QuLog: A flexibly typed logic based language with function and action rules

K. L. CLARK

Dept. of Computing, Imperial College, London
ITEE, University of Queensland, Brisbane

P. J. ROBINSON

ITEE, University of Queensland, Brisbane
(*e-mail:* k.clark@imperial.ac.uk, pjr@itee.uq.edu.au)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

QuLog is a declarative flexibly typed higher order logic programming language with syntactic extensions to support strict higher order functional programming and pattern match string processing. It has an imperative action rule language on top of this declarative core. Its logic programming subset is both simpler and more declarative than Prolog. Its dynamic relations may only be defined using facts.

QuLog also supports and the compiler ensures type and mode safe meta-programming. Term lists representing sequences of relation calls and function applications, or sequences of action calls, can be manipulated and eventually evaluated if each call conforms to the type and mode of use constraints of the called code.

In this paper we introduce all the features of the language with an emphasis on its logic programming subset. We do this using example definitions, a semi-formal description of its syntax, and an example based introduction to the type and mode checking abstract interpretation of its complier. This uses formally defined rules given in an Appendix

The paper assumes familiarity with Prolog and typed functional languages such as Haskell.

KEYWORDS: logic programming, functional programming, hybrid declarative/imperative languages, call log debugging, moded types, type safe meta-level programming

1 Introduction

QuLog is a hybrid higher order logic, functional and action rule programming language which has evolved out of many years of extending, teaching and using our multi-threaded Qu-Prolog system. *QuLog* code is typed and moded using type declarations. Its type system is flexible allowing a mix of compile and runtime type checking.

Its logic programming subset is both simpler and more declarative than Prolog. Its dynamic relations may only be defined with facts. They are a sub-type of the static relations defined by fixed facts and rules. There is no disjunction and backtracking control is restricted to the use of committed choice relation rule and the

use of a `once` operator restricting a call or conjunction to a single solution. Explicit universal and existential quantifiers must be used in negated conditions and `forall` quantified implications where they would have to be used in predicate logic for clarity of expression.

Its function rules are syntactic sugar for committed choice relation definitions with ground input mode of use for all but one argument. Arguments to relation calls may be function call expressions. The unification primitive evaluates expressions and does the occurs check unless mode analysis determines it is not needed. List and set comprehension expressions, and pattern match string and list processing, further enhance the usability of the language.

The logic and functional declarative core of *QuLog* was developed to encode the knowledge (the functions and static relations) and the perceptual and other changing beliefs (the dynamic relation facts) of *TeleoR* (4) robotic agents. The *QuLog* action rules, and its action primitives for dynamic fact updating, I/O, thread forking and inter thread and inter process message communication, were added to enable *TeleoR* multi-threaded agents to be implemented entirely in *QuLog*. Action rules can call functions and relations, but not vice versa.

QuLog does not do type inference on code; relations, functions and actions have to have their types declared, and relations and action arguments must have their use modes specified by annotating their type expression. The annotations are `!` (*ground input*), prefix `?` (*ground output*), postfix `?` or prefix `??` (*unconstrained*). There is also a special mode operator `@` for indicating the argument has type `term` and will be left *unchanged*.

`term` and `code` are system types. `code` covers every higher order type, i.e. every relation, function and action type. `term` covers the type of every other value that can be the binding of a variable. The *QuLog* types lie in a complete lattice with `top`, the union of `code` and `term`, as the only maximal element, and `bottom`, with single element the atom `bottom_`, as the only minimal element.

All relation, function and action definitions must be top level statements of the program with names that have scope the entire program. There are no *lambda*s or *lets* in *QuLog*. The use of such constructs can be otherwise programmed by currying functions or relations given top level definitions. We also do not have disjunction or conditionals, because the purpose of each can be achieved using auxiliary top level definitions. On the other hand, we have enriched Prolog syntax by insisting on the use of explicit existential quantification, where a default implicit universal quantification of all the variables of a rule does not capture the declarative semantics, and existential quantification would be required in predicate logic.

QuLog supports type and mode safe meta-programming. It does this by allowing the name *CodeNm* of primitive or program defined code to be used as a value of type `atom`. In the context of a call use $CodeNm(Arg_1, .., Arg_k)$, or when given as the argument of another call where a code value is needed, the type checker will assign $CodeNm$ its code type *CodeType*, and will check the use is type and mode correct. However, if $CodeNm$ is used in a context where a none code value is required, the type checker assigns it the type `atom_naming(`*HEType*`)`, a sub-type of `atom`.

We can 'recover' the code type of an `atom` typed variable $PossCodeNm$ using

a type test `type(`*PossCodeNm*`,atom_naming(`*PossCodeType*`))`. The runtime test will only succeed if *PossCodeNm* is the name of code of the given type. This runtime test is possible in *QuLog* because type declarations for code are accessible at runtime. The use of the type test gives the compiler's abstract interpretation type/mode checker the type expression *PossCodeType* it needs to check the use of *PossCodeNm* as a higher order code value in a call $PossCodeNm(Arg_1, .., Arg_k)$ following the type test, or should *PossCodeNm* be subsequently used as a code argument. After the run time type test the type checker switches the type value of *PossCodeNm* from `term` to *PossCodeType*.

A similar type switch happens to relation and action calls. If a relation or action call $HE(Arg_1, .., Arg_k)$ is used anywhere other than at the top level of conjunctive condition or action sequence, the call is assigned the type `relcall` or `actcall`, both of which are sub-types of `term`. The 'call' value of a variable *CallTrm* of type `term` can be recovered, and at the same time checked for type and mode correctness, using runtime type tests `type(`*CallTrm*`,relcall)`, `type(`*CallTrm*`,actcall)`. The meta calls `call` *CallTrm* and `do` *CallTrm* may then be used to execute the term as a relation call or action call respectively. There is a similar but less direct method for passing around and then invoking function calls as data.

QuLog's action rules are used to program behaviours. Actions can call functions and relations but *cannot* be called from a function or relation. Using actions we can perform atomic updates of dynamic relations, fork new action executing threads, do terminal and file I/O, and do inter-thread and inter-process message communication using ground and non-ground terms. Inter-process communication uses the same message send action primitive using an email style address such as `robot1@loatzu.doc.ic.ac.uk`. The message is then automatically routed to a default message handling thread in the identified process, providing the process is connected to the same Pedro communications server (11). Action rules can be used to implement communicating multi-threaded agents as illustrated in (3).

The paper is structured as follows. In Section 2 the *QuLog* relation subset is introduced. In Section 3 its type definition syntax and runtime type checking primitives are described. In Section 4 the use of the *QuLog* interpreter for developing and debugging programs is illustrated. Its function and action rule subset are illustrated in Sections 5 and 6, respectively Meta level programming is exemplified in Section 7. The primitive *QuLog* types and the syntax of type expressions used to define new types is covered in Section 8, where the sub-type relation of the type lattice is defined. This is a precursor to the description of the type and mode checking done by the compiler in Section 9. The discription uses examples and type checking rules given in the Appendix. We conclude with a discussion of some related work in Section 10. The paper assumes familiarity with Prolog and higher order functional programming, as in languages such as Haskell (10).

2 *QuLog* Relational Subset

Below are some simple data base style relation definitions with their type definitions.

```
male ::= peter | bill | john | ....     % more alternative atom names
```

```
% Enumerated type giving all the names of males being used
female ::=  mary | jane | rose | ...    % ditto for names of females
person ::= male || female       % person is the union of male and female types
age::= 0..120           % Range type giving all the legal values for an age

child_of(C,P): dyn(person,person)  % Type declaration for a dynamic relation
"C is the child of P, a fact defined updateable relation"
% Above is a special remembered comment string
child_of(mary, peter)
child_of(bill,peter)
...   % etc

age_is(P,A): dyn(person,age)   % Another dynamic relation
"A is the age of P, a fact defined updateable relation"
age_is(peter,40)
...   % etc

ancestor_of(A,D): rel(?person,?person)  % A static relation
"For finding or checking ancestor pairs, A is ancestor of D"
ancestor_of(A,D) <= child_of(D,A)
ancestor_of(A,D) <= child_of(D,P) & ancestor_of(A,P)

descendant_is(A,D): rel(person,?person)  % In any call first arg. must be a given
"For finding or checking descendants, A is ancestor of D"
descendant_is(A,D) <= child_of(D,A)
descendant_is(A,D) <= child_of(C,A) & descendant_is(C,D)

only_has_adult_sons(P): rel(?person)
% In a call  P may be an unbound variable or given as a person atom
only_has_adult_sons(P) <=
    isa(P,person) &   % This checks P is a male or female, or finds one
    forall C (child_of(C,P) => exists A isa(C,male) & age_is(C,A) & A>20)
```

Notice there are no full stop terminators for type declarations and rules. They can be given but are ignored. Text starting at the left end of a newline is what signals the end of the previous *QuLog* statement, a convention we copied from Python. Type declarations and rules can be given over several lines just by indenting to the right all but the first line.

QuLog has two forms of comment. Any text following a `%` up to the end of line is ignored as a comment. Any text enclosed in `/*....*/` brackets, that may go over several lines, is treated as a comment. Finally, a string enclosed in `"..."` quotes, *immediately* following a type declaration, is a comment but it is *remembered* and will be displayed whenever the type declaration is displayed in the interpreter, as illustrated in Section 4. The remembering and re-display of a specially located comment associated with some code is another idea we borrowed from Python.

In the code type declarations the use of a call term pattern such as `child_of(C,P)` is optional but allows the arguments to be refereed to succinctly in comments. We could just use the code name as in `child_of: dyn(person,person)`.

The enumerative type definitions for `male` and `female` mean that only the given atoms can be used in the `child_of` and `age_is` facts. `person` is defined as the union

of these two types. `age` is a range type. If any fact for these relations is not type correct there is a compile time type error.

`male` and `female` are both sub-types ($\leq$) of `person`, where `person` $\leq$ `atom` $\leq$ `atomic` $\leq$ `term` $\leq$ `top`. `top` is the top of the *QuLog* type lattice. `top` is also a direct super-type of `code`, a super-type of all relation, function and action types.

The `dyn` wrapper for the argument type tuple of `child_of` and `age_is` indicates the relations are defined only by facts and that these facts can be updated by interpreter commands and action rules. Their arguments are implicitly `?` moded allowing test or generate use for either or both arguments.

