolog rule

Head `:-` *Commit*`, !,` *Body*

Note, however, that cut (`!`) is not part of Qulog.

As examples, the definitions of the relations `greater` and `sum_list` are given below.

```
greater: (!num, !num, ?num) <=
greater(A, B, C) :: A > B <= C = A
greater(A, B, C) :: B > A <= C = B
sum_list : (![num], ?num) <=
sum_list([], 0)
sum_list([H,..T], N) <= sum_list(T, M) & N = H+M
```

Note that in `N = H+M`, `H+M` is evaluated before unification and that the second rule of `greater` could have been written as

```
greater(A, B, C) <= B > A & C = B
```

2.8.5 Action Definitions

An action definition consists of a sequence of rules of the form

Head `::` *Commit* `~>>` *Body*

where *Head* is an atom or simple compound term, *Commit* is a conjunct of goals, and *Body* is a sequence of goals and actions. The elements of the sequence are separated by `;`. Both the `::` *Commit* and `<=` *Body* parts of the rule are optional. The heads of each rule of an action have the same functor and arity. The semantics of action definitions is the same as for relation definitions. The difference is that at least one of the elements of the *Body* sequence is an action which typically has a side effect such as writing, reading, sending a message or updating the database. As examples, the definitions of the actions `ask_query` and `handle_response` are given below.

```
ask_query: (atom,  term?, [term ?], handle)
ask_query(QId,Ans,QList,Ag) ~>>
    ask(QId,Ans,QList) to Ag; Reply from Ag;
    handle_response(QId,Ag,Reply,Ans)
handle_response: (atom,handle, term?, term?)
handle_response(QId,_,tell(QId,ans(Ans)),Ans) :: true
handle_response(QId,Ag,tell(QId,Reply),_) ~>>
    writeLine(['Agent ',Ag,' responded ',Reply,' to query ',QId]); fail
```

2.8.6 Function Definitions

A function definition consists of a sequence of rules of the form

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

where *Head* is an atom or simple compound term, *Commit* is a conjunct of goals, and *Expression* is a term. The `::` *Commit* part of the rule is optional. The heads of each rule of a relation have the same functor and arity.

When the function is called, the rules are tried and the first rule whose *Head* unifies with the function call and where *Commit* is true then the result returned is the evaluation of *Expression*. Note that rules without an explicit *Commit* have an implicit `true` commit and so no backtracking occurs.

As examples, the definitions of the functions `factorial` and `tree2list` are given below (using the tree type given above).

```
factorial : nat->nat
factorial(0)->1
factorial(N) ::  N1 = N-1 & type(N1,nat) -> N*factorial(N1)
tree2list : tree(T) -> [T]
tree2list(empty()) -> []
tree2list(tr(LT, V, RT)) -> tree2list(LT) <> [V] <> tree2list(RT)
```

Note that `type(N1,nat)` is necessary above in order that the type checker can type check the recursive call on `factorial`.

2.8.7 TR Program Definitions

A TR-program definition has the form

Head `{`
    TR Rule
    ...
    TR Rule
`}`

where *Head* is an atom or simple compound term and each *TR Rule* has one of the forms given below.

The most simple TR rule has the form

Guard `~>` *TR Action*

where *Guard* is a conjunct of goals and *TR Action* is of the form given below.

At the other extreme, the most complete form of a TR rule is

Guard `while` *While* `min` *Duration* `until` *Until* `min` *Duration* `~>` *TR Action*

where *While* and *Until* are conjuncts of goals and *Duration* is a number.

The most simple rule above is a particular form of the full rule where both *While* and *Until* are `false` and both durations are `0`.

Other variations of the general form are:

Guard `min` *Duration* `~>` *TR Action*
which is the same as
Guard `while false min` *Duration* `until false min` *Duration* `~>` *TR Action*

Guard `while_until` *Goal* `min` *Duration* `~>` *TR Action*
which is the same as
Guard `while not {\em Goal} min` *Duration* `until {\em Goal} min` *Duration* `~>` *TR Action*

Guard `while_until` *Goal* `~>` *TR Action*
which is the same as
Guard `while not` *Goal* `min 0 until` *Goal* `min 0 ~>` *TR Action*

Semantically, when a TR program is executed, the guards are checked in order until a guard is found that is true (with a given instantiation of variables). The corresponding *TR Action* is then executed.

While *Guard* (with the same instantiation of variables) is true or *While* is true or the duration of the while part hasn't expired (since the time this rule was chosen) then this rule will continue to be the chosen rule until *Until* becomes true and the until duration has expired.

At that point, the guards will be checked again from the beginning. If no earlier

rule guards are true, the same rule will refire if the guard is true with a different instantiation of variables (and execute the corresponding actions using the new instantiation of variables) or will continue as long as the guard remains true with the same instantiation of variables or *While* is true or the while duration hasn't expired. Otherwise, the guards of the rules below this rule are checked.

Note that executing *TR Action* will typically mean stopping durative actions from previous rule firings and starting new actions (both discreet and durative). As an optimization, instead of stopping a durative action with given arguments and starting the same action with different arguments, the system will generate a 'modify action'.

The forms of *TR Action* for each TR rule are given below.

TR Program

A *TR Action* can be a call on a TR program (possibly a recursive call on the same program). When such a rule is fired this TR program is executed.

Sequence of discreet and durative actions

A *TR Action* can be a comma separated sequence of discreet and durative actions. The special sequence `()` is the 'do nothing' action.

Timed sequence

A *TR Action* can be a comma separated sequence of the form
Action `for` *Duration* `,` ... `,` *Action* `for` *Duration*.
where the last duration can be elided.

Each action above can be either a call on a TR program or a sequence of discreet or durative actions.

When called, the first action is called and then after that duration is up, the second action is called and so on until the last action in the sequence is called. After its duration has expired then the sequence is repeated from the start. This will repeat until a different rule is fired. If the last duration is missing, it as treated as infinity.

