QuLog: a modern logic based language for engineering agent applications

Keith Clark
Imperial College London
University of Queensland,
University of New South Wales
(joint work with Peter Robinson
University of Queensland)

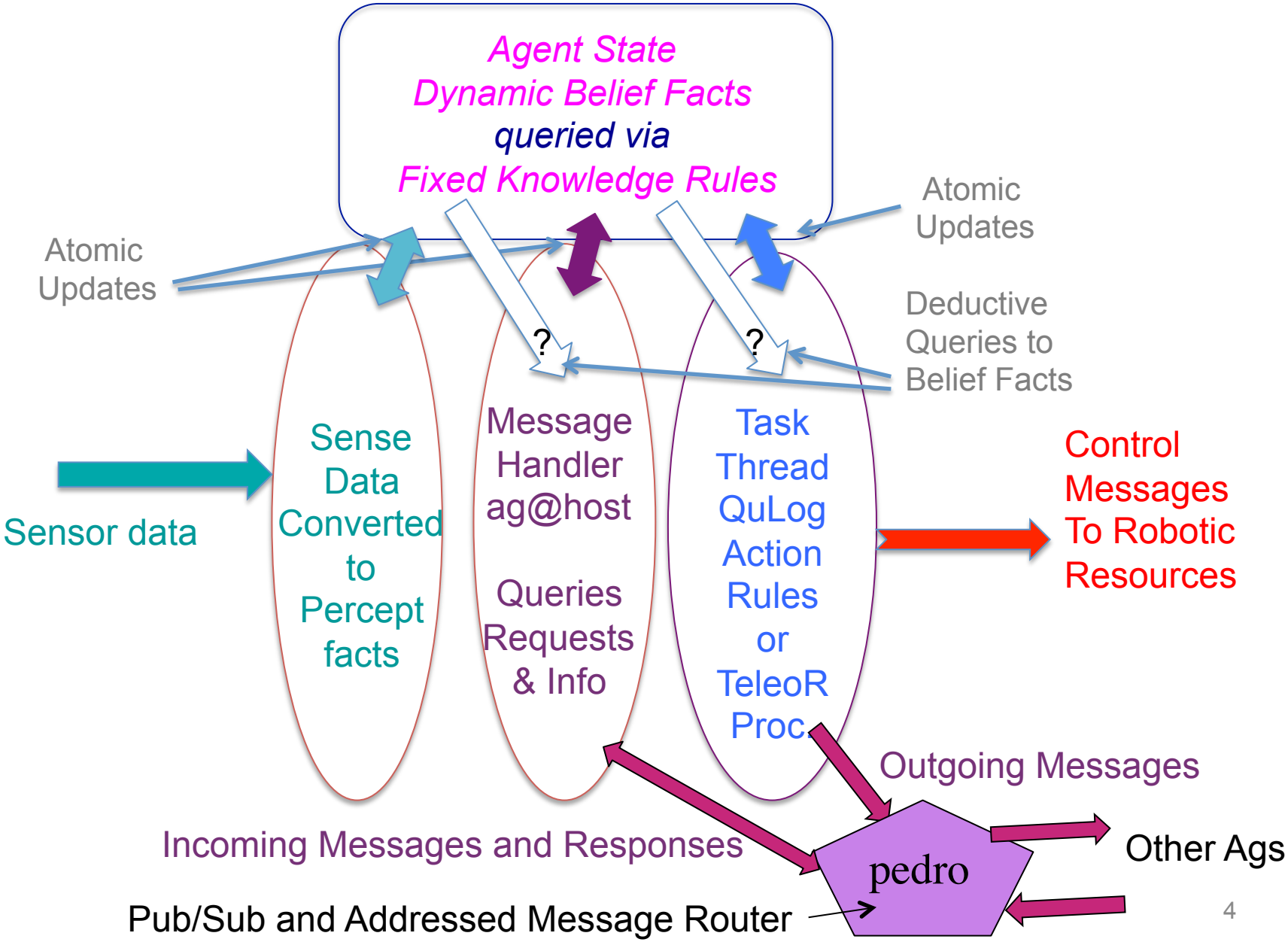
Key features of QuLog

- Flexibly typed, higher order multi-threaded declarative/imperative language with:
  - **Dynamic relations** – defined *only* by updatable facts
  - **Static Relations** – defined by sequences of logically sound conditional rules, committed choice rules, and fixed facts
  - **Functions** – defined by sequences of *committed choice* rewrite rules
  - **Expression arguments**, including *set expressions* and *list comprehension expressions* in relation, action and function calls
  - **Non-deterministic pattern matching** string + list processing
  - **Actions** – defined by sequences of *committed choice* action rules
  - **Type and mode safe meta-level programmer**
- **TeleoR** - domain specific extension for programming *multi-tasking robotic agents*
- **Agents** are named *multi-threaded QuLog processes*, each thread has a unique name within the process and a private message buffer
- **Agent thread co-ordination** via messages, suspending queries to and atomic updates of dynamic relation facts
- Activity co-ordination *across* agents *only* via messages – no shared memory
- Automatic inter-agent communication via our **Pedro** publish/subscribe and addressed message router – Msg to keith_ag@zeus.doc.ic.ac.uk

Beliefs and Knowledge

- *Dynamic facts* can be queried and updated *atomically* by any of an agent's threads
- The dynamic facts are the agent's *Beliefs*
- Relation and function rules are its *Declarative Knowledge*
- Action rules are its *Behavioural Knowledge*
- Action rules can call functions and relations but *not* vice versa

Typical QuLog Robotic Agent Architecture



Overview of tutorial

- Introduction to typed, higher order declarative subset
- Example programs, queries and watch debugging
- Brief introduction to action rule behavioural subset
- Use of action and relation rules to program a simple QuLog bottle collecting robotic agent
- Description and simulated demo of a TeleoR version of the bottle collector – TeleoR is the syntax extension of QuLog supporting robotic agent programming