The `rel` wrapper indicates a static relation. These are usually defined by rules but may be defined using fixed facts. In their type declaration we have to indicate their allowed modes of use. The `ancestor_of` relation use mode is test or generate a ground value for both arguments, indicated by the `?` mode annotations for each argument type. In contrast, in calls to `descendant_is` the first argument has to be given as a `person` atom because its first argument does not have the prefix `?`, so is implicitly `!` (ground input) annotated. Its definition is logically redundant but its use for finding descendants is much more efficient than the logically equivalent `ancestor_of` because it will 'walk' down the descendant chain from the given ancestor `A` whereas the `ancestor_of` definition will search for an ancestor by testing if the parent of each `child_of` pair is a descendant. `descendant_is` is similarly inefficient for finding ancestors of a given descendant, but its mode prevents its being used for this purpose. Any such attempted use will raise a mode error. If we know that we would not need to find ancestor pairs we could use the moded type `rel(?parent,parent)` for `ancestor_of` causing a mode error when a call does to have the second argument given.

`isa` is a test or generate type checking primitive for types with a finite domain. These are enumerated or range types, or unions of such types. Its first use `isa(C,male)` in the `only_has_adult_sons` rule is a test or generate use. Its second use `isa(C,male)` is a checking use for which we could also use `type(C,male)`. `type` is the more general type check primitive as its second argument can be any type expression, but it may only be used for checking.

The explicit existential quantification in the consequent of the *forall* quantified implication is needed in *QuLog*, as it would be needed in the equivalent predicate logic universally quantified implication.

Recursive types

`list` is a primitive recursive parameterised data type. New parameterised recursive data types can be defined as in Haskel. Here is a definition for a binary tree structure with root labels of any type `T`, with a polymorphic relation that will check or find labels on a given ground tree.

```
tree(T) ::= empty() | tr(tree(T),T,tree(T))
 % empty and tr are unique constructor functors of the new type
on_tree: rel(?T, tree(T)) % The tree arg. must be a ground term in any call
on_tree(Lab, tr( _, Lab, _)
on_tree(Lab, tr(Left, _, _) <= on_tree(Lab,Left)
```

```
on_tree(Lab, tr(_, _, Right) <= on_tree(Lab,Right)
```

Using run-time type tests

Our next example has a necessary type test condition. `add_nums_on_any_list` can be given a complete list of any terms, including terms that are or contain variables. It may be used to find or check the sum of all the numbers appearing on the list, ignoring non-number values and variables.

```
add_nums_on_any_list(TermList,Sum): rel(list(@),?num)
"TermList is a complete list of possibly non ground terms, which will
 not be further instantiated by a call to the relation"
add_nums_on_any_list([E,..L], Sum) :: type(E,num) <=
      add_nums_on_any_list(L,LSum) & Sum = LSum+E
add_nums_on_any_list([E,..L], Sum) :: not type(E,!num) <=
      add_nums_on_any_list(L,Sum)
```

The `@` signals the argument may be a variable or any ground or non-ground term. It also tells us that the argument will be *unchanged* by a call to the relation, that no variable will be bound.

The type test `type(E,num)` will only succeed if `E` is a given number. It fails if it is a variable. `type(E,?num)`, with the prefix `?` on the type expression, would succeed even if `E` was a variable or a number.

`::` should be read as *such that*. The conditions before the `::` are a rule's single solution *commit test*. If they hold no later rule will be used to find an instance of the relation and no second solution of the *commit test* is sought even if the conditions following the `::` have no solution. Given this semantics, the last rule's complement test `not type(E,num)` may be dropped and the evaluation behaviour will be the same.

Type expressions as arguments

Here is a relation that will filter from a list of ground terms all those terms of a specific type, the type expression being given as the last argument.

```
filter_from_any_ground_list_TE(L1,L2,TE): rel(list(T1],?list(T2),typeE(T2))
"L1 is a list of ground terms, L2, which may be generated, is the sub-list
 of L1 of just those members with type denoted by type expression TE"

filter_from_any_ground_list_TE:([],[],_)
filter_from_any_ground_list_TE([E,..L], [E,..FL],TE) :: type(E,TE) <=
      filter_from_any_ground_list_TE(L,FL,TE)
filter_from_any_ground_list_TE([E,..L], FL,TE) :: not type(E,TE) <=
      filter_from_any_ground_list_TE(L,FL,TE)
```

A call

```
 filter_from_any_ground_list_TE([2,apple,7.5,-7,"hello"],FL,int)
```

will succeed binding `FL` to `[2,-7]` and the type checker will be able to infer that `FL` will be a list of type `list(int)`, from the fact that the type expression argument is given as `int`.

Relations as arguments

An alternative way of achieving the same filtering is to pass in a test relation, that is to make the filter relation higher order.

```
filter_from_any_ground_list_HO(L1,L2,Test): rel(list(T),?list(T),rel(T))
"L1 is a list of ground terms, L2, which may be generated, is the sub-list
 of L1 of just those members satisfying the condition Test"
filter_from_any_ground_list_HO([],[],_)
filter_from_any_ground_list_HO([E,..L], [E,..FL],Test) :: Test(E) <=
      filter_from_any_ground_list_HO(L,FL,TE)
filter_from_any_ground_list_HO([E,..L], FL,TE) :: not Test(E)  <=
      filter_from_any_ground_list_HO(L,FL,TE)
```

A call

```
filter_from_any_ground_list_HO([2,apple,7.5,-7,"hello"],FL,integerTest)
```

where

```
integerTest: rel(term)
integerTest(Trm) <= type(Trm,int)
```

will also succeed binding `FL` to `[2,-7]`. The drawback is that the type checker will only be able to infer that `FL` is of the same type as the list given as the first argument, which has inferred type `list(num||atom||string)`.

The inability to infer that `FL` will be of type `list(int)` for this use of the higher order version is compensated by the fact that we can now have any test, not just a type test. We can filter out all the numbers between -2 and 2 for example by passing in the test relation:

```
numRange: rel(term)
numRange(Trm) <= type(Trm,num) & -2 =< Trm & Trm =< 2
```

Comprehension expressions

Instead of Prolog's `bagof` and `setof` there are two forms of comprehension expressions, both may have existentially quantified conditions and be used as arguments of relation, function and action calls.

```
adult_children_of: rel(?person, ?list(person))
adult_children_of(P,Child_List) <=
    isa(P,person) &
    Child_List=[C :: exists A child_of(C,P) & age_is(C,A) & A>17]
 % Child_List is the list of adult children of P in order found
ages_all_adult_children: ?set(atom)  % Will test or generate a ground set
ages_all_adult_children(AgeSet) <=
     AgeSet = {A :: exists P, C isa(P,person) & child_of(P,C) & age_is(C,A)}
% Age_Set is the set (no duplicates) of ages of all adult children
```

Their general form is:

Term `exists` *Vars SimpleConj*

inside `[..]` or `{..}` brackets where the `exists` *Vars* is optional.

There are constraints on their use. All global variables of the expression - variables used elsewhere in the query or rule in which the expression appears - must

have ground values before the expression is evaluated. All other variables must either appear in *Term* or be in the sequence of existentially quantified *Vars*. Finally, all local variables appearing *Term* must be given ground values by the query condition *SimpleConj*. The two ground value conditions ensure that comprehension expressions only generate lists and sets of ground values.

There is a relational form of list comprehension, on the lines of Prolog's `findall`, that may be used to aggregate all answers to a query where some or all the answer terms *may not be ground*. Its form of use is:

`L listof (`*Term* `::` `exists` *Vars* *SimpleConj*`)`

where the `exists` *Vars* is optional.

```
common_elements: rel(list(T),list(T),?list(T))
common_elements(L1,L2,L) <= L listof (E :: member(E,L1) & member(E,L2))
```

defines a relation that will test or find an `L` that is the sublist of terms on `L1` that also appear on `L2`, which may include non-ground terms.

The constraints on its use regarding global and local variables not in *Term* are the same as for list and set comprehension expressions. The one difference is that shared local variables of *Term* and *SimpleConj* do not have to be given ground values by the *SimpleConj* condition.

Manipulation of sets and lists

Sets and lists are different types. Literal list values are sequences of terms surrounded by `[..]` brackets, e.g. `[a,3.4,f(X),Y]`, and may contain non-ground terms. Literal sets are sequences of ground terms surrounded by {`..`} brackets, e.g. {`a,3.4,f(3),a,"hello"`}. Duplicate terms are ignored, so this set is reduced to have only one occurrence of `a`.

The primitive `in` relation will access elements of lists and sets. It will return the list elements in their before after order, and set elements in `@<` Prolog term order. `E in` {`a,3.4,f(3),a,"hello"`} will generate successive bindings `E=3.4`, `E=a`, `E="hello"`, `E=f(3)`.

Sets are manipulated using `union`, `inter` and `diff` primitives. Lists are manipulated using the primitive `<>` function and the primitive `append` relation. Being a function, `<>` will only concatenate ground lists whereas `append` has the flexibility of use of the Prolog `append`.

`<>` does have another use for splitting ground lists. When one or more `<>` operators appear at the top level of the right hand side of `=?` it acts as a *non-deterministic* or multi-valued match operator. As an example, a call

```
L =? L1::(#L1>0) <> [U]::type(U,int) <> L2
```

will split a ground list `L` at each integer element `U`, not the first element of `L` as `L1` has to have non-zero length, making `L2` the remainder of the list after `U`. If `L=[3,4.5,2,a,4]` the call has solutions `L1=[3,4.5], U=2, L2=[a,4]` and `L1=[3,4.5,2,a], U=4, L2=[]`.

The above example illustrates the use of tests such as `type(U,int)` to constrain the assignments to the pattern variables. The general form of such tests is

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

Failure of the test will cause backtracking to find an alternative value for `Var`. When there are no more alternatives 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.

Another example use is in the following definition.

```
splitsAroundE(L1,L2,E,BefE,AftE): rel(list(T),list(T),T,?list(T),?list(T))
"BelE, AftE are sublists before and after each E in L1<.L2"
splitsAroundE(L1,L2,E,OL1,OL2) <= L1<>L2 =? OL1<>[E]<>OL2

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

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

String pattern matching

For string manipulation we can use ++ for both string concatenation and splitting. 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=""`.

The pattern match

```
"hello" =? S1::#S1>0 ++ S2::#S2>0
```

excludes the first and last solutions we gave above. `#` is the *QuLog* primitive for finding the length of a string or list, or the size of a set.

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.