Wait repeat

A *TR Action* can be of the form
Action Sequence `wait` *Duration* `^` *Repeats*

When called, the actions will be executed, and after *Duration* seconds the actions will be executed again. This will be repeated *Repeats* times unless another rule is chosen. If another rule is not chosen then an error will be generated.

Attached Qulog actions

TR Actions can have Qulog actions attached as below.
TRAction `++` *Action*

When called, both *TRAction* and *Action* will be executed. The Qulog action is

typically a modification to the belief store or a message send action.

Example TR program

As an example of TR Programs, consider the following TR program (from the `examples/introduction` directory of the release) controlling a robot whose objective is to find, approach, and pick up an object using grippers.

```
%% We assume that if the program receives a dead_centre percept
%% it will also receive a centre percept
dir::= left | right | centre | dead_centre
percept
    see : (num, dir),
    holding : ()

durative
    move : (num),
    turn : (dir)

discrete
    grab : (),
    release : ()

%% We interpret holding true and see(0, centre) not true to mean that
%% the grippers are closed but not actually holding an object

get_object : () ~>
get_object {
    holding & see(0, centre)       ~> ()
    not holding  & see(0, centre)  ~> grab wait 10^2
    not holding                    ~> get_to
    true                           ~> release wait 10^2
}

get_to : () ~>
get_to {
    see(0, centre)    ~> ()
    see(0, Dir)       ~> turn(Dir)
    see(_, centre)    ~> move(6)
    see(_, Dir) while see(_, centre) until see(_, dead_centre)
            ~> move(4) , turn(Dir)
    true              ~> turn(left) for 10 ; move(4) for 10
}
```

Consider an initial state where no objects are seen and holding is false. When `get_object` is executed then the third rule is fired causing `get_to` to be executed. The last of rules of `get_to` will be chosen (a timed interval). This will

first cause the robot to start turning for 10 seconds and then start moving for 10 seconds. This will be repeated until an object is spotted.

At some point, say `see(10, left)` becomes true. This causes the fourth rule of `get_to` to fire (with `Dir` instantiated to `10`). Assuming this object is not moved by the environment, then eventually, say `see(8, centre)` becomes true. It might seem that the third rule should now fire because its guard becomes true. However, the until condition prevents higher rules from firing. Once, say, `see(8, dead_centre)` becomes true then rule three will fire. By over-achieving the guard of the third rule the "fluttering" between the third and fourth rule (without the until condition) is eliminated.

The while condition is necessary because it takes over from the guard and mantains rule four as the active rule when seeing to the left is no longer true but seeing to the centre becomes true.

On the other hand, if, before `see(8, dead_centre)` becomes true the environment moves the object so that `see(8, right)` becomes true then there would be a refiring of rule four and the robot will start turning to the right.

Note that, if before the object is seen dead centre, `see(0, centre)` becomes true then rule two of `get_object` will fire. The until only has a local effect - affecting rule choices within its own TR program.

Eventually, without interference from the environment, `see(0, centre)` will become true. The second rule of `get_object` will now fire (stopping the execution of `get_to`), causing the robot to grip the object. Under normal circumstances holding will become true and then the first rule will fire causing the robot to stop.

If, however, there was a mechnical problem with the gripper, the robot will wait for 10 seconds and try to close the gripper again in the hope that the problem will disappear. If the problem doesn't clear up after two attemps (with a 10 second wait for each) then an error will be produced.

It may seem that the robot's job is done now that it has achieved its goal. However, the TR program is still monitoring the state and say the environment now removes the object from the robot's grip. Rule four will fire, opening the grippers, and then, once holding is no longer true, rule three will fire and the robot will go back to searching for an object.

3 Built-Ins

3.1 Introduction

This section contains descriptions of the functions, relations and actions of the Qulog library. In the interpreter you can see all their names and types by entering the command

```
| ?? stypes.
```

Many of these are Qu-Prolog primitive relations 'lifted' to QuLog by giving them appropriate type/mode declarations. Other Qu-Prolog relations and actions can be 'lifted' to the Qulog level be giving them a QuLog type declaration in your QuLog program file.

For example, if the primitives described on Section Section 3.6 had not already been made available for use in QuLog, all you would have needed to do was include their type declarations as given in that section.

As another example, there is a Qu-Prolog primitive

```
between(From,To,N)
```

for generating or testing an integer value `N` between given integer values `From` and `To`.

To use this in a QuLog program, add the moded type declaration

```
between: (integer,integer,?integer) <=
```

to your program file. It tells the QuLog mode/type checker that the relation is a Qu-Prolog primitive (that will be checked), and that in every use the first two arguments should be given as integers but the third integer argument may be given or may be returned as value of an unbound variable.

3.2 Input / Output

3.2.1 Term Input/Output Actions

Actions:

```
writeL(List)
```

> Write the elements of `List`. If the atom `nl_` appears in `List` it is written as a newline.
>
> You can also use `sp_(N)` where N is a positive integer to insert N spaces. Strings in `List` are displayed without the string quotes ".." unless you write them with `q_("...")`. The quotes are then put around the string.
>
> ```
> mode/type   writeL(![??term])
> ```
>
> Example:
>
> ```
> | ?? writeL(["List of atoms ",[a,b], nl_,
>           "Set of nats ", {2,1,4,1}]).
> List of atoms [a, b]
> Set of nats {1, 2, 4}
> ```

with the next output on the next line. `nl_` causes a new line to be output as would the string `"\n"`. In fact any of the C string control characters, such as `"\t"`, `"\s"` for tab and space respectively, can be put into a string and will have the intended effect unless the string is wrapped inside a `q_` term. So we could have written the above query as:

```
| ?? writeL(["List of atoms ",[a,b],
          "\nSet of nats ", {2,1,4,1}]).
```

Other control term we can put in the list argument of `writeL` are:
`sp_(n)`, n positive integer. It will display n spaces.
`uq_(Atom)`, where `Atom` is an atom that normally needs to be quoted. It will be displayed without the single quotes.
`wr_(Var)`, will not display `Var` as an underscore followed by a sequence of digits, as is normal, but will give it a name such as `A`, `B`, `C` when displayed and will give subsequent occurence of `Var` in the list to be output using the given name for `Var`.