- Type and mode safe meta-programming, inter-agent querying and information exchange
- Description and demo of a simulated block tower building TeleoR program that can be helped or hindered
- Multiple tower building using two arms – multi-tasking TeleoR
- Video of Baxter robot being controlled by a multi-tasking agent programmed using QuLog/TeleoR

Types

- Any set of atomic values can be a type
  age::=0..110 gender::= male | female
- Parameterized recursive constructor types as in Haskell
  tree(T)::=emp() | tr(tree(T),T,tree(T))
- Type unions allowed, run-time tests select sub-type if need be:
  atint::= atom || int
  - Primitive type atomic is union of atom nat, int, num, string
  - Strings are not lists
- Runtime type checking
  - type(V,Type) allows run-time checking of any primitive or defined Type
  - isa(A,Type) allows run-time check or generation of any finite type
    isa(A,age) gives A=0, A=1,…,A=110
- Sub-type lattice for data and code types
  - top > term > … > bottom, top > code > … > bottom
  - atom < atomic, string < atomic, nat < int < num < atomic
- *Type declarations* should be given for relations, functions and actions
  - if not given default term type assumed for all args. with ground input mode
- Sub-type values of declared argument types may be given or returned.
- *Sets* as well as *lists* with convertors between the two
  - union, inter, diff for efficient set manipulation

Mode annotations

- Modes annotate argument type expressions for relations and actions
- Prefix ! or no annotation
  - that argument in a call *must* be *ground* (fully instantiated)
  - except for belief relations, which have prefix ? mode by default
- Prefix ?
  - argument *may* also be given as a variable or partially instantiated term
  - *will be ground* after success of the call
- Postfix ?
  - argument *may* also be given as a variable or partially instantiated term
  - *may not be ground* after success of the call
- Hybrid modes for generic data types
  - list(?int) - complete list of ints and vars fully ground on success
- All function args have ground input mode
  - no annotations allowed
- Moded type constraints checked using compile time abstract interpretation.