```
spaces: rel(string)
spaces(S) :: (S =? " " ++ RS::spaces(RS))  % S has more than 1 space
spaces(" ")                                % We have a single space string

sepchar: rel(?string)
sepchar(" ")
sepchar(",")
sepchar(";")

endchar: rel(?string)
endchar(".")
```

```
endchar("?")
endchar("!")

sep_endchar: rel(?string)
sep_endchar(C) <= sepchar(C)
sep_endchar(C) <= endchar(C)

wordchar: rel(string)                          % Test only use
wordchar(S) :: 1 = #S & not sep_endchar(S)
 % S is a single character string which does not contain a sep or end character

word: rel(string)
word(S) :: (S =? C::wordchar(C) ++ RS::word(RS))
word(S) :: wordchar(S) % Single char alternative for a string being a word

seps: rel(string)
seps(Seps) :: Seps =? (O::sepchar(O) ++ RSeps::seps(RSeps))
seps(Sep) <= sepchar(Sep)

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

`words` converts a string sentence beginning with at least one word substring terminated with an end char, and with spaces and/or any number of sep chars between each word, into a list of word strings. An example use is

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

which generates the binding:

```
Ws = ["Hello","Keith","how","are","you"]
```

2.1 General form of relation definitions

These comprise a type declaration using a new relation name r, optionally immediately followed by an associated `"..."` string comment, and somewhere in the same program file a sequence of contiguous facts and/or relation rules for r.

General Form of Relation Declarations

There are two forms of relation in *QuLog*. Dynamic relations defined only by a sequence of updatable ground facts and static relations which are defined by a sequence of non-changing facts and rules. A dynamic relation might not have any facts in the program file, just a declaration.

A dynamic d relation is declared:

d: `dyn(`t_1`,....,`t_k`)`

where the t_i are unmoded type expressions. A dynamic relation does not need

explicit mode annotations. Being defined only by facts it can be queried in any mode.

A static relation r is declared:

r: `rel(`mt_1`,....,`mt_k`) | rel(`mt'_1`,....,`mt'_k`) | ...`

Here each mt_i, mt'_i etc. is a type expression with a mode annotations or `@` on its own. The mode annotations are prefix `!`, `?`, `??`, or a postfix annotation `?`. No mode annotation is the same as prefix `!`. Prefix `??` and postfix `?`, are alternatives.

`!` indicates *ground input*, that in a call that argument must be given and be ground. Prefix `?` indicates *ground output*, that in a call a variable or term containing variables may be given but that the argument will be ground on success of the call. Postfix `?` or prefix `??` indicates *unconstrained*, that the call argument can be as for prefix `?` mode but is not necessarily ground on success of the call. `@` indicates *unchanged*, any term as argument value which will always be left unchanged by a successful call.

If a type expression is an instance of a parameterised structure type, such as `list(int)`, the type expression for the structure's elements may also be moded with a more *relaxed* mode than that for the structure. Here postfix `?` is more relaxed than prefix `?`, which is more relaxed than `!`. For example, we can have a moded type `list(?int)`, `list(int?)` or `?list(int?)`, meaning respectively: a complete list which may contain variables *all* of which will have integer values on success, a complete list which may contain variables *not all* of which may have integer values on success, a partial list (a list such as `[3,-6,N,..L]` where `L` is a variable) which will be a complete list, possibly containing variables, on success (for example `[3,-6,N,8]`).

If only a mode annotation `m` is given this is equivalent to `m term`, e.g. `?` on its own is shorthand for `?term`. The omission of a mode annotation on a type expression is equivalent to there being a prefix `!`, indicating a ground argument value is required. Generally, the `!` is dropped.

The fully moded declaration `test: rel(!rel(?nat,num),!atom)` tells us that `test` is a relation which must be given an atom second argument and a relation first argument. The relation will be used always giving a `num` second argument but the first argument may be a `nat`, or a variable to be bound to a `nat`.

A relation can have more than one type declaration. There are constraints on the allowed alternatives checked by the compiler to ensure that when type/mode checking a program there is always just one minimum type alternative that covers the inferred types and groundedness of the call arguments.

The primitive `append` (list append) relation has the *QuLog* definition:

```
append: rel(list(T)?,list(T)?,list(T)?) | rel(list(T),list(T),?list(T)) |
        rel(list(T?),list(T?),?list(T?)) | rel(?list(T),?list(T),list(T)) |
        rel(?list(T?),?list(T?),list(T?))
append([],L,L)
append([U,..L1],L2,[U,..L3]) <= append(L1,L2,L3)
```

The five type descriptions differ only regarding modes. The first is the most general use where all arguments can be variables or list patterns, and may each be left as

such terms after a successful call. The second type declaration is the appending use of ground lists, and the fourth is the splitting use of a ground list. The third and fifth are relaxations of the second and fourth. They treat the cases of appending and splitting complete lists of possibly non-ground terms. Their declarations say that complete lists will be generated, but they may contain non-ground terms.

General form of Dynamic Relation Facts

A fact for a dynamic relation d has the form

d`(`Arg_1`,...,`Arg_k`)`

where each Arg_1 is a ground term or code name of the required type t_i of d's type declaration. The fact may not contain any variable.

Relation rules

QuLog has two forms of relation rules, *unguarded* and *guarded*.

An *unguarded rule* for a relation is written

Head `or` *Head* `<=` *ComplexConj*

where *Head* is a *head predication* of the form $rel(Arg_1,...,Arg_k)$, $k \geq 0$. Here *rel* has atom syntax but is different from any `atom` of an enumerated type, and any name used as a constructor of a defined type, and no Arg_i contains a function call.

A *ComplexConj* has the form $Cond_1$ `&` $Cond_2$ `&` `...` `&` $Cond_n$`,` `n`$\geq$ `1`, with $Cond_i$

- an *expression predication* $RelExp(Exp_1,...,Exp_k)$, $k \geq 0$, where each Exp_i is an expression - a term that may contain function calls. *RelExp* is an expression returning a k-ary relation *rel'* such that the values of the argument expressions will satisfy the mode and type constraints of *rel'* when it is called.
- a negated condition `not (exists` *VarSeq SimpleConj*), where *VarSeq* is a sequence of local variables of *SimpleConj*. This is the *QuLog* negation-as-failure (1) operator
- a body predication, or a `(..)` bracketed *SimpleConj*, prefixed with `once` - find one solution only
- an expression value unification $Exp_1 = Exp_2$
- a non-deterministic pattern match *Exp =? PtnTerm*
- a meta-call `call` *Call*
- a meta-call `apply(`*F,Args,Term)*
- a universally quantified implication (a *forall*) of the form
  `forall` $\mathtt{V}_1$`,...,`$\mathtt{V}_j$ `(exists` $VarSeq_1$ $SimpleConj_1$ `=>`
  `exists` $VarSeq_2$ $SimpleConj_2$`)`, `j`$\geq$`1`
  The $\mathtt{V}_i$ variables are *universally* quantified over the implication. The sequence of variables of $VarSeq_i$ are existentially quantified over $SimpleConj_i$, i=1,2.

A *SimpleConj* is a *ComplexConj* which does not contain any *forall*.

In a negated condition the existentially quantified variables are local variables of *SimpleConj*, variables that do not appear elsewhere in the query or rule body in which the negation appears. All other variable of *SimpleConj*, except _ underscore anonymous variables, must be given ground values before the condition is evaluated. (_ variables are always implicitly existentially quantified just before the predication in which they appear.) If there are no such local variables the `exists` *VarList* is absent. The `(...)` brackets are not needed if there is only one negated condition.

In a `forall`, each variable $\texttt{V}_i$ *must* appear in both $SimpleConj_1$ and $SimpleConj_2$. It must be a local variable of the `forall`. All other variables of the `forall`, except the existentially quantified variables $VarsSeq_1$ and $VarsSeq_1$ and _ underscore variables, must have ground values when the `forall` condition is evaluated. Each $VarsSeq_i$ contains variables local to $SimpleConj_1$.

The type of `call` is `rel(relcall)`. `relcall` is a system generated term type. For every program defined and primitive relation $r : (m_1t_1, .., m_kt_k)$, where the m_i are modes, the system type `relcall` has a constructor $r(t_1, .., t_k)$.

`relcall` is a very special type because a `relcall` term *Call* is considered ground if the functor r is given and there is a type declaration for r that is type and mode compatible with the arguments of *Call*, i.e. only the required ground input arguments of the compatible declaration need to be ground in *Call*.

If *Call* has been generated by a proceeding call as a value of type `term`, or passed into the relation in which the meta call is used as an argument of type `term`, the meta-call has to be preceded by a run-time type test `type(Call,relcall)`. This type test checks that term *Call* is a compound term of the system type `relcall`, *and* that all the arguments that must be ground are ground. The use of `call` and the `relcall` type is illustrated in Section 7.

The type of `apply` is `rel(T->T,T,?T)`. An example us is `apply(+,(2,4),R)` which will bind `R` to `6`, the value of `2+4`. It allows us to invoke any function as a relation call and is particularly useful for meta programming as illustrated in Section 7.

The unification primitive $Exp_1 = Exp_2$ evaluates its expression arguments and unifies the values. It does an occurs check unless the compiler can determine that one of the expression arguments will be a ground value when the `=` is evaluated, or one argument contains only new variables outside its function calls.

The *PtnTerm* right hand side of a `=?` is restricted to three forms of patterns: a top level sequence of `++`'s or `<>`'s or a *Functor@..ArgList* term. Its use is for pattern match decomposition of strings, lists, and compound terms.

A *guarded rule* for a relation has the form:

Head `::`	*Head* `::` *SimpleConj*
Head `::` `<=` *ComplexConj*	*Head* `::` *SimpleConj* `<=` *ComplexConj*

The first rule form is equivalent to *Head* `::` `true` `<=` `true`, the second rule to *Head*

`::` `true` `<=` *ComplexConj*, the third to *Head* `<=`*SimpleConj* `::` `true`. `true` is the no argument *QuLog* primitive that is trivially provable.

The *SimpleConj* between the `::` and the `<=` is the rule's *commit* test. When such a rule is used for a call *Cond*, should the rule's *SimpleConj* succeed, no later rule for the predicate of *Cond* will be used to find solutions and only one solution to *SimpleConj* is found. In Prolog terms, there is a cut after the commit test.