The following query illustrates the the use of `uq_` and `wr_`.

```
| ?? writeL([uq_('Hello')," there\n",wr_(_895),sp_(2),
          _895,sp_(2),wr_(_678),nl_]).

Hello there
A  A  B
_895 = A : Ty1
_678 = B : Ty2
success
```

`writeLine(List)`

The same as `writeL` but with a trailing newline.

Example:

```
| ??writeLine([s("A list "), [a, b], sp_(2), {2, 1, 4, 1},
            s(" followed by a set.")]).
A list [a, b]  {1, 2, 4} followed by a set.
```

with the next output being at the beginning of the next line.

`readT(Term)`

Unifies `Term` with the next term denoted by the next sequence of characters typed at the terminal followed by fullstop, return.

Example:

```
| ?? readT(X).
f(A).
X = f(A) : term
```

Note that this read remembers the names of variables (the `A` above). A consequence of this, given the occurs check in unification, is that the following query fails.

```
| ?? readT(A).
f(A).
no
```

3.3 Terms

3.4 Comparison of Terms

Two terms are compared according to the standard ordering, which is defined below. Items listed at the beginning come before the items listed at the end. For example, numbers are less than atoms in the standard ordering.

1. Variables, in age ordering (older variables come before younger variables).
2. Numbers, in numerical ordering.
3. Atoms, in character code (ASCII) ordering.
4. String, in standard string odering.
5. Compound terms are compared in the following order:
   (a) Arity, in numerical ordering.
   (b) Functor, in standard ordering.
   (c) Arguments, in standard ordering, from left to right.
6. Sets, in dictionary order on elements.
7. Lists, in dictionary order on elements.

The above ordering is used when constructing sets.

The following relations use the above ordering to test terms.

`Term1 @> Term2`

Succeeds if `Term1` is greater than`Term1` in the above ordering.

`Term1 @>= Term2`

Succeeds if `Term1` is greater than or equal to `Term2` in the above ordering.

`Term1 @< Term2`

Succeeds if `Term1` is less than`Term1` in the above ordering.

`Term1 @=< Term2`

Succeeds if `Term1` is less than or equal to `Term2` in the above ordering.

3.5 Testing of Terms

These testing predicates are used to determine various properties of the data objects, or apply constraints to the data objects.

Relations:

`type(Term, Type)`

Succeed if `Term` is a non-variable of type `Type`.
mode/type `type : (??top, !typeE(_)) <=`

Example:

```
| ?? type(a, atomic).
yes
| ?? type(a, int).
no
| ?? type([a,2], [int || atom]).
yes
```

`ground(Term)`

Succeed if `Term` is ground.
mode/type `ground : (??top) <=`

`isa(Term, Type)`

Succeed if `Term` is of type `Type` and `Type` is a finite type.
mode/type `isa : (?Term, !typeE(_)) <=`

Example: For this example we assume the following type declarations.

```
name ::= "Alice" | "Bob" | "Carol"
status ::= good(name) | bad(name)

| ?? isa(X, status).
X = good("Alice")
...
X = good("Bob")
...
X = good("Carol")
```

```
...
X = bad("Alice")
...
X = bad("Bob")
..
X = bad("Carol")
| ?? isa(good("Bob"), status).
yes
| ?? isa(2, nat).
no
```

`template(Term)`

Succeed if `Term` is atomic or a compound term with a ground functor.
mode/type `template :(??top) <=`

`ground_inputs(Term)`

Succeed if `Term` is a relation or action term and that the modes of its arguments are correct.
mode/type `ground_inputs : ??(relcall || actcall)`

3.6 List Processing

`append(L1, L2, L3)`

Succeed if `L3` is the concatentation of `L1` and `L2`
mode/type
`append: ([T]?, [T]?, [T]?)<= | (![T], ![T], ?[T])<= | (?[T], ?[T], ![T])<=`

`reverse(L1, L2)`

Succeed if `L2` is the reverse of `L1`.
mode/type
`reverse : (![T?], ?[T?]) <=`

`sort(L1, L2)`

Succeed if `L2` is `L1` sorted.
mode/type
`sort : (![T?], ?[T?]) <=`

`member(X, L)`

Succeed if `X` is in `L`.
mode/type
`member : (T?, [T]?) <=`

`X in L`

Succeed if `X` is in `L`.
mode/type
`in: (?T,[T])<= | (T?,![T?])<= | (?string,[string])<= | (?T,{T}) <=`

Almost exactly the same uses as `member` except that it it must be given a complete list of possibly non-ground terms.

As its type indicates, `in` can also be used to access single character substrings of a string and ground elements of a set.

3.7 Arithmetic

The following arithemetic functions are available.

`Num1 + Num2`

Returns the sum of `Num1` and `Num2`.
`+ : (nat, nat) -> nat | (int, int) -> int | (num, num) -> num`

`Num1 - Num2`

Returns the difference of `Num1` and `Num2`.
`- : (int, int) -> int | (num, num) -> num`

`-Num1`

Returns the negation of `Num`.
`- : int -> int | num -> num`

`Num1 * Num2`

Returns the product of `Num1` and `Num2`.
`* : (nat, nat) -> nat | (int, int) -> int | (num, num) -> num`

`Num1 // Num2`

Returns the integer division of `Num1` and `Num2`.
`// : (nat, nat) -> nat | (int, int) -> int`

`Num1 / Num2`

Returns the division of `Num1` and `Num2`.
`/ : (num, num) -> num`

`Num1 mod Num2`

Returns the mod of `Num1` and `Num2`.
`mod : (nat, int) -> nat | (int, int) -> int`

`Num1 ** Num2`

Returns `Num1` raised to the power `Num2`.

```
** : (nat, nat) -> nat | (int, nat) -> int | (num, num) -> num
```

`Int >> Nat`

Returns the bitwise right shift of Int with respect to Nat.

```
>>: (int, nat) -> int
```