Sub-typing for relations and actions uses modes

- (?num,!atomic)<= < (!int,!atom)<=
- (?[T])<= < ([T]?)<= T a type variable.
- *Not* (?num,?atomic)<= < (?int,?atom)<=

A relation or action sub-type SubT must allow all the uses of the super-type SuperT and accept all the argument types of SuperT

Sub-typing for functions just uses arg. and value types

(int,atomic) -> int < (nat,string) -> num

A function sub-type must accept all the arguments of the super-type and return a value no greater than that of the super-type function

Example relation definitions

```
age ::= (0 .. 110)
% age is a sub-type of nat, int, num, atomic, term
man ::= roger | tom | bill | alan | graham | keith | sam | ... % more man names
woman ::= june | mary | rose | penny | sue | helen | holly | ... % more woman names
% restricts names that can be used for men and women
human ::= man || woman % type union

| ?? isa(H,human).  % query that finds names of all humans

belief age_of(H,A): (human,age)
"Updatable facts giving an age for each human"
age_of(roger,60)
age_of(tom,26)
….
belief child_of(C,P) : (human,human)
"C is a child of P"
child_of(tom,roger)
child_of(mary,roger)
…
```

Implicitly prefix ? mode for each argument

Optional string comment that will be remembered

Separating full stops optional. All statements must begin at left end of a new line

More relation defs

Explicit existential quantification needed for A

only_has_adult_children(H): (?human)
"H has children and all are aged greater than 17"
only_has_adult_children(H) <= child_of(_,H) &
forall C (child_of(C,H) => exists A age_of(C,A) & A>17)

no_children(H): (?human) "Tests or finds an H that has no recorded child"
no_children(H) <= isa(H,human) & not child_of(_,H)

has_female_child(H):(?human) "Tests or finds an H that has at least one recorded female child"
has_female_child(H) <= child_of(C,H) & isa(C,woman)

<= annotation says second argument is a relation

only_has_children_such_that(H,Test) : (?human, (human)<=)
"H has children and all satisfy condition Test - may be used to find or test H"
only_has_children_such_that(H,Test) <= child_of(_,H) & forall C (child_of(C,H) => Test(C))

| ?? only_has_children_such_that(H,no_children) % All H's children do not have children
| ?? only_has_children_such_that(H,has_female_child)
% All H's children are themselves parents with a female child

Relations over structures

app: (list(T),list(T),?list(T)) | (?list(T),?list(T),list(T)) | (list(T)?,list(T)?,list(T)?)

“a multi-use polymorphic relation for appending or spitting lists of any type”

app([],L2,L2)

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

| ?? app([X,2],[6,8,..L1],[1,V,W,..L2]).

X = 1 : nat

L1 = L1 : list(Ty1)

V = 2 : nat

W = 6 : nat

L2 = [8,..L1] : list(Ty1 || nat)

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

on_tree : (?T,tree(T))

“a polymorphic relation for accessing the root labels on a tree of any label type”

on_tree(E,tr(_,E,_))

on_tree(E,tr(Left,_,_)) <= on_tree(E,Left)

on_tree(E,tr(_,_,Right)) <= on_tree(E,Right)

Negated and quantified tests

- not *Predication*

- All vars in *Predication* (other than _ vars) must have ground values when cond. is evaluated

  person(P) & not child_of(_,P)

- forall *AVars* (*SimpleConj1* => exists *EVars SimpleConj1*)

- All variables in *SimpleConj1 SimpleConj2* (other than _ vars and explicitly quantified vars) must have ground values when cond. is evaluated. Simple conjunctions cannot contain a forall

  forall C, A (child_of(C,mary) & age_of(C,A) & A > 30 =>
  exists Sp married(C,Sp))

Set and list comprehension expressions

{ A :: exists C age_of(C,A) & A>17}

Set of ages over 17 of recorded children, no duplicates

[A :: exists C age_of(C,A) & A>17]