3 Defining New Types

There are five forms of type definition: *enumerative*, *range*, *constructor*, *union* and *macro*. All program defined types must have a unique name different from any of the primitive type names.

An *enumerative* type definition has the form:

typeName `::=` v_1 `|` `...` `|` v_k

where the v_i are different atomic terms, not necessarily the same type. The defined *typeName* is a sub-type of `atomic`. But it will also be a sub-type of a more specific type, e.g. `num`, providing all its values belong to that sub-type.

Overlapping enumerative type definitions are allowed providing the set of values of one is a proper subset of those of the other. This allows the type inference system to assign a unique minimal enumerative type to any atomic value included in at least one enumerative type. We can have partially overlapping sets of atomic values as different types using type unions. For example:

```
digit ::= 0 | 1 | ... | 9
small_nat ::= 0 | 1 ... | 5
small_neg_int ::= -1 | ... | -5
small_int ::= small_neg_int || small_nat
```

small_int comprises the integers `-5, -4,..., 0, 1. ..., 5` but is a not the minimal type for any of its values.

A *range* type definition has the form:

typeName `::=` *m* `..` *n*

where m and n are integers with $m < n$. It is equivalent to the enumerative type:

typeName `::=` *m, m+1, ..., n*

and is subject to the same constraints regarding overlapping types.

A *constructor* type definition has the form:

typeName `::=` c_1`(`t_1`,..,`t_n`)` `|` c_2`(....)` `|` c_k`(...)`

or

typeName`(`*T*`)` `::=` c_1`(`t_1`,..,`t_n`)` `|` c_2`(....)` `|` c_k`(...)`

Here each t_i is a type expression which may containg variable T in the second form of definition. The c_i constructor names are atoms. A constructor name may appear more than once providing it has the same number of arguments with different types. This flexibility is rarely needed. It can also be used in other constructor type definitions which are either restrictions or extensions of this type. That is they

are either sub-types, having fewer constructors but with the same argument types, or they are super-types having these types constructors, with the same argument types, and more. A constructor type is only a sub-type of `term` and `top`.

A *union* type definition has the form:

typeName `::=` $\texttt{te}_1$ `||` $\texttt{te}_2$ `||` `...` `||` $\texttt{te}_k$

where each te_i is a type expression. *typeName* is a sub-type of any other union type which includes a super type for each type te_i.

A *macro* type definition has the form:

typeName `::=` *te*

where each *te* is a type expression. It gives the name *typeNm* to that type expression for brevity. A union type definition is also a macro type definition since a union of types is a type expression. Type expressions are described in Section 8, where the type lattice sub-type relation is also defined.

Type test primitives

There are two general type test primitives for determining whether a data term has a specific program defined or primitive type, `type` and `isa`.

`type(`*trm,te*`)` - *test only use.* Will succeed if *trm* is a term of type denoted by the type expression *te*.

`isa(`*trm,fte*`)` - *test or generate use. fte* must be a type expression denoting a finite extension type. If *trm* is a variable it can be used to find instances of the type. For example, `isa(P,person)`.

Program dependent system types

There are four special program dependent types generated by the compiler:

- `dyncall` - A compound term of the form *d(*arg_1*,...,*arg_k*)* where the program contains a declaration *d*:`dyn`$(t_1, .., t_k)$ and each arg_i is either of term of type t_i or a variable. That is, if `type(Trm,dyncall)` *Trm* denotes a mode and type correct call to the dynamic facts about *d*. It need not be ground.
- `dynfact` - A `dyncall` term that contains no variables, it is ground.
- `relcall` - A `dyncall` term, or a compound term of the form *r(*arg_1*,...,*arg_k*)* where the program contains rules for a relation with a moded type declaration of the form $r : rel(t_1, ..., t_k)$, or r is a system defined relation with such a moded type declaration, and each arg_i is either of term of type t_i or a variable, *and* where an argument is moded `!` the corresponding argument of *r(*arg_1*,...,*arg_k*)* is ground. That is, if `type(`*Trm*`,relcall)` *Trm* denotes a mode and type correct call to the relation *r*.
- `actcall` - A is a compound term of the form *a(*arg_1*,...,*arg_k*)* where the program contains rules for an action with a moded type declaration of the form

$a : act(t_1, ..., t_k)$, or a is a system defined action with such a moded type declaration, and each arg_i is either of term of type t_i or a variable, *and* where an argument is moded `!` the corresponding argument of $a(arg_1,...,arg_k)$ is ground. That is, if `type(`*Trm*`,actcall)` *Trm* denotes a mode and type correct call to the action a.

4 Interpreter queries and commands, and watch debugging

At the *QuLog* interpreter prompt `| ??` one can enter:

- a *ComplexConj* relation query, possibly prefixed with an existential quantification of those variables whose answer bindings should not be displayed
- an expression - a list or set comprehension, an arithmetic expression, a call to a primitive function such as `cos`, or a call to a program defined function
- an action sequence of the form $ActOrCond_1$; ... ; $ActOrCond_n$, $n > 0$ where each $ActOrCond_i$ is either a *SimpleConj* condition, or an *action call* of the form $ActExp(Exp_1,...,Exp_k)$, $k \geq 0$, with *ActExp* is an expression returning a $k-ary$ action, `a'`, such that the values of the argument expressions of the action call will satisfy the mode and type constraints of *a'* when it is called

The first and last entries are terminated by a full stop, return, the second by an exclamation mark, return, Each will be type/mode checked. Answers are displayed paired with their minimum type in the type lattice.

All current program type definitions and type declarations will be displayed in response to a `types` action. All system type definitions and type declarations for primitives will be displayed, along with a short comment indicating the purpose, in response to `stypes`. `types Nm` will display the type of the code named `Nm`, `stypes SNm` will display the type of system defined `SNm`, with a descriptive comment. `show Nm` will display the type and the rules for program defined `Nm`.

The changeable default response for a relation query is to display up to five answers at a time, each new batch being given in response to a `..` input.

```
| ?? app([dog,..L1],[1,..L2],[U,3,V,"pear",7.5]).
L1 = [3] : [nat]
L2 = ["pear", 7.5] : list(num || string)
U = dog : atom
V = 1 : nat

| ?? 2 of exists P child_of(C,P) & age_is(C,A) & A>17.
C = mary : female  % Most specific type of mary displayed
A = 19 : age
...   % Displayed between different answers
C = bill : male  % Most specific type of bill displayed
A = 23 : age
..
% The .. input will cause display of 2 more answers, if there are 2 more
| ?? [(C,A) :: exists P child_of(C,P) & age_is(C,A) & A>17] !
[(mary,19), (bill,23),....] : list((person,age))
              % A list of pairs of persons and ages
```

The type of the list binding `["pear",7.5]` for query variable `L2` is `list(num || string)`, a list of numbers or strings.

```
| ?? [qlexamples].  % Consult file in current directory named qlexamples.qlg
Consulting qlexamples.....qlexamples consulted
| ?? show app.
app: rel([T]?,[T]?,[T]?) | rel([T],[T],?[T]) | rel([T?],[T?],?[T?] |
     rel(?[T],?[T],[T]) | rel(?[T?],?[T?],[T?])
app([],L,L)
app([U,..L1],L2,[U,..L3]) <= append(L1,L2,L3)
| ?? stypes cos.
cos : num -> num  % Returns the cosine of radian value argument.
| ?? replace age_is(peter,A) by age_is(peter,A+1).
Type Error: A + 1 has type num but is required to be of type age in
age_is(peter, A + 1)  ( with age_is : (?human, ?age) <= )
of the action replace age_is(peter, A) by age_of(peter, A + 1)
```

Unlike the Prolog `listing`, *QuLog* show displays the rules with variable names of the program file. `replace .. by ..` is a primitive action and its use has caused a type error. This is because the type checker cannot infer that `A+1` will be in the range 0 to 120 knowing that `A` has that range. Indeed, `A+1` might well be 121. What is needed is a reformulation and a run-time type check to satisfy the type checker.

```
| ?? age_is(peter,A) ; NewA = A+1 ; isa(NewA,age) ;
     replace age_is(peter,A) by age_is(peter,NewA).
```

The presence of the `isa(NewA,age)` test before the `replace` tells the type checker that the new fact will be type correct. If the type test fails, the old fact is left in place and the action sequence fails. Notice `;` separates the actions and conditions. There is a `forall` action iterator, useable in action rules, with the consequent an action sequence. Apart from that, there is no backtracking in an action sequence.

There is no query trace in *QuLog*. Instead, a `watch` can be put on any number of relations, functions and actions when using the interpreter. Then every call to watched code is logged, no matter when called in some evaluation. The rule heads with which it unifies or matches are displayed, as are the bindings that result. For relations and actions these are split into *input* bindings for variables of the rule, and *output* bindings for variables of the call. Whether the use of the rule succeeds or fails, and the instantiated call result of a successful use of that rule, are also logged. The `watchC` action will additional display the instantiated body of the rule being used. `watch` invisibly inserts writes into the code which cannot be inserted by the programmer, as they are actions. They will not be displayed by `show`. `unwatch` removes them. Below is part of the output for the `app` query above with a `watch` on `app`.

```
| ?? app([dog,..L1],[1,..L2],[U,3,V,"pear",7.5]).
1:app([dog,..L1], [1,..L2], [U, 3, V, "pear", 7.5])
  Call 1 unifies clause 2
       input U_0 = dog L2_0 = [1 |L2] L3_0 = [3, V, "pear", 7.5]
       output U = dog
2:app(L1, [1,..L2], [3, V, "pear", 7.5])
....  % Similar output for recursive app call
3:app(L1_1, [1,..L2], [V, "pear", 7.5])
  Call 3 unifies clause 1
       input L2_2 = [1, "pear", 7.5]
       output V = 1 L2 = ["pear", 7.5] L1_1 = []
```

```
3:app([], [1, "pear", 7.5], [1, "pear", 7.5]) succeeded
2:app([3], [1, "pear", 7.5], [3, 1, "pear", 7.5]) succeeded
1:app([dog, 3], [1, "pear", 7.5], [dog, 3, 1, "pear", 7.5]) succeeded
```

5 Function Definitions and Currying

The functional subset of *QuLog* is syntactic sugar for relations with a single mode of use whereby all but the last argument have `!` mode and the last has prefix `?` mode. Function call expressions can entered as top level queries in the interpreter and appear as arguments of relation and action calls. Such expressions *cannot* appear in the heads of relation, action or function rules. Functions allow much more compact programs to be written.