`Int << Nat`

Returns the bitwise left shift of Int with respect to Nat.

```
<<: (int, nat) -> int
```

`Int1 /\ Int2`

Returns the bitwise AND of Int1 and Int2.

```
/\: (int, int) -> int
```

`Int1 \/ Int2`

Returns the bitwise OR of Int1 and Int2.

```
\/: (int, int) -> int
```

`\ Int`

Returns the bitwise complement of Int.

```
\: (int) -> int
```

`abs(Num)`

Returns the absolute value of Num.

```
abs: int -> nat | num -> num
```

`sqrt(Num)`

Returns the square root of Num.

```
sqrt: num -> num
```

`round(Num)`

Returns the round of Num.

```
round: num -> int
```

`floor(Num)`

Returns the floor of Num.

```
floor: num -> int
```

`ceiling(Num)`

Returns the ceiling of Num.

```
ceiling: num -> int
```

`pi_()`

Returns PI.
```
pi_: () -> num
```

`e_()`

Returns E.
```
e_: () -> num
```

`sin(Num)`

Returns the sin of Num.
```
sin: num -> num
```

`cos(Num)`

Returns the cos of Num.
```
cos: num -> num
```

`tan(Num)`

Returns the tan of Num.
```
tan: num -> num
```

`asin(Num)`

Returns the arcsin of Num.
```
asin: num -> num
```

`acos(Num)`

Returns the arccos of Num.
```
acos: num -> num
```

`atan(Num)`

Returns the arctan of Num.
```
atan: num -> num
```

`atan2(Y, X)`

Returns the atan2 of Y and X. This returns an angle in the range `(-pi, pi]`.
```
atan2: (num,num) -> num
```

3.8 Other Functions

`now()`

Returns the current time.
```
now: () -> num
```

`exec_time()`

Returns the lapsed time in seconds since this qulog process was started
```
exec_time: () -> num
```

`start_time()`

Returns the time at which this qulog process was started
```
start_time: () -> num
```

`random_num()`

Returns a random number in `[0,1)`.
```
random_num: () -> num
```

`random_int(Lower, Upper)`

Returns a random number in the interval `[Lower, Upper]`.
```
random_int: (int, int) -> num
```

`S1 union S2`

Returns the union of sets `S1` and `S2`.
```
union: ({T}, {T}) -> {T}
```

`S1 inter S2`

Returns the intersection of sets `S1` and `S2`.
```
inter: ({T}, {T}) -> {T}
```

`S1 diff S2`

Returns the set difference of sets `S1` and `S2`.
```
diff: ({T}, {T}) -> {T}
```

`L1 <> L2`

Returns the concatination of lists `L1` and `L2`.
```
<> : ([T], [T]) -> [T]
```

`S1 ++ S2`

Returns the concatination of strings `S1` and `S2`.
```
++ : (string, string) -> string
```

`#L`

Returns the length of the list, set, or string `L`.
```
# : [T] -> nat | {T} -> nat | string -> nat
```

`F@..Args`

Returns the compound term obtained by applying `F` to `Args`.
```
@.. : (term, [term]) -> term
```

Example:

```
| ?? @..(a, [1,2]).
a(1, 2) : term
```

`@..` can also be used to split up a compound term as in the following example.

```
| ?? a(1,2) =? F@..Args.
F = a : atom
Args = [1, 2] : [nat]
```

`$Name`, where `$` is a prefix operator.

Here `Name` is an atom that must have been initialised with a statement

`int Name:=Integer`, e.g. `int count:=0` or

`num Name:=Number`, e.g. `num savings:=678.50`

in the program. It returns the current value associated with `Name` which can be updated by primitive actions (see `:=` (page 43)).

3.9 Other Relations

`true`

Always succeeds.
mode/type `true : () <=`

`false`

Always fails.
mode/type `false : () <=`

`Term1 = Term2`

Succeeds if `Term1` and `Term2` unify.
Any function calls in each argument term are evaluated first using strict evaluation
- expression arguments evaluated first - as with every relation call.
mode/type `= : (term?, term?) <=`

`N1 > N2`

Succeeds if `N1` is greater than `N2`.
mode/type `> : (!num, !num) <=`

`N1 >= N2`

Succeeds if `N1` is greater than or equal to `N2`.
mode/type `>= : (!num, !num) <=`

`N1 < N2`

Succeeds if `N1` is less than `N2`.
mode/type `< : (!num, !num) <=`

`N1 =< N2`

Succeeds if `N1` is less than or equal to `N2`.
mode/type `=< : (!num, !num) <=`

`X in T`

Succeeds if `X` is a term and and `T` is a list or set of terms and `X` is an element of `T` or if `X` and `T` are both strings and `X` is a single character string occurring in`T`.
mode/type `in: (T?,![T?]) <= | (?T,![T]) <= | (?T,!{T}) <= | (?string, !string) <=`

`string2term(S, T)`

Succeeds if `T` is the term obtained by parsing the string `S`as a Qulog term.
mode/type `string2term : (!string, term?) <=`

`current_thread(Name)`

Succeeds if `Name` is the name of this thread
mode/type `current_thread : (?atom) <=`

`get_active_resources(ResourceInfo)`

Succeeds if `ResourceInfo` is the list of terms `res(atom,[resource])` giving resources used by each running task in a multi-tasking agent. The atom is task name.
mode/type `get_active_resources : (?term) <=`

`get_waiting_resources(ResourceInfo)`

Succeeds if `ResourceInfo` is the list of terms `res(atom,[resource])` giving resources needed by each waiting task in a multi-tasking agent. The atom is task name.
mode/type `get_waiting_resources : (?term) <=`

3.10 Other Actions

`remember(Belief)`

Adds its ground relcall argument (`Belief`) as a new last dynamic fact for its functor relation name `R`. `R` must have been declared as a belief.
mode/type `remember : (relcall) ~>>`

`remember_for(Belief, Secs)`

The same as `remember` except that `Belief` is forgotten after `Secs` seconds.
mode/type `remember_for : (relcall, num) ~>>`