List of ages over 17 of recorded children, poss. duplicates

Sets can only contain ground values

Example function defs

```
order: list(T) -> list(T)  % T is a type variable representing any type, a polymorphic function
order(L) -> []({}(L))      % {}(L) converts a list L to a set, [](S) converts a set S to a list

last_of : list(T) -> T
last_of(L) :: L =? _<> [E]  -> E
% <> a primitive function for appending ground lists, or used to split ground lists in a pattern
last_of(L) -> bottom_   % bottom_ is only value of the bottom type

oldest_person: ()->human
oldest_person() :: (_,OP) = last_of( []( {(A,P) : age_of(P,A)} )) -> OP
% The conversion of the set of pairs to an ordered list using []( .. ) needed for last_of function

children_of: human -> set( (age,human) )
"returns the set of children of a given human ordered by increasing age"
children_of(H) -> {(A,C) : child_of(C,H) & age_of(C,A)}
| ?? C in children_of(mary).
C=(5,penny)
…                          % ... is separator of the different answers
C=(7,john)
```

Currying functions and relations

curry: ((T1,T2) -> T3) -> T1 -> (T2 -> T3)

"transforms a 2 arg. function into a 1 arg. function returning a 1 arg. function"

curry(BinF)(X)(Y) -> BinF(X,Y)

curry(+)(4) % A single arg function for adding 4 to a given number

curryR: ((T1,?T2)<= -> (T1 -> (?T2)<=)

"transforms a binary relation into a function returning a unary relation"

curryR(BinR)(X)(Y) <= BinR(X,Y)

curryR(child_of)(mary) % unary relation for finding or testing mary's parents

| ?? MarysParentRel=curryR(child_of)(mary) & MarysParentRel(P).

Example action defs

```
birthday(H): (human)
"adds 1 to recorded age of H if not already 110"
birthday(H) ::
    age_of(H,A) & NewA is A+1 & isa(NewA,age) ~>>
        replace age_of(H,A) by age_of(H,NewA)
birthday(H) ~>> writeLine([H," already has maximum allowed age"])

new_child(H1,H2,H3): (human,human,human) ~>>
"records that H3 is a child of H1 and H2"
new_child(P1,P2,C) ~>>
    atomic {remember child_of(C,P1) ; remember child_of(C,P2)}
```

This right arrow used in an action rule

Condition between :: and ~>> is the rule *commit* test

Watch debugging

- No step by step query tracing
- Instead a watch can be placed on any number of relations, functions and actions
- Each time a watched def. is called
  - the call is displayed with a unique identifier
  - the def position and input and output bindings of each unifying/ matching rule being tried are displayed
  - the bindings are displayed using V_n extensions of the variable names of the program file
  - the fully instantiated call is displayed if rule use succeeds, else its failure is logged
  - any backtracking to get another solution of a relation call is similarly logged
- Optionally the instantiated rule body can be displayed
- The watched rules have hidden writes inserted

watching an app call

```
| ?? watch app.
| ?? app([X,2],[6,8,..L1],[1,V,W,..L2]).
1:app([X, 2], [6, 8,..L1], [1, V, W,..L2])
  Call 1 unifies rule 2
     input U_0 = 1 L1_0 = [2] L2_0 = [6, 8 |L1] L3_0 = [V, W |L2]   output X = 1
2:app([2], [6, 8,..L1], [V, W,..L2])
  Call 2 unifies rule 2
      input U_1 = 2 L1_1 = [] L2_1 = [6, 8 |L1] L3_1 = [W |L2]   output V = 2
3:app([], [6, 8,..L1], [W,..L2])
  Call 3 unifies rule 1
      input L2_2 = [6, 8 |L1]   output L2 = [8 |L1] W = 6
3:app([], [6, 8,..L1], [6, 8,..L1]) succeeded
2:app([2], [6, 8,..L1], [2, 6, 8,..L1]) succeeded
1:app([1, 2], [6, 8,..L1], [1, 2, 6, 8,..L1]) succeeded
X = 1 : age
L1 = L1 : list(Ty1)
V = 2 : age
W = 6 : age
L2 = [8,..L1] : list(Ty1 || age)
...
```