5.1 Example Function Definitions

```
double: num->num
double(N) -> 2*N

flatten: tree(T) -> list(T)
"A polymorphic function for flattening a tree of labels into a list"
flatten(leaf()) -> []
flatten(tr(Left,Lab,Right)) -> flatten(Left)<>[Lab]<>flatten(Right)

fact: nat -> nat
"Returns factorial value for a natural number"
fact(0) -> 1
fact(N) :: N_Less1 = N-1 & type(N_Less1,nat) -> N*fact(N_Less1)

mapF(Fun,List): ((T1->T2),list(T1)) -> list(T2)
"Returns the list got by applying Fun to each element on List"
mapF(_,[]) -> []
mapF(Fun,[N,..Nums]) -> [Fun(N),..mapList(Fun,Nums)]
```

`<>` is the primitive list appending function. The type test of the second `fact` rule both checks `N>0` and assures the type checker that the recursive call is type correct.

`mapF` is a higher order function in that it takes a function as an argument. A call `mapF(double,[2.5,3.1,4.6])` will return `[5,6.2,9.2]`.

If we have a function of more than one argument it is often convenient to create a *closure* or partial application, a function of fewer arguments in which some of the arguments of the original function have fixed values. For example, we may want to pass in a function to the `mapList` function that multiplies each of the elements of the argument list by some specific number `N`. We can do this by defining:

```
multby: num -> (num -> num)
multby(N)(M) -> N*M
```

One can now pass in to the list map function, `multby(7.5)`, which is a function of one argument of type `num->num`. This closure function will multiply its argument by `7.5`. So, `mapList([1.2,4,7,...],multby(7.5))` will return a list in which every number on the list `[1.2,4,7,...]` is multiplied by `7.5`.

We can do the same with relations.

```
a_parentOf: person -> rel(?person)
a_parentOf(C)(P) <= child_of(C,P)

a_childOf: person -> rel(?person)
a_childOf(P)(C) <= child_of(C,P)
```

We can pass as an argument the monadic relation `parentOf(tom)` whenever a monadic relation of type `rel(parent)` or `rel(?parent)` would suffice. When applied it will test or find the parents of `tom`. If we pass as an argument the monadic relation `a_childOf(mary)`, it may be used to test of find children of `mary`.

Currying

We can support this partial application by defining higher order functions for creating a partial application of a function or relation. This is called *currying*. Here we define currying functions for two argument functions and relations but the same technique can be used to return partial applications where there are any number of arguments.

```
curry(F) : ((T1,T2) -> T3) -> T1 -> T2 -> T3  % -> is right associative
"Allows two stage invocation of F. curry(F)(X)(Y) is equivalent to F(X,Y)"
curry(F)(X)(Y) -> F(X,Y)

curryR(R): (rel(T1,T2)) -> T1 -> rel(T2) |
           (rel(T1,?T2)) -> T1 -> rel(?T2) |
           (rel(T1,T2?)) -> T1 -> rel(T2?)
"Allows two stage calling of R. curryR(R)(X)(Y) = R(X,Y)"
curryR(R)(X)(Y) <= Rel(X,Y)
```

curry(*)(7.5) is the same function as `multby(7.5)`. `curryR` has alternative types because of the alternative possible uses of the monadic curried relation - testing, generating a ground value, generating a possibly non-ground value. Note it can be given a relation that has a more relaxed mode for its first argument. That is, the relation being curried need not require its first argument to be ground, because both the prefix `?` and postfix `?` modes *allow* the first argument to be ground. More technically, this is because the relation types `rel(?T1,m T2)`, `rel(T1?,m T2)` are both sub-types of the relation type `rel(T1,m T2)`, for any mode `m`.

`curryR(child_of)` is a function of type `person -> rel(?person)`. It is the same as the above `parentOf`.

`curryR(child_of)(tom)` is a monadic relation of type `rel(?person)` that can be passed as an argument to a higher order function or relation requiring a monadic relation over the `person` type. The curried relation can be used to find or check the parents of `tom`.

What is actually passed as an argument is the term `curryR(child_of)(tom)`. When this closure is invoked, to find the age of the 'hidden' person, the relation rule

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

is used to evaluate the call

```
curryR(child_of)(peter)(P)
```

by replacing it by `child_of(tom,P)`.

Example expression queries

```
| ?? fact(4)!
24 : age
 % age is given as type of 24 as this is its minimal type
 % because of type def.  age::= 1..120 in the consulted program

| ?? curry!
curry : (T1,T2 -> T3) -> (T1 -> T2 -> T3)
 % -> is right associative

| ?? curry(+)!
curry(+) : nat -> nat -> nat | int -> int -> int | num -> num -> num
 % Answer just gives the type of the expression query

| ?? curry(+)(8)!
curry(+)(8) : nat -> nat | int -> int | num -> num

| ?? curry(+)(8)(7.5)!
15.5 : num

| ?? mapF(curry(*)(3),[1,7,-4])!
[3, 21, -12] : list(int)

| ?? curryR(age_is)!
curryR(age_is) : person -> rel(?age)
 % The value is a function from a person to gen/test relation for the age of the person

| ?? curryR(age_is)(peter)!
curryR(age_is(peter) : rel(?age)

| ?? PetersAge = curryR(age_is)(peter) & PetersAge(A).
A = 41 : age
```

5.2 General form of Function Definitions

These comprise a type declaration for a new function f, optionally immediately followed by an associated `"..."` string comment, and somewhere in the same program file a sequence of contiguous function rules for f.

Function Type Declarations

A type declaration for a k-adic function f has the form:

f: (t_1,....,t_k) -> t

where each t_i and t are type expressions.

Function Rules

These have two forms:

f(*Arg*$_1$`,...,`*Arg*$_k$`)` `->` *Exp*

f(*Arg*$_1$`,...,`*Arg*$_k$`)` `::` *SimpleConj* `->` *Exp*

where each *Arg*$_1$ is a variable, or a non-variable term which may not be ground, or a code name, of the required type t_i of f's type declaration. No *Arg*$_i$ may contain a function call.

To evaluate a function call f(*Exp*$_1$`,...,`*Exp*$_k$`)` the expressions *Exp*$_1$, ...,*Exp*$_k$ are first evaluated. Then the *first* rule for *fun* with a left hand side (a head) that matches the argument evaluated call, and which has a commit test (if present) that succeeds, determines the call's value. This is the value of the right hand side expression *Exp*, made ground by the call/head match and the evaluation of the rule's commit test. The type/mode checker ensures *Exp* will be ground.

6 Actions

Actions are the imperative subset of *QuLog*. A repertoire of primitive actions may be extended by defining new actions. When a fully compiled *QuLog* agent is started it executes an action in its initial thread, typically forking new action executing threads using the thread fork primitive. An action sequence may also be invoked by an interpreter command. They *cannot* be called from relation rules, or the commit tests of function rules.

6.1 Example action definitions

```
add_child(F,M,C): act(person, person, person)
"An action that atomically adds facts recording that C is a
 child of F and M"
% act type tag indicates an action. all arguments implicitly ! moded
add_child(F,M,C) ~>
          atomic{remember child_of(F,C) ; remember child_of(M,C)}

birthday(P): act(person)
"Add 1 to age of P if not already at recordable age maximum of 110"
birthday(P) ::  age_is(P,A) & NewA = A+1 & type(NewA,age) ~>
     replace age_is(P,A) by age_is(P,NewA)
birthday(P) ~> writeLine([P, "already has maximum recordable age 110"])
% Above rule only used if guard of first rule fails
```

We can then use action calls

```
| ?? add_child(philip,june,holly).
| ?? birthday(peter).
```

in the interpreter to record two new `child_of` facts and to increment `peter`'s recorded age by 1. The two `remember` calls of `add_child` are enclosed in an `atomic`

{...} bracketed sequence so that when one thread executes a call to `add_child` it will not be interrupted until both `remember` calls have succeeded. No other thread will be able to query the dynamic `child_of` facts until both have been added. The term argument of `remember` and both arguments of `replace...by...` must denote ground facts for some program dynamic relation when called.

QuLog action procedure types are moded with the same mode annotations as used for relation types. Like relation rules they comprise a *Head* and a *Body* but with ~> separating the two instead of <=. Also actions in the body are separated by ; rather than & to emphasise their behavioural aspect. ~> should be read as *do* and ; read is *next do*. *QuLog* actions usually only have ! and ? moded arguments. ? is to allow a terminating action to return a value to be passed to a later action. An action procedure may be non-terminating, particularly those defining agent behaviours as exemplified in (3).

6.2 General form of Action Definitions

These comprise a type declaration for a new action a, optionally immediately followed by an associated "..." string comment, and somewhere in the same program file a sequence of contiguous actions rules for a.

Action type declarations

An action type declaration has the form

a: `act(`mt_1`,...,`mt_k`)`

The moded type expressions mt_1,...,mt_k are as for a relation type declaration. Like a relation, an action may have several moded types separated by |.

Action Rules

The general forms of rules for an action a are:

a`(`Arg_1`,...,`Arg_k`)` ~> *ActSeq*
a`(`Arg_1`,...,`Arg_k`)` :: *SimpleConj* ~> *ActSeq*

where each Arg_1 is a variable, or a non-variable term which usually contains variables, or a code name of the type t_i required by r's type declaration. No Arg_i may contain a function call.

SimpleConj is a relational query as in relation and function rules. The first form of rule is shorthand for the second form where *SimpleConj* is `true`, so every rule has a commit test. Note that this means that no action call or relation query in the rule body should fail. If one does an error is signalled.
An *ActSeq* has the form

{} (the empty action sequence)
or
ActOrQuery$_1$; *ActOrQuery*$_2$; ... ; *ActOrQuery*$_n$, n$\geq$ 1,

with each *ActOrQuery*$_i$

- an *expression action* *ActExp*(*Exp*$_1$,...,*Exp*$_k$), $k \geq 0$, where each *Exp*$_i$ is an expression - a term that may contain function calls. *ActExp* is an expression returning a k-ary action *a'* such that the values of the argument expressions *Exp*$_1$,...,*Exp*$_k$ will satisfy the mode and type constraints of *a'* when it is called.
- an expression predication, or a `(...)` bracketed *SimpleConj* preceded by the operator `?` - find at most one solution to a relation query
- an action meta-call `do` *A*, *A* a variable
- an *iterated action* (an action *forall*) of the form

`forall` V$_1$`,..,`V$_j$ `(exists` *VarSeq*$_2$ *SimpleConj* `~>` `exists` *VarSeq*$_2$ *SimpleActSeq*`)` $\quad j \geq 1$

V$_1$`,..,`V$_j$ are *universally* quantified over the entire iteration. *VarSeq*$_1$ is a possibly empty sequence of different variables existentially quantified over *SimpleConj* and *VarSeq*$_2$ is a possibly empty sequence of different variables existentially quantified over *SimpleActSeq*. (If either is empty the preceding `exists` is absent.) All other named variables of the iteration must have ground values before it is evaluated. Any _ anonymous variables are implicitly existentially quantified just before the condition or action in which they appear. An existentially quantified action or action sequence should be read as *exists such that the action (or action sequence) succeeds.*

A *SimpleActSeq* is an *ActSeq* with no iterated action.