Alternative syntax: `remember Belief for Secs`

`rememberA(Belief)`

Adds its ground relcall argument (`Belief`) as a new first dynamic fact for its functor relation name `R`. `R` must have been declared as a belief.
mode/type `rememberA : (relcall) ~>>`

`rememberA_for(Belief, Secs)`

The same as `rememberA` except that `Belief` is forgotten after `Secs` seconds.
mode/type `remember_for : (relcall, num) ~>>`

Alternative syntax: `rememberA Belief for Secs`

`forget(Belief)`

Remove the first dynamic fact matching `Belief`. Note that `Belief` may contain variables within the arguments. `forget` always succeeds even if there are no matching facts.
mode/type `forget : (relcall) ~>>`

`forget_after(Belief, Secs)`

The same as `forget` except that `Belief` is forgotten when `Secs` seconds has elapsed and `Belief` must be ground at the time of call.
mode/type `forget_after : (relcall, num) ~>>`

Alternative syntax: `forget Belief after Secs`

`replace_by(Belief1, Belief2)`

The same as `forget(Belief1) ; remember(Belief2)`. `Belief1` and `Belief2` may share variables.

`Name := Expression`

Here `Name` is an atom that must have been initialised with a statement

`int Name:=Integer`, e.g. `int count:=0` or

`num Name:=Number`, e.g. `num savings:=678.50`

in the program. These statements are shorthand for `belief` declarations and a definition using one fact of a unary relation called `Name`. They are respectively expanded into:

```
belief Name: int <=
```

```
 Name(Integer)

belief Name: num <=
 Name(Number)
```

The action `Name := Expression` is the same as
`forget(Name(_));remember(Name(Expression))`.

mode/type `:= : (!(?int <= ), !int) ∼>> |`
`(!(?num <= ), !num) ∼>>`

`Name` can be used as though it were a global variable. To access its value the operator `$` is applied. The expression `$Name` evaluates to the current `int` or `num` value stored in `Name`, i.e. in the current `Name` belief.

`Name +:= Expression`

As above, `Name` is an atom that must have been initialised with a statement

`int Name:=Integer` or `num Name:=Number`

in the program.

The action `Name +:= Expression` is the same as
`forget(Name(Val));remember(Name(Val+Expression))`.

mode/type `:= : (!(?int <= ), !int) ∼>> |`
`(!(?num <= ), !num) ∼>>`

Example use `count +:= 1` for increasing value held in `count` by 1.

`Name -:= Expression`

As above, `Name` is an atom that must have been initialised with a statement

`int Name:=Integer` or `num Name:=Number`

in the program.

The action `Name -:= Expression` is the same as
`forget(Name(Val));remember(Name(Val-Expression))`.

mode/type `:= : (!(?int <= ), !int) ∼>> |`
`(!(?num <= ), !num) ∼>>`

Example use `savings -:= 67.90` for decreasing the value held in `savings` by `67.90`.

`fork_as(Action, Name)`

Fork a new Qulog thread, give it the name `Name`, and start the thread executing `Action`. If `Name` is a variable it will be instantiated to a name given by the system. If `Name` is given it must not be the name of an existing thread.
mode/type `fork_as : (actcall, ?atom) ~>>`

Alternative syntax: `fork Action as Name`

`from(Term, Handle)`

This is the message receive action. It will succeed it there is a message term in that threads message buffer whose message term unifies with `Term` and whose message handle unifies with `Handle`. If not the call will suspend and be repeatedly retried as new messages arrive until it succeeds. When it does succeed, the matched message will be removed from the message buffer.
mode/type `from : (term?, ?handle) ~>>`

Alternative syntax: `Term from Handle`

`to(Term, Handle)`

This is the message send action. It sends `Term` as a message to the thread (of possibly another process on another machine) whose message address is `Handle`.
mode/type `to : (??term, !handle) ~>>`

Alternative syntax: `Term to Handle`

`thread_sleep(Secs)`

Causes the executing thread to suspend for `Secs` seconds.
mode/type `thread_sleep : (!num) ~>>`

3.11 TeleoR Specific Actions

The following actions are available when the system is running in TeleoR mode (after the command `teleor` has been executed in the interpreter).

`actions(Actions)`

An interpreter command that can be used in teleor mode when an agent has been started using `start_agent`. `Actions` is a list of actions that agent wants the robotic interface to perform. The given actions must have been declared as discrete or durative.
mode/type `actions : ([discrete || durative]) ~>>`

`kill_agent`

Kill the current agent started using `start_agent`.
mode/type `kill_agent : () ~>>`

`kill_task(Task)`

Kill the task with name `Task` started using `start_task`.
mode/type `kill_task : (atom)` ∼>>

`start_agent(Name, Handle, Convention)`

Start a new agent whose name is `Name`. `Handle` is the message address of the robot interface or simulation with which the agent will interact. `Convention` is the percepts update convention being used. This is one of: `all` if the robot sends all the percepts each time it sends percepts; `updates` if the robot only sends changes to percepts; or `user` if the percept management is application specific in which case the action `handle_percepts_` needs to be defined in the program.
mode/type `start_agent : (atom, handle, atom)` ∼>>

`start_task(Name, TRCall)`

Start a new task (as a thread) whose name is `Name`. `TRCall` is the TeleoR call to be executed in the thread.
mode/type `start_task : (atom, trcall)` ∼>>

4 Standard Operators

We use the Prolog notation for operator declarations even though, unlike Prolog, QuLog's syntax cannot be extended by adding application specific operators and the QuLog parser is not an operator precedence parser. `op` is not a system predicate of QuLog. Using Prolog `op` statements gives us a succint way of summarising the precedence relationship between the operators that is implicit in the formal syntax rules.

In each `op` statement the number is the 'binding' power of the operator, called the *precedece* of the operator. The higher the precedence, the higher up the parse tree, so the less binding the operator. For example, `+` has higher precedence than `*` , so `X+Y*Z` is really `X+(Y*Z)` and we have to use brackets if we want to have the expression `(X+Y)*Z`.