Seeing instantiated rule bodies

```
| ?? watchC app.
| ?? app([X,2],[6,8,..L1],[1,V,W,..L2]).
1:app([X, 2], [6, 8,..L1], [1, V, W,..L2])
  Call 1 unifies clause 2
      input U_0 = 1 L1_0 = [2] L2_0 = [6, 8 |L1] L3_0 = [V, W |L2]
      output X = 1
   Clause body is:
      app([2], [6, 8,..L1], [V, W,..L2])
.....
| ?? unwatch app.  % Turns watch debugging of app off.
```

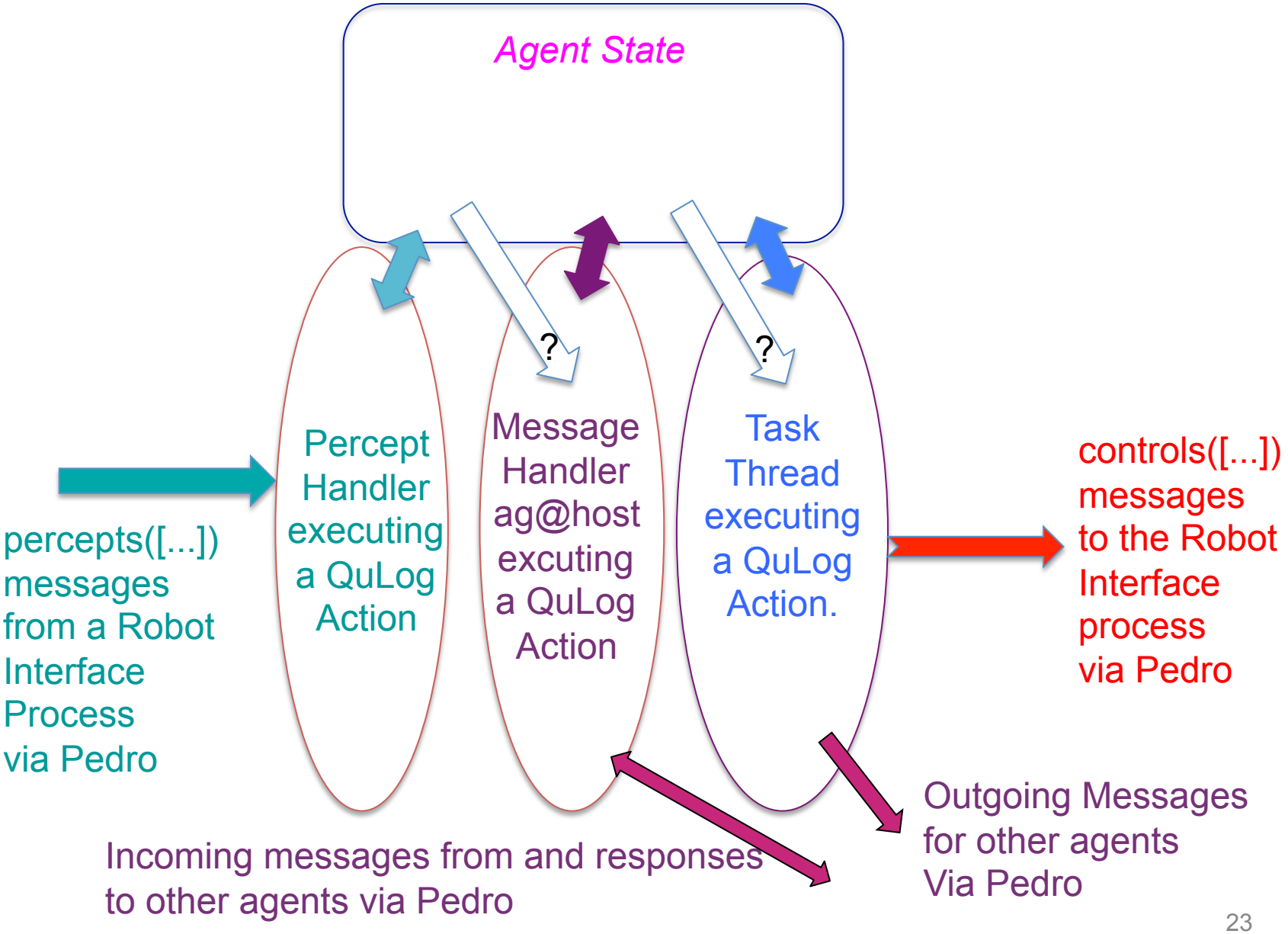
Example string processing