The type of the `do` action meta call is `do: act(actcall)`. `actcall` is a program dependent system type satisfied by any compound term that denotes a type and mode correct call to either a primitive or a program defined *QuLog* action. The use `do` and `actcall` is exemplified in Section 7.

7 Type and mode safe meta-level programming

The AI programming languages Prolog and LISP both allow fragments of program to constructed and manipulated and then executed but to ensure type and mode safe execution of that program fragment call specific type and mode checks (in the case in Prolog) have to be inserted in the program fragment. If the program fragment has been read in or sent in a message, each call must be examined to find the function or relation being called in order to do the required checks on its arguments. *QuLog* provides two system generated but program linked types, `relcall` and `actcall`, for both the compile time and runtime checking of any relation or action call for mode/type correctness. They access the name of the relation or action being called, access its moded type declaration, and check that the call conforms to one of its alternative moded types. Using these primitives considerably facilitates meta level programming.

Prolog has a primitive `=..` for decomposing and composing compound terms of the form `p(...)`. In *QuLog* this is done using the invertible `@..` function. `g@..[2,3]` evaluates to the compound term `g(2,3)`. `(h@..[d])@..[2,3]` evaluates to the more

complex compound term `h(d)(2,3)`. To use `@..` invertibly, as with the invertible uses of `++` and `<>`, we must use the pattern operator `=?`. The following queries illustrate its use.

```
| ?? g(2,3) =? P@..Args.
P = g : atom
Args = [2,3] : list(nat)

| ?? h(s)(2,3) =? P@..Args.
P = h(s) : term
Args = [2, 3] : list(nat)
```

A very simple version of an evaluator for a list of terms naming relation calls is:

```
evalPosCalls: rel(list(term?))
evalPosCalls([])
evalPosCalls([Call,..Calls]) <=
    type(Call,relcall) & call Call & evalPosCalls(Calls)
```

This checks each `Call` term on the list just before it is evaluated using the type test `type(Call,relcall)` to ensure that at the point of its evaluation the `Call` term will have functor the name of a relation r with arguments that conform to the moded type constraints of at least one type of r. Unlike a normal type test `type(V,TypeExp)` where `TypeExp` is unnotated or explicitly `!` moded, the `relcall` type test does not require the tested value to be ground. This is also the case when testing for the `actcall` type. A type test `type(Call,?relcall)` is a further relaxation, `Call` it does not need to be mode correct. If it is a compound term its functor must be the name of a relation and its arguments must be of correct type for one of the relation's types. But if `Call` is an unbound variable, the test will succeed. Example use is:

```
| ?? evalPosCalls([age_is(peter,A),apply(+,(A,1),APlus),<(APlus,111)]).

A = 40 : age
APlus = 41 : age
```

In the argument list of this call `age_is, apply, <` are all being uses as constructors of the type `relcall`.

Below is an analogous action evaluator of a list of action calls and `?(CallList)` single solution relation call queries.

```
doActs: act(list(term?))
doActs:([])
doActs(?(CallList),..Acts]) :: type(CallList,list(?term)) ~>
     ? evalPosCalls(CallList) ; doActs(Acts)
doActs:([Act,..Acts]) :: type(Act,actcall) ~>
     do Act; doActs(Acts)
```

Example use is:

```
| ?? doActs([?([age_is(peter,A),apply(+,(A,1),APlus),<(APus,111)]),
                          replace(age_is(peter,A),age_is(peter,APlus))]).

A = 40 : age
APlus = 41 : age
success
```

In (3) more elaborate meta interpreters are defined called by actions that are executed as non-terminating behaviours of multi-threaded agents. The agents ask questions and request actions of one another.

8 Type expressions and the sub-type relation

The allowed *QuLog* type expressions are:

`bottom` *bottom of the type lattice*
`atom nat int num string` *primitive non-structure types*
`atom_naming(`*ct*`)` *an atom naming code of type ct, sub-type of* `atom`
`atomic` *union of primitive non-structure types*
`list(`*t*`)` `set(`*t*`)` *primitive parameterised structure types*
`(`t_1`,..,`t_k`),` $k \geq 0$ *k-tuple type*
type *program defined non-parameterised type*
ptype`(`*t*`)` *program defined parameterised type*
`compound` *a term of the form c(...) with a functor c and zero or more arguments*
`compound_naming(`*ct*`)` *a compound term naming code of type ct, sub-type of* `compound`
`term_naming(`*ct*`)` *an atom or compound term naming code of type ct, sub-type of* `term`
`term` *union of all the above types*

`rel(`mt_1`,..,`mt_k`),` $k \geq 0$ *relation type*
`(`t_1`,..,`t_k`) -> `t`,` $k \geq 0$ *function type*
`act(`mt_1`,..,`mt_k`),` $k \geq 0$ *action type*
`code` *union of all relation, function and action types*
t_1 `||` t_2 `||` `...` `||` t_k`,` $k \geq 0$ *union of k types*
`typeE(`*t*`)` *a type expression naming type t*
T *a type variable*
`top` *top of the type lattice*

t and each t_i is a type expression. Each mt_i is a moded type expression. As described fully in Section 2.1, the mode is indicated by a prefix `!`, `?`, `??` or a postfix `?`, with no mode annotation being equivalent to a prefix `!`. `@` on its own stands for `@term` moded type.

`bottom` is the special enumerated type that has one data value `bottom_`, a value which may be returned when a function is undefined for some values of its arguments.

Tuples are first class in *QuLog*. This is not the case in Prolog because of the associativity of ',', also used as the associative 'and' connective. Because of this associativity, Prolog maps 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. `(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 $((t_1, ..., t_k))$. The type expression `((atom,int))->int` is for a function of one argument, that is a pair.

type is a non-parameterised program defined type. *ptype* is a program defined parameterised type with a single parameter, the only form of parameterised type allowed in *QuLog*.

`top` 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. In the *TeleoR* extension of *QuLog* there is one more code type, that of a *TeleoR* procedure, of the form `tr(`t_1`,..,`t_k`)`.

`typeE(`t`)` is the type of the type t. It allows us to pass a type expression as an argument as exemplified in Section 2..

8.1 Data sub-type relation

This is a partial specification of the sub-type relation. We complete it below by giving the sub-type relation for moded relation and moded action types, and for function types. Here each t is a type expression, each mt is a moded type expression. ct is a code type.

`bottom < nat < int < num < atomic < term < top`
`bottom < atom_naming(`ct`) < atom < atomic`
`bottom < string < atomic`
type `<` *type'* *if type, type' are defined finite types and* $values(type) \subset values(type')$
`list(`t`) < list(`t'`)` *if* t `<` t'
`set(`t`) < set(`t'`)` *if* t `<` t'
`(`t_1`,..,`t_k`) < (`t'_1`,..,`t'_k`)` *if* $\exists$ *i such that* t_i `<` t'_i `&` $\forall$ `j` $\neq$ `i,` $t_j \leq t'_j$
type `<` *type'* *if type*`::=`*te*`,` *type'*`::=`*te'*`,` *te* `<` *te'*
ptype`(`t`)` `<` *ptype*`(`t'`)` *if* t `<` t'
`bottom < list(`t`) < compound < term`
`bottom < set(`t`) < compound`
`bottom < compound_naming(`ct`) < compound`
`bottom < term_naming(`ct`) < term`
`atom_naming(`ct`) < compound_naming(`ct`) < term_naming(`ct`)`
`bottom < (`t_1`,..,`t_k`) < term`
`bottom < (`t_1`,...,`t_k`)->`t `< top`
`bottom < rel(`mt_1`,...,`mt_k`) < top`
`bottom < act(`mt_1`,...,`mt_k`) < top`

8.2 Sub-type relationships for relations, functions and actions

Suppose a relation, function or action H has an input argument C with a code type. The declared type *Type* of C is a use requirement for C, and a promise that C will not be called by H except in conformance with the use modes of *Type*, and with arguments that are no greater than the argument types of it HType. The compiler checks that all `H`'s uses of C conform to that promise. It must then ensure that any code value given as the value of C in a call of H, has at least the *the same uses* required by *CType*. *Having the same uses* is the $\leq$ (the sub-type) relationship for code types. It C is a `?` moded argument, or is the value returned by a function, the compiler also checks that any value given to C by H has type $CType'$ such that $CType' \leq CType$.

For functions we have just one *covers use* rule.

`(`t_1`,...,`t_k`)->`t $\leq$ `(`t'_1`,...,`t'_k`)->`t' *if* $t_1 \geq t'_1$ `&...&` $t_k \geq t'_k$ `&` $t \leq t'$

That is, a covering function must be able to be given for each argument a value of

type at least as great as the argument value type of the super-type function, and the sub-type function must return a value no greater than that returned by the super-type function.

For relation and action types we define the sub-type relationship with every element of the argument type tuple explicitly moded with either prefix `!` or `?`, or a postfix `?`. If an argument is mode typed as `@` it must be so mode typed in both type tuples.

$(mt_1,\ldots,mt_k) \leq (mt'_1,\ldots,mt'_k)$ *if* $mt_1 \geq mt'_1$ `&...&` $mt_k \geq mt'_k$

$mt \geq !t'$ *if* $t \geq t'$ *for test only use* $!t'$ *code of any mode m handling at least same type ok*
$?t \geq t?$ *for test/poss non-ground generate use t? ground test/generate ?t of same type ok*
$!st(mt) \geq !st(mt')$ *if* $mt \geq mt'$
When structure will be completely given with possibly non-ground components $!st(mt')$ *covering relation must allow the required test or generate use for the components* mt'
$?st(t) \geq ?st(t?)$ *when use is test/generate of a complete skeleton of a possibly non-ground structure* $?st(t?)$*, test/generate* $?st(t)$ *of same type ground structure ok*

The covering relation or action must have the same or a greater *moded* type for each of argument types of the covered relation. This $\geq$ relation depends on both the type and the mode. Intuitively it means - allows all the uses of the covered relation.