`fx` means the operator is prefix and cannot be followed immediately by an expression with top operator of the same precedence unless that expression is bracketed.

`xfx` means that the operator is an infix non-associative operator and must have expression arguments for which the top operator has lower precedence, or the expressions arguments are bracketed. So, `(X**Y)**Z` needs the brackets.

`xfy` means that the operator is an infix right associative operator, and `yfx` means that the operator is an infix left associative operator. More specically, a `xfy` operator can have an expression to the right with a top operator of equal or lower precedence and that expression being implicitly bracketed. For a `yfx`

this implicit bracketing applies to the expression on the left hand side. So, `X*5 mod 6` is implicitly `(X*5) mod 6`.

Note that `?` has two precedences as an infix operator. One is for its use in an interpreter query after the required initial number of solutions and/or variable bindings have been given. For this use it must have higher precedence than `&`. The second use is in `<>` and `++` patterns when giving a single condition constraint on a sub-string or sub-list. For this use it must have lower precedence than `<>` and `++`.

The mode annotations are not in the table as prefix or infix operators as they will always be inside a bracketed sequence of annotated types.

We have not included `forall` or `exists` as they are both prefix operators taking two arguments, the sequence of variables that immediately follows and then some operator expression. This is an `=>` or `~>>` implication in the case of `forall` and possibly an `&` conjunction in the case of `exists`.

`start_task` and `start_agent` are just reserved words and are never followed by an operator expression.

In QuLog comma is not treated as an operator. It is just an expression separator.

```
op(1100, xfx, [ <=, ~>>, ->, ~>, =>])
op(1050, xfx, [ ::= ])
op(1030, xfx, [ |, || ])
op(1030, xfx, [ .. ])
op(1020, xfx, [ ::, ? ])
<op(1020, xfy, [ while, until ])
op(1000, xfy, [ &, ; ])
op(900,  fx,  [not , once,  watch , watchC, unwatch, show,
      types, stypes, remember, forget, call, do, wait])
op(850, xfx, [ for, after ])
op(800, xfx, [ ? ])
op(700,  xfx, [ = , \= , :=, +:=, -:=, =? , == , \== , @< ,
     @=< , @> , @>= , @.. , in , =:= , =\= , < , =< , > , >= ])
op(600, xfy, [ ++, <> ])
op(550, xfx, [ ? ])
op(500, yfx, [ + , - , /\ , \/, union, diff ])
op(450, yfx, [ to, from ])
op(400, yfx, [ * , / , // , rem , mod , inter, << , >> ])
op(200, xfx, [ ** ])
op(200, fy,  [ + , - , \ ])
op(100, xfx, [ @.. ])
op(50, xfx, [ : ])
op(50, fx, [ $, # ])
```