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

endchar : (?string)
endchar(".")
endchar("?")
endchar("!")

symbolchar : (?string)
symbolchar(C) <= sepchar(C)
symbolchar(C) <= endchar(C)
%%% No disjunction in QuLog

wordchar: string   % test use only
wordchar(S) <=  #S = 1  & not symbolchar(S)
```

String matching

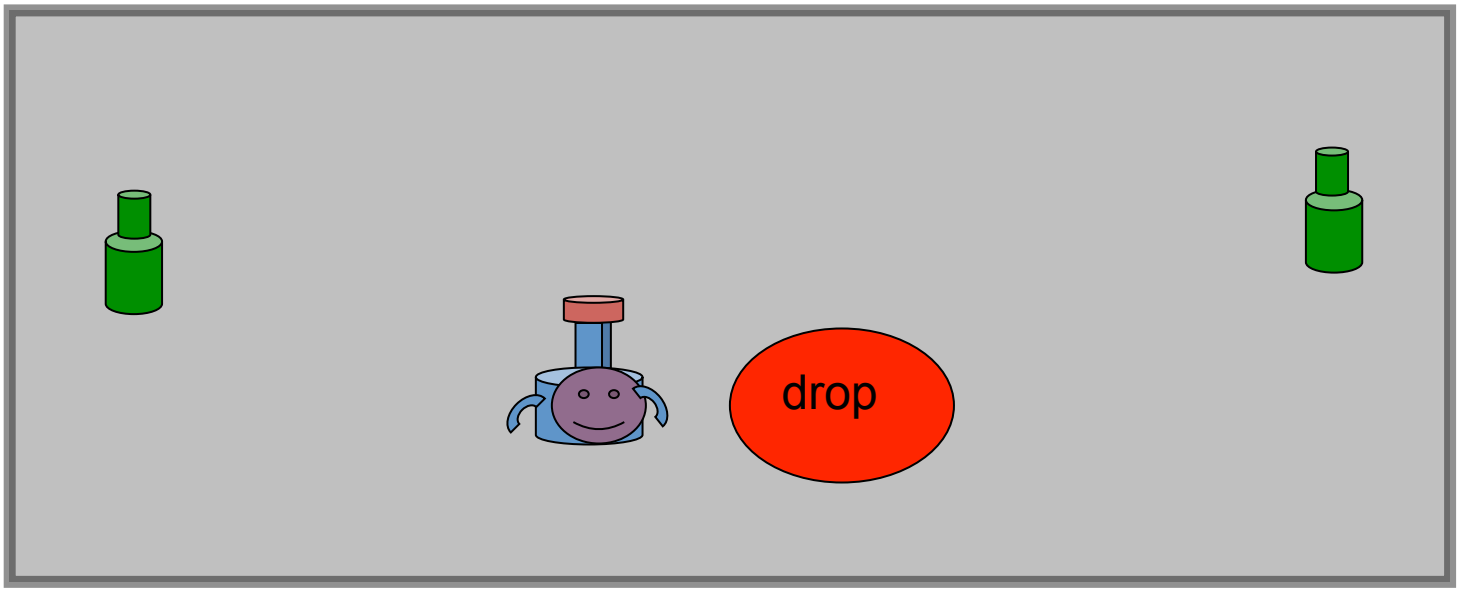
```
word: (string)
word(S) :: wordchar(S)
word(S) <= S =? C:wordchar(C) ++ RS:word(RS)