Suppose a function argument f is typed `(atom,num) -> num`. A function typed `(atomic,atomic)->int)` allows all the uses of f, so could be passed as an argument. If a relation argument r is typed `rel(!string,!int,!atom)`, a relation typed `rel(?string,?num,!atomic)` can be used in every way r will be used, so may be passed as an argument value for r.

The last two inequalities are for parameterised structure types like lists and trees, where the outer structure must be complete (! mode), or will be made complete (? mode), but values for the structure's components, such as the elements of a list or the labels of a tree, may be given as variables or non-ground terms.

As an example, if an argument of a relation has 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`. We do not have `rel(list(?int)`$\leq$`rel(list(?num))` because we do not have `?int`$\geq$`?num`.

The last rule is the covering rule for a relation or action which may be used to test, or generate, a complete skeleton structure which does not need to be ground. It tells us 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 tree of numbers.

9 Type Checking

The type checker ensures that all function calls are given ground arguments with a type no greater than their declared argument types, and that they will return a

ground value no greater than their declared value type. It checks that each relation and action call will have, for each ground input argument, a ground value of type no greater than the argument's declared type. For each other argument of type `T` it checks that the given argument is either a fresh variable which will be assigned the type `T`, or a variable known to have already been assigned a ground value of type no greater than `T`, or a variable not known to already have a ground value but which has assigned type `T`, or any non-variable term of inferred type no greater than `T`. The third case requires the possibly non-ground valued variable `V` to have exactly the declared argument type `T` as the call may be being used to test an already assigned value for `V`, which must be of

Here we informally describe the type/mode abstract interpretation using examples. The type checking rules it uses are given in the Appendix.

Consider the following type declarations and relation rule.

```
p : rel(int, ?int)
q : rel(T, ?T)
r : rel(int, ?num)
p(X, Y) <= q(X, Z) & r(Z, W) & Y = round(W)
```

where `round` is a system-defined function with type `round : num -> int`. We will look at type/mode checkingb the rule.

During checking we maintain a *program context* set of the form

$$\{Name_1 : !T_1, \ldots, Name_m : !Type_m\}$$

which lists the declared moded types for all user and system type declarations, and a *variable-type context* set of the form

$$\{Var_1 : ModedType_1, \ldots, Var_n : ModedType_n\}.$$

The variable-type context set contains the type constraints on variables produced when type checking a rule or query. For the `p` rule, its initial value is determined by applying the Relation Application Type rule (2) in the Appendix) to `p(X, Y)` : `!`**relcall**. Using the `p : !(rel(!int, ?int))`, which will be in the program context set, gives us

```
{X : !int, Y : ?int}
```

At this point we separately record the variables with `?` mode as we need to check that their mode has changed to `!` after processing the body since they must be grounded by the rule.

We now check the body of the rule, left to right one call at the time, using and updating the variable-type context using the Relation Application Type rule. If need be we use other type inference rules to reduce type constraints on compound term arguments that contain variables, to variable only constraints. If this process produces a new constraint $V : M_V\, T_V$ then:

1. if M_V is `!` and V does not appear in the variable-type context with mode `!` then fail with a mode error

2. else if V does not appear in the variable-type context then add this constraint (with the mode changed to `!` if it was `?`)
3. else if $V : !T'_V$ appears in the variable-type context then check that $T'_V \leq T_V$ (else fail with a type error)
4. else $V : M'_T\, T'_V$ appears in the variable-type context (M'_T must be `?` or `??`), check if $T'_V = T_V$ (else fail with a type error) and if M_V is `?` then replace $V : M'_T\, T'_V$ by $V : !T'_V$

The intuition for (1) is that, if a call has a variable with mode `!` (i.e. it is given) then it must have been ground earlier (i.e. it is already has a `!` mode). For (3) we already know V has a (ground) type T'_V and so it can only be used in a position that requires an equal or larger type. For (4) if V already has a (ground) type T'_V (the call is for testing) then, as for (3) we require $T'_V \leq T_V$. On the other hand, if the call is generating a value then we require $T_V \leq T'_V$, and so the types must be equal.

The above only deals with single-level modes. Multi-level modes are processed in a similar, but slightly more complex way.

Applying this step to `q(X, Z)` : !**relcall**, we get the variable-type context

```
{X:!int, Y:?int, Z:!int}
```

`T` of the polymorphic type of `q` is constrained to be no smaller than `int` (from (3) above). Because this is the only constraint on `T` then we instantiate it to `int`.

Applying the same process to the next call, `r(Z, W): !relcall`, we get

```
{X:!int, Y:?int, Z:!int, W:!num}
```

Now we check `Y = round(W)`. Using the current variable-type context, and the Function Application rule (1), the expressions on both sides have type `int` as required since `=` has moded type `??T=??T`. Since the `round(W)` is ground, we change the mode of `Y` to `!`. The final context is

```
{X:!int, Y:!int, Z:!int, W:!num}
```

Finally we must check that `Y` (since it had `?` mode in the head) has `!` mode. It has.

For the second, more complex example, we consider the following type declarations and rule. It is concocted to illustrate checking a higher order relation and sub-type filtering.

```
filter_nums: rel(list(??term) ?list(num))
gen_list: rel(int,?list(??term))
r: rel( rel(T,?list(??term)), T, ?list(num))
rel(Gen,Seed,Nums) <= Gen(Seed,GenLst) & filter_nums(GenLst,Nums)
```

The relation `filter_nums` will take a complete list of terms, some of which may be variables or compound terms containing variables, and will extract the ground sub-list of number terms unifying that with its second argument. Its definition is similar to `add_nums_on_any_list`. The relation `gen_list` takes an integer and unifies its second argument with a complete list of integers or variables, as indicated by the type expression `?[??term]` for its second argument.

`r`'s first argument is a relation of type `rel(T,?[??term])`, where `T` is the type of the ground input second argument. The relation argument takes a ground term of

type `T`, and must unify its second arguement with a complete list of `terms`, some of which may be variables or terms containing variables (the inner `??` of `?[??term]`).

Because `r` is a polymorphic relation, when we type check the rule we are not allowed to instantiate `T` or have it appear in any $\leq$ constraint as this would imply some restriction on `T`. The context set generated from processing the head is

```
{Gen:!rel(!T,?list(??term)), Seed:!T, Nums:?list(num)}
```

We put `Nums` in the variable set that must be ground at the end of body processing.

After processing `Gen(Seed,GenLst)` the context set becomes

```
{Gen:!(rel(!T,?list(??term)), Seed:!T, Nums:?list(num), GenLst:!list(??term)}
```

and, finally, after processing `filter_nums(GenLst,Nums)` the context set is

```
{Gen:!rel(!T,?list(??term)), Seed:!T, Nums:!list(num), GenLst:!rel(??term)}
```

with `Nums` with `!` mode, as required.

We finish by type checking the query call `r(gen_list, 4, N)` We start with the context set empty and end with the context set `{N : ?list(num)}` This requires satisfying `4:T` and `gen_list:(rel(!T,?list(??term)))` for some `T`. Using the rules Type Order (18), Natural Type (13), Relations (38) and Moded Type 1 (41) gives the constraints `nat` $\leq$ `T` $\leq$ `int`. Instantiating `T` to `nat` the constraint is satisfied.

10 Related Work

There are many papers (starting with (7)) treating type inference/checking for logic programming languages. We have not come across any that has *all* of: function rules, sub-types, integration of types and modes, higher-order types, run-time type checking for constraining both types and modes, type checking of meta-level programming, a clear and clean separation between behavioural and declarative programming. This brief survey of related is not exhaustive.

Mercury (13) has both types and modes. However, Mercury does not have Prolog variables as first class values, nor, it appears, does it integrate reasoning about types and modes. (9) considers types and modes for Prolog in an approach which is close to ours, but with different motivations. His language has polymorphic types, multi-level modes (e.g. complete lists of non-ground terms) and allows multiple type/mode declarations of relations. However, it does not have an unconstrained mode, nor higher order types.

Typed Prolog (12) has explicit type declarations and allows a mixture of both dynamic and compile time type checking. lambda-Prolog (8) is based on the intuitionistic higher-order theory of hereditary Harrop formulas. It has polymorphic types and unification of lambda function expressions, but no modes or sub-typing of data.

Esher (6) is essentially an extension of Haskel with logical variables and relations modelled as boolean functions that can be called with unground arguments. Curry (5) is similar, both have monadic I/O as in Haskell (10). As both treat relations as boolean functions there is no need for modes.

Go! (2) is a multi-threaded higher order logic, functional and object oriented language for multi-agent programming. Class instances have state manipulated by

action rule methods. It has data structure type definitions and type inference. It is not moded, does not have term sub-typing nor meta-programming. It has influenced our design of *QuLog*.

References

Clark, K. L. Negation as failure. In *Logic and Data Bases* (1978), J. Minker and H. Gallaire, Eds., Plenum.

Clark, K. L., and McCabe, F. G. Go! A multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence 41*, 2-4 (2004), 171–206.

Clark, K. L., and Robinson, P. J. Qulog: A relation, function and action rule language for engineering agent applications, 2014. Downloadable from: www.doc.ic.ac.uk/ klc/QLOverview.pdf.

Clark, K. L., and Robinson, P. J. Robotic agent programming in TeleoR. In *Proceedings of International Conference of Robotics and Automation* (2015), IEEE.

Hanus, M., et al. An integrated functional logic language. *Choice 33* (2003), 1.

Lloyd, J. W. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3 (1999).

Mycroft, A., and O'Keefe, R. A. A polymorphic type system for Prolog. *Artificial Intelligence 23* (1984), 295–307.

Nadathur, G., and Miller, D. An Overview of lambda-Prolog. In *Fifth International Conference Symposium on Logic Programming* (1998), MIT Press.

Naish, L. A declarative view of modes. In *Proceedings of Joint International Conference on Logic Programming* (1996), MIT Press.