5 Index

Appendices

A EBNF Grammar for Qulog

```
(* The EBNF grammar for qulog *)

(*

We assume the following non-terminals that group tokens. All other
tokens in the grammar are given as strings.

atom:  the allowed atoms of qulog
string:  double-quoted strings
var: the variables of qulog
int: integers
num: floating point numbers

*)

any_number = int | num;

(* ------ program item ------ *)
(* Note: a program file is parsed one program item at a time *)

program_item =
    type_definition |
    declaration |
    function_rule_definition |
    relation_rule_definition |
    action_rule_definition |
    tr_program_definition;

(* ------ type definitions ------ *)

type_definition = definition_head, "::=", definition_type;

(* either atom type or polymorphic type with one arg *)
definition_head =  atom | ( atom, var ) | ( atom, "(", var, ")" );

definition_type =
      ( int, "..", int ) |
      ( atom, "|", atom, {"|", atom} ) |
      ( string, "|", string, {"|", string} ) |
```

```
      ( any_number, "|", any_number, {"|", any_number} ) |
      ( compound, "|", compound, {"|", compound} ) |
      ( atom_or_compound,  "||",
              atom_or_compound,  {"||", atom_or_compound} ) |
      ( "(", definition_type, ")" ) |
      atom |               (* type macro *)
      compound |           (* type macro *)
      type_expression;

atom_or_compound = atom | compound;
atom_or_simple_compound = atom | simple_compound;

(* ------ types ------ *)

simple_type_expression =
     ( "(", ")" ) |
     var |
     atom |
     compound |
     ( "[", type_expression, "]" ) |
     ( "{", type_expression, "}" ) |
     ( "(", type_expression, ")" ) |
     ( "(", type_expression, ",", type_expression,
             {",", type_expression}, ")" ) (* tuple type_expression *);

type_expression =
     simple_type_expression |
     function_type_expression |
     relation_type_expression |
     action_type_expression |
     tr_type_expression;

type_expression_seq = type_expression, {",", type_expression};

fun_rel_act_tr_type_expression_seq =
    fun_rel_act_tr_type_expression |
    fun_rel_act_tr_type_expression, {"|", fun_rel_act_tr_type_expression};

fun_rel_act_tr_type_expression =
    annotated_type_expression;

declaration_type_expression =
    ( "(", annotated_type_expression,
           {",", annotated_type_expression}, ")" ) |
    annotated_type_expression;
```

```
function_type_expression_seq =
    function_type_expression, ["|", !, function_type_expression_seq];
function_type_expression =
    ( "(", function_type_expression, ")" ) |
    ( domain_type_expression, "->",  type_expression ) |
    ( domain_type_expression, "->",  function_type_expression );

domain_type_expression =
    (basic_inner_annotated_type_expression | annotated_tuple_type_expression) |
     relation_type_expression |
     action_type_expression |
     tr_type_expression;


relation_type_expression_seq =
    relation_type_expression, ["|", !, relation_type_expression_seq];
relation_type_expression =
    (basic_inner_annotated_type_expression | annotated_tuple_type_expression),
         "<=", !;


action_type_expression_seq =
    action_type_expression, {"|", action_type_expression};
action_type_expression =
    (basic_inner_annotated_type_expression | annotated_tuple_type_expression),
          "~>>", !;

tr_type_expression =
    simple_type_expression, "~>", !;

pre_annotation = "!" | "?" | "??";
post_annotation = "?";

annotated_type_expression =
    [pre_annotation], inner_annotated_type_expression, [post_annotation];

inner_annotated_type_expression =
   basic_inner_annotated_type_expression |
   function_type_expression |
   relation_type_expression |
   action_type_expression |
   tr_type_expression;

basic_inner_annotated_type_expression =
    basic_annotated_type_expression |
    annotated_compound_type_expression |
```

```
    annotated_list_type_expression |
    "{", type_expression, "}";

basic_annotated_type_expression =
     atom | var | annotated_tuple_type_expression | ("(", ")");

annotated_tuple_type_expression =
    "(", annotated_type_expression, {",", annotated_type_expression}, ")";

annotated_compound_type_expression =
    ( atom, "(", annotated_type_expression,
       {",", annotated_type_expression}, ")" ) |
    ( atom, annotated_type_expression );


annotated_list_type_expression =
    "[", annotated_type_expression, "]";

(* ------ declarations ------ *)

declaration =
    rule_declaration |
    global_declaration |
    percept_declaration |
    belief_declaration |
    durative_declaration |
    discrete_declaration |
    task_start_declaration |
    task_atomic_declaration |
    resources_declaration;

rule_declaration =
    name_seq, ":",
    (
        fun_rel_act_tr_type_expression_seq |
        function_type_expression_seq |
        relation_type_expression_seq |
        action_type_expression_seq |
        tr_type_expression
     );

name_seq = atom, {",", atom};

global_declaration = ("int" | "num" ), atom, ":=",  arith_term;

percept_declaration =
```

```
    "percept", tr_decl_element, {",", tr_decl_element};

belief_declaration =
    "belief", tr_decl_element, {",", tr_decl_element};

durative_declaration =
    "durative", tr_decl_element, {",", tr_decl_element};

discrete_declaration =
    "discrete", tr_decl_element, {",", tr_decl_element};

tr_decl_element =
    atom, ":", "(", [type_expression_seq], ")";

task_start_declaration =
    "task_start", atom, ":", type_expression_seq;

task_atomic_declaration =
    "task_atomic", atom, ":", type_expression_seq;

resources_declaration =
    "resources", type_expression_seq;

(* ------ function definitions ------ *)
function_rule_definition =
    ( atom_or_compound,"->", term ) |
    ( atom_or_compound, "::", simple_condition_seq, "->", term );

(* ------ relation definitions ------ *)
relation_rule_definition =
    atom_or_compound |
    ( atom_or_compound, "<=", condition_seq ) |
    ( atom_or_compound, "::", simple_condition_seq, "<=", condition_seq ) |
    ( atom_or_compound, "::", simple_condition_seq );

(* ------ action definitions ------ *)
action_rule_definition =
    atom_or_compound |
    ( atom_or_compound, "~>>", action_seq ) |
    ( atom_or_compound, "::", simple_condition_seq, "~>>", action_seq );

(* ------ TR program definitions ------ *)
tr_program_definition =
    atom_or_simple_compound, "{", tr_rules, "}";

tr_rules =  tr_rule, [tr_rules];
```

```
tr_rule  = tr_rule_LHS, "~>", tr_rule_RHS;

tr_rule_LHS =
    simple_condition_seq, [tr_rule_while_until];

tr_rule_while_until =
    ( "min", arith_term ) |
    ( "while_until", simple_condition_seq ) |
    ( "while", while_until_part, "until", while_until_part ) |
    ( "while", while_until_part ) |
    ( "until", while_until_part );

while_until_part =
    simple_condition_seq |
    ( simple_condition_seq, "min", arith_term ) |
    ( "min", arith_term );

tr_rule_RHS =
    ( tr_action, [wait_repeat], ["++", call_term] );

tr_action =
    ( "(", ")" ) | tr_action_seq | tr_timed_seq;

tr_action_seq = ( ( "(", ")" ) | call_term ), [",", tr_action_seq];
tr_timed_seq =
    ( tr_timed_for, ";", call_term ) |
    ( tr_timed_for , ";",  tr_timed_for,