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

Words(Str,ListOfWordStrs) : (string, ?list(string))<=
“Will find or check ListOfWordStrs which is the list of word substrings of Str,
 ignoring spaces and punctuation”
words(Str,[WStr]) :: Str  =? WStr?word(WStr) ++ E?endchar(E)
words(Str,[WStr|Words]) ::
    Str =? W?word(W)++Seps?seps(Seps)++RStr?words(RStr,Words)

| ?? Ws :: words(“Hello Paolo,     how are   you today?", Ws).
Ws = [“Hello”,”Paolo”,”how”,”are”,”you”,”today”]
```

Creating a Robotic Agent Architecture



Bottle collecting robotic agent

- Enclosed space with green bottles scattered about and a drop area painted red



- No other objects in the space
- Task is to find, grab and deliver a bottle to the red drop area
- Repeat behavior when bottle removed from drop
- Automatically recover if bottle is dropped

Generic Agent Shellß

```
agent_shell(Robot, MessageHandler, ActsRel):  (handle, ()~>>, (?list(robot_act))<=)
"Robot is handle of a robot interface or simulator process,
 MessHandler is app. specific QuLog action for handling messages from other agents,
 ActsRel is app. specific relation that will query the latest beliefs to return appropriate robot actions"
agent_shell(Robot, MessageHandler, ActsRel) ~>>
      fork handle_percepts(Robot) as percepts;                  % fork a generic percepts handler thread
      fork MessageHandler() as messages;                % fork the app. specific message handler thread
      do_task(1,Robot,ActsRel)    % Execute generic task action with app. specific ActsRel as argument
```

Invoked using a query such as:

```
    agent_shell(robot@...., handle_messages, collector)
```

in interpreter entered using

```
    qulog –A bottle_collector   -N zeus.doc.ic.ac.uk
% command line options register name bottle_collector with Pedro server on zeus.doc.ic.ac.uk

robot_act ::= durative || discrete                % type union of application specific action types
robot_control ::= start_(durative) | stop_(durative) | exec_(discrete)
belief last_acts: (list(robot_act))
last_acts([])
belief updates:(nat)
updates(0)
```

Generic percept handler

```
% We assume the robot interface will send frequent percept(Ps) messages
% where Ps is a list of r_(P) and f_(P) terms, P a ground percept belief, giving
% a perceptual update.  r_(P) says remember P, f_(P) says forget P
handle_percepts: (handle)
handle_percepts(Robot) ~>>
    initialise_ to Robot;
    handle_percepts_cycle(Robot)
handle_percepts_cycle(Robot): (handle)
handle_percepts_cycle(Robot) ~>>
     percepts(Ps) from Robot  ::  type(Ps,list(term)) ;
    % suspends task until message received,
    % discarding all other messages forms whilst waiting
     atomic {                    % Do all the belief updates atomically
       replace  updates(N) by updates(N+1) ;
       forall P (r_(P) in Ps & ground_belief(P) ~>> remember P) ;
       forall P (f_(P) & ground_belief(P)  ~>>  forget P) ;
       } ;
     handle_percepts_cycle(Robot)
```

Generic task action

```
do_task:(nat, (?list(robot_act))<=)
do_task(N,ActsRel) ~>>
    wait updates(N) ;    % wait for N'th percepts update to be signalled
    atomic {
     ? ActsRel(NewActs) ;
    % The above queries the latest percept facts to determine a list of robot appropriate action
    %  responses which are used to generate action change controls, which may be empty, for the robot
     forget last_acts(LastActs));
     ? acts_to_controls(NewActs, LastActs, Controls);
     controls(Controls) to Robot ;
     remember last_acts(NewActs)
    } ;
    collector(N+1, ActsRel)    % recurse to wait for next belief store update, by percepts or messages threads

acts_to_controls: (list(robot_act), list(robot_act),?list(robot_control))
acts_to_controls(NewActs,OldActs,Controls) <=
     Controls = [stop_(RobAct) : RobAct in OldActs & type(RobAct, duractive) & not RobAct in NewActs] <>
                [start_(RobAct) : RobAct in NewActs & type(RobAct, duractive) & not RobAct in OLdActs] <>
                [exec_(RobAct) : RobAct in NewActs & type(RobAct,discrete)]
```

Collector specific declarations

```
dir ::= left | right | centre
thing ::= bottle | drop
durative ::= move(num) | turn(dir,num)
% actions that only terminate when explicitly stopped or can be modified whilst
% executing
discrete ::= close_gripper() | open_gripper()
% actions that terminate after a short time and which when cannot be modified or
% prematurely terminated whilst executing

belief see: (thing,dir,num)
belief holding: ()
belief gripper_open: ()
belief stopped: ()
```

Collector message handler

```
belief started_by: (handle)

handle_messages( ) ~>>
    receive {
        start from Ag  ~>
                    {forget stopped(); remember started_by(Ag)}
        stop from OAg :: started_by(Ag) & same_handle(Oag,Ag) ~>
                    remember stopped()
        stop from _ ~> {}
                % discard stop messages received before being started
                % or from other agents after being started
        };
     handle_messages( )
```

Top level action selection

```
collector: (list(robot_act)) <=

collector(Acts) :: stopped() <= Acts=[]
collector(Acts) :: see(drop,_,0) & see(bottle,_,0) & not holding() <= Acts=[]
% {} indicates no robot action, so terminate any previous durative actions
collector(Acts) :: see(drop,_,0) & holding() <=  Acts = [open_gripper()]
% Robot just has to release the held bottle
collector(Acts) :: holding() <=  get_to_thing(drop,Acts)
% Call aux relation to get actions eventually getting the robot to the drop holding a bottle
collector(Acts) :: see(bottle,centre,0) & gripper_open() <= Acts = [close_gripper()]
% Action should result in bottle being held
collector(Acts) :: see(bottle,Dir,0) & gripper_open() <= Acts = [turn(Dir,0.5)]
% Turn slowly to position the robot so bottle is directly in front of the gripper
collector(Acts) :: see(bottle,_,_) & gripper_open() <= get_to_thing(bottle,Acts)
% Call aux relation to get actions eventually getting the robot next to seen bottle
collector(Acts) & gripper_open()  <=  Acts = [turn(left,0,5)]
% Robot is turned on the spot to look for a bottle
collector(Acts) <= Acts = [open_gripper()]
% Default rule to open gripper before looking for a bottle
```

Auxiliary action selection

```
get_to_thing: (thing,?list(robot_act))

get_to_thing(Th,Acts) :: see(Th,_,0) <= Acts=[]
% Robot is next to Th, stop do nothing, i.e. stop any durative action

get_to_thing(Th,Acts) :: see(Th,centre,D) & D<80 <= Acts=[move(0.5)]
% Th is less then 80 units away from the robot, approach slowly

get_to_thing(Th,Acts) :: see(Th,centre,_) <=  Acts=[move(1.5)]
% Th is more then 80 units away from the robot, approach quickly

get_to_thing(Th,Acts) :: see(Th,Dir,D) & D<80 <= Acts=[move(0.5),turn(Dir,0.5)]
% Dir is left or right

get_to_thing(Th,Acts) :: see(Th,Dir,D) <= Acts=[move(1.5),turn(Dir,0.5)]
% and Th is more then 80 units away from the robot
```

Bottle collector in TeleoR