Peyton-Jones, S., et al. Report on the programming language Haskell 98, 1999. At: http://research.microsoft.com/apps/pubs/default.aspx?id=67041.

Robinson, P. J., and Clark, K. L. Pedro: A publish/subscribe server using Prolog technology. *Software Practice and Experience 40*, 4 (2010), 313–329.

Schrijvers, T., et al. Towards typed prolog. In *Proceedings of the 24th International Conference on Logic Programming* (2008), pp. 693–697.

Somogyi, Z. A system of precise modes for logic programs. In *Proceedings of the Third International Conference on Logic Programming* (1986), MIT Press, pp. 469–787.

Appendix: Type Inference Rules

The following type inference rules use moded types, which is necessary for checking the modes and types of definitions. We also require checking and inference to apply to types without modes. The type inference rules can be used simply by eliding the modes from the rules. The rules below are used in two ways in *QuLog*. The first way is when doing type inference - the type is to be deduced. In this case the rules are tried in the order given. By doing this we obtain the minimal type for the term. The second use is for type checking and in that case the Type Order rule can also be used when A is atomic. The other collections of rules are used for simplification and determining minimal types.

We use the notation $Term : Type$ to mean that $Term$ is of type $Type$ and

$T_1 \leq T_2$ to mean that T_1 is either the same as T_2 or lower in the type hierarchy and $T_1 \leq_{\mathbf{m}} T_2$ to mean that T_1 is less that or equal to T_2 as moded types. We also use $T_1 < T_2$ to mean T_1 is strictly lower in the type hierarchy than T_2.

When using these rules in practice we can introduce type variables that come from polymorphic types. If we are checking that a term is of a polymorphic type we don't allow such type variables to be instantiated. On the other hand if we are finding the type of a term that contains a subterm whose type is polymorphic then we allow such type variables to be instantiated.

As an example of an instance of the Polymorphic Constructor Type rules below consider the type

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

In this case $PT(T)$ of the rule is `tree(T)`, the C_1 term is `empty()` and the C_2 term is `tr(tree(T), T, tree(T))`.

We consider the inference rules below to use pattern matching rather than unification. This means, for example, that the Equality rule (21) cannot be used to bind a type variable to a type.

Type Inference Rules

In the rules below the M's are modes.

(1) Function Application Type:
$$\frac{F : !\,(T_1 \to T_2) \quad A : !\,T_1}{F(A) \ : \ M\ T_2}$$

(2) Relation Application Type:
$$\frac{R : !\,((M_1\ T_1, \ldots, M_n\ T_n){<}{=}) \quad A_1 \ : \ M_1\ T_1 \quad \ldots \quad A_n \ : \ M_n\ T_n}{R(A_1, \ldots, A_n) : !\ \mathbf{relcall}}$$

(3) Action Application Type:
$$\frac{R : !\,((M_1\ T_1, \ldots, M_n\ T_n)\ {\sim}{>}{>}) \quad A_1 \ : \ M_1\ T_1 \quad \ldots \quad A_n \ : \ M_n\ T_n}{R(A_1, \ldots, A_n) : !\ \mathbf{actcall}}$$

(4) Polymorphic Constructor Type:
$$\frac{\begin{array}{c} PT(T) ::= (C_1(T_1^1(T), \ldots, T_{m_1}^1(T)) \mid \ldots \mid C_n(T_1^n(T), \ldots, T_{m_n}^n(T))) \\ A = C_i(A_1, \ldots, A_{m_i}) \quad A_1 \ : \ M\ T_1^i(M'\ T) \quad \ldots \quad A_{m_i} \ : \ M\ T_{m_i}^i(M'\ T) \end{array}}{\begin{array}{c} A \ : \ M\ PT(M'\ T) \\ T ::= (A_1 \mid \ldots \mid A_i \mid \ldots \mid A_n) \end{array}}$$

(5) Enumerated Type:
$$\frac{T ::= (N_1..N_2) \quad A \ : \ \mathbf{int} \quad N_1 \leq A \leq N_2}{A \ : \ M\ T}$$

(6) Integer Range Type:

(7) Empty List Type: $\dfrac{}{[] \ : \ M \ [M' \ T]}$

(8) Non-Empty List Type: $\dfrac{A \ : \ M' \ T \quad B \ : \ M \ [M' \ T]}{[A|B] \ : \ M \ [M' \ T]}$

(9) Set Type: $\dfrac{[A_1, \ldots, A_n] : ! \, [! \, T]}{\{A_1, \ldots, A_n\} : ! \, \{! \, T\}}$

(10) Tuple Type: $\dfrac{A_1 \ : \ M \ T_1 \quad \ldots \quad A_n \ : \ M \ T_n}{(A_1, \ldots, A_n) \ : \ M \ (T_1, \ldots, T_n)}$

(11) Bottom Type: $\dfrac{}{\texttt{bottom\_} : M \ \textbf{bottom}}$

(12) String Type: $\dfrac{}{A \ : \ M \ \textbf{string}}$ provided A is a *QuLog* string

(13) Natural Type: $\dfrac{}{A \ : \ M \ \textbf{nat}}$ provided A *QuLog* integer and $A \geq 0$

(14) Integer Type: $\dfrac{}{A \ : \ M \ \textbf{int}}$ provided A is a *QuLog* integer

(15) Number Type: $\dfrac{}{A \ : \ M \ \textbf{num}}$ provided A is a *QuLog* number

(16) Atom Type: $\dfrac{}{A \ : \ M \ \textbf{atom}}$ provided A is a *QuLog* atom

(17) Type Type: $\dfrac{}{T \ : \ M \ \textbf{typeE}(T)}$ provided T is a type

Type Order

(18) Type Order: $\dfrac{A \ : \ T_1 \quad T_1 \leq T_2}{A \ : \ T_2}$

Type Simplification

(19) Union Reduce:

$$\frac{A_i \leq A_j \wedge i \neq j}{(A_1 \,||\, \ldots \,||\, A_n) = T} \text{ where } T \text{ is } (A_1 \,||\, \ldots \,||\, A_n) \text{ with } A_i \text{ removed}$$

Type Ordering Rules

For the last four rules, if the moded types have multi-level modes then the inner most mode is the one used and the T's are the types with all the modes removed.

(20) Transitivity: $\dfrac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3}$

(21) Equality: $\dfrac{}{T \leq T}$

(22) Bottom: $\dfrac{}{\mathbf{bottom} \leq T}$

(23) Top: $\dfrac{}{T \leq \mathbf{top}}$

(24) Code:

$$\frac{}{(T_1 \rightarrow T_2) \leq \mathbf{code} \wedge (T_1 <=) \leq \mathbf{code} \wedge (T_1 \sim >>) \leq \mathbf{code} \wedge (T_1 \sim >) \leq \mathbf{code}}$$

(25) Term: $\dfrac{\neg(T \leq \mathbf{code}) \wedge T \neq \mathbf{typeE}(T_1)}{T \leq \mathbf{term}}$

(26) Base Types:

$$\frac{}{\mathbf{nat} < \mathbf{int} < \mathbf{num} < \mathbf{atomic} \;\wedge\; \mathbf{atom} < \mathbf{atomic} \;\wedge\; \mathbf{string} < \mathbf{atomic}}$$

(27) Lists: $\dfrac{T_1 \leq T_2}{[T_1] \leq [T_2]}$

(28) Sets: $\dfrac{T_1 \leq T_2}{\{T_1\} \leq \{T_2\}}$

(29) Tuples: $\dfrac{A_1 \leq B_1 \quad \ldots \quad A_n \leq B_n}{(A_1, \ldots, A_n) \leq (B_1, \ldots, B_n)}$

(30) Polymorphic: $\dfrac{T_1 \leq T_2}{PT(T_1) \leq PT(T_2)}$ where PT is a polymorphic type name

(31) Union Type Element: $\dfrac{}{T_i \leq (T_1 \,||\, \ldots \,||\, T_i \,||\, \ldots \,||\, T_n)}$

(32) Union Subset: $\dfrac{\forall i : (1..n)\ \exists j : (1..m)\ A_i \leq B_j}{(A_1 \,||\, \ldots \,||\, A_n) \leq (B_1 \,||\, \ldots \,||\, B_m)}$

(33) Enumerated Type Element: $\dfrac{\forall i : (1..n)\ T_i \leq \mathbf{atom}}{(T_1 \mid \ldots \mid T_n) \leq \mathbf{atom}}$

(34) Enumerated Subset: $\dfrac{\forall i : (1..n)\ \exists j : (1..m)\ A_i = B_j}{(A_1 \mid \ldots \mid A_n) \leq (B_1 \mid \ldots \mid B_m)}$

(35) Subrange: $\dfrac{N_2 \leq N_1 \wedge M_1 \leq M_2}{(N_1..M_1) \leq (N_2..M_2)}$

(36) Range: $\dfrac{}{(N_1..M_1) \leq \mathbf{int}}$

(37) Functions: $\dfrac{D_2 \leq D_1 \quad R_1 \leq R_2}{(D_1 \to R_1) \leq (D_2 \to R_2)}$

(38) Relations: $\dfrac{MT_1 \leq_{\mathbf{m}} MT'_1 \wedge \ldots \wedge MT_n \leq_{\mathbf{m}} MT'_n}{(MT_1, \ldots MT_n) <= \ \leq \ (MT'_1, \ldots MT'_n) <=}$

(39) Actions: $\dfrac{MT_1 \leq_{\mathbf{m}} MT'_1 \wedge \ldots \wedge MT_n \leq_{\mathbf{m}} MT'_n}{(MT_1, \ldots MT_n) \sim\!>> \ \leq \ (MT'_1, \ldots MT'_n) \sim\!>>}$

(40) Procedures: $\dfrac{T_1 \leq T_2}{T_1 \sim\!> \ \leq \ T_2 \sim\!>}$

(41) Moded Types 1: $\dfrac{T_2 \leq T_1}{!\, T_1 \ \leq_{\mathbf{m}} \ !\, T_2}$

(42) Moded Types 2: $\dfrac{}{?\, T \leq_{\mathbf{m}} ??\, T}$

(43) Moded Types 3: $$\frac{T_2 \leq T_1}{?\ T_1\ \leq_{\mathbf{m}}\ !\ T_2}$$

(44) Moded Types 4: $$\frac{T_2 \leq T_1}{??\ T_1\ \leq_{\mathbf{m}}\ !\ T_2}$$