          { ";",  tr_timed_for}, [";", call_term]);

tr_timed_for = tr_timed_seq_action, "for", arith_term;

tr_timed_seq_action = ( "(", ")" ) | call_term | ( "(", tr_action_seq, ")" );

tr_action_seq = call_term, [",",  tr_action_seq];

wait_repeat = "wait", arith_term, ["^", arith_term];

(* ------ terms ------ *)

term =
     ( predication | simple_action |
     arith_term | append_term | concat_term |
     set_term | apply_term | global_val | pedro_addr | compound | basic_term);

basic_term =
```

```
      (
   ("(", term, ")") |
   list_comprehension |
   set_comprehension |
   tuple_constructor |
   list_constructor |
   set_constructor |
   size_term |
   atom | string | var | int | num
    ), !;

arith_term =
    mult_div_term,
    [
("+", arith_term ) |
( "-", arith_term ) |
( "\\/", arith_term ) |
(  "/\\", arith_term )
    ];

mult_div_term =
   (
       exp_term,
       [
  ( "*", mult_div_term ) |
  ( "/", mult_div_term ) |
  ( "//", mult_div_term ) |
  ( "rem", mult_div_term ) |
  ( "div", mult_div_term ) |
  ( "<<", mult_div_term ) |
  ( ">>", mult_div_term )
       ]
   ) |
   ( "(", arith_term, ")" );

exp_term =
    basic_arith_term, [ ( "**", exp_term ) ];


basic_arith_term =
int | num | var | compound | size_term | ("$", atom) |
        ( "(", arith_term, ")" ) |
        ( "+", basic_arith_term ) |
        ( "-", basic_arith_term );

size_term = "#", size_term_body;
```

```
size_term_body = (var | compound | list_constructor | string | set_constructor |
          list_comprehension | set_comprehension ) |
 ("(", append_term, ")") |
 ("(", concat_term, ")") |
 ("(", size_term_body, ")");

append_term = list_term, "<>", ( list_term | append_term );
concat_term = string_term, "++", ( string_term | concat_term );

list_term = var | compound | list_constructor | list_comprehension  |
      ("(", list_term, ")") |
              ("(", append_term, ")");

string_term = (var | compound | string ) |
      ("(", string_term, ")") |
              ("(", concat_term, ")");

list_comprehension =
    "[", (var | ("(", var_list, ")")), "::",
                simple_exists_condition, "]";
set_comprehension =
    "{", (var | ("(", var_list, ")")), "::",
                simple_exists_condition, "}";

var_list = var, {",", var};

tuple_constructor = "(", term, ",", term, {",", term}, ")";

%compound = basic_compound, { term | ("(", ")") };
%basic_compound = (atom | var), (term | ("(", ")"));

compound = (atom | var), compound_args;
compound_args = (basic_term | ("(", ")")), [compound_args].

simple_compound = atom, term;

arg_list = term, {",", term};
braketed_arg_list  = ("(", ")") | ("(", arg_list, ")");

list_constructor = "[", list_constructor_args,
    ("]" | ("|", list_term, "]") |
(",", "..", "]") | (",", "..", list_term, "]")), !;

list_constructor_args = term, [",", list_constructor_args].
```

```
set_constructor = "{", arg_list, "}";

basic_set_term = var | compound | set_comprehension | set_constructor;

set_term = set_inter_expr |
           (set_inter_expr, "union", set_term) |
           (set_inter_expr, "diff", set_term);

set_inter_expr = basic_set_term |
               (basic_set_term, "inter", set_inter_expr) |
               ("(", set_term, ")");

apply_term = basic_term, "@..", (list_term);

global_val = "$", atom;

pedro_addr = (atom | var), [":", (atom | var)], ["@", (atom | var)];
agent_handle = (atom | var), ["@", (atom | var)];


(* ------ conditions ------ *)

condition_seq = condition, {"&", condition};

simple_condition_seq =
    simple_condition, ["&", simple_condition_seq];

simple_condition =
     predication |
     ( "not", predication ) |
     ( "not", "(", predication, ")" );

forall_condition =
    "forall", var_list, "(", simple_condition_seq, "=>",
            simple_exists_condition, ")";

simple_exists_condition =
     simple_condition_seq |
    "exists", var_list, simple_condition_seq |
    "exists", var_list, "(", simple_condition_seq, ")";

exists_condition =
    condition_seq |
    "exists", var_list, condition_seq ;
```

```
condition =
    forall_condition |
    simple_condition |
    ( "once", predication ) |
    ( "once", "(", predication, ")" );

predication =
     compound | atom |
     type_test |
     ( "mode_correct", brac_call_term ) |
     ( "call", brac_call_term ) |
     ( arith_term, ">", arith_term ) |
     ( arith_term, ">=", arith_term ) |
     ( arith_term, "<", arith_term ) |
     ( arith_term, "=<", arith_term ) |
     ( term,  "=", term ) |
     ( term,  "\=",  term ) |
     ( term,  "@>", term ) |
     ( term,  "@>=", term ) |
     ( term,  "@<", term ) |
     ( term,  "@=<", term ) |
     ( term,  "@=", term ) |
     ( term, "in", (list_term | string_term | set_term) ) |
     ( compound, "=?", simple_term, "@..", list_term ) |
     ( ( list_term | append_term ), "=?", append_term ) |
     ( ( string_term | concat_term ), "=?", eq_string_term );

type_test = "type", "(", term, ",", annotated_type_expression, ")";

call_term = var | atom | compound;
brac_call_term = ("(", call_term, ")") | call_term;

simple_call_term = var | atom | simple_compound;
brac_simple_call_term = ("(", simple_call_term, ")") | simple_call_term;

eq_string_term = string_q, "++", string_q, {"++", string_q};
string_q = string_term | (string_term, "?", condition );


(* ------ actions ------ *)

action_seq = action, {";", action};

action = simple_action | forall_action;

simple_action_seq =
```

```
    simple_action, {";", simple_action};

forall_action =
     "forall", var_list, "(", simple_condition_seq, "~>>",
          simple_action_seq, ")";

simple_action =
     compound | atom |
     ( "do", brac_call_term ) |
     ( atom, ":=", arith_term ) |
     ( atom, "+:=", arith_term ) |
     ( atom, "-:=", arith_term ) |
     ( "remember", brac_simple_call_term, ["for", arith_term] ) |
     ( "rememberA", brac_simple_call_term, ["for", arith_term] ) |
     ( "forget", brac_simple_call_term, ["after", arith_term] ) |
     ( "replace", brac_simple_call_term, "by", simple_call_term ) |
     ( "replaceA", brac_simple_call_term, "by", simple_call_term ) |
     ( "start_task", (atom | var), call_term ) |
     ( "kill_task", (atom | var) ) |
     ( "start_agent", (atom | var), agent_handle,
                           ("all" | "updates" | "user") ) |
     ( "log", agent_handle ) |
     ( term, "to", pedro_addr ) |
     ( term, "from", pedro_addr ) |
     ( "fork", brac_call_term, "as", (atom | var) );


(* ------ interpreter entry ------ *)

interpreter_entry =
    exists_condition |
    action_seq |
    ( "watch", atom ) |
    ( "watchC", atom ) |
    ( "unwatch", atom ) |
    ( "set_num_answers", int ) |
    ( int, var_list, "?", condition_seq ) |
    ( var_list, "?", condition_seq ) |
    ( int, "?", condition_seq ) |
    ( "prolog", prolog_calls ) |
    "prolog"  |
    "types" |
    ( "types", atom ) |
    "stypes" |
    ( "stypes", atom ) |
    "show" |
```

```
    ( "show", atom ) |
    ( "consult", atom ) |
    ( "pconsult", atom ) |
    ( term, "::", ( condition_seq ) );

(* can use "&", and "," in prolog calls *)
prolog_calls =
    ( action | condition ),  [("&" | ","), prolog_calls];
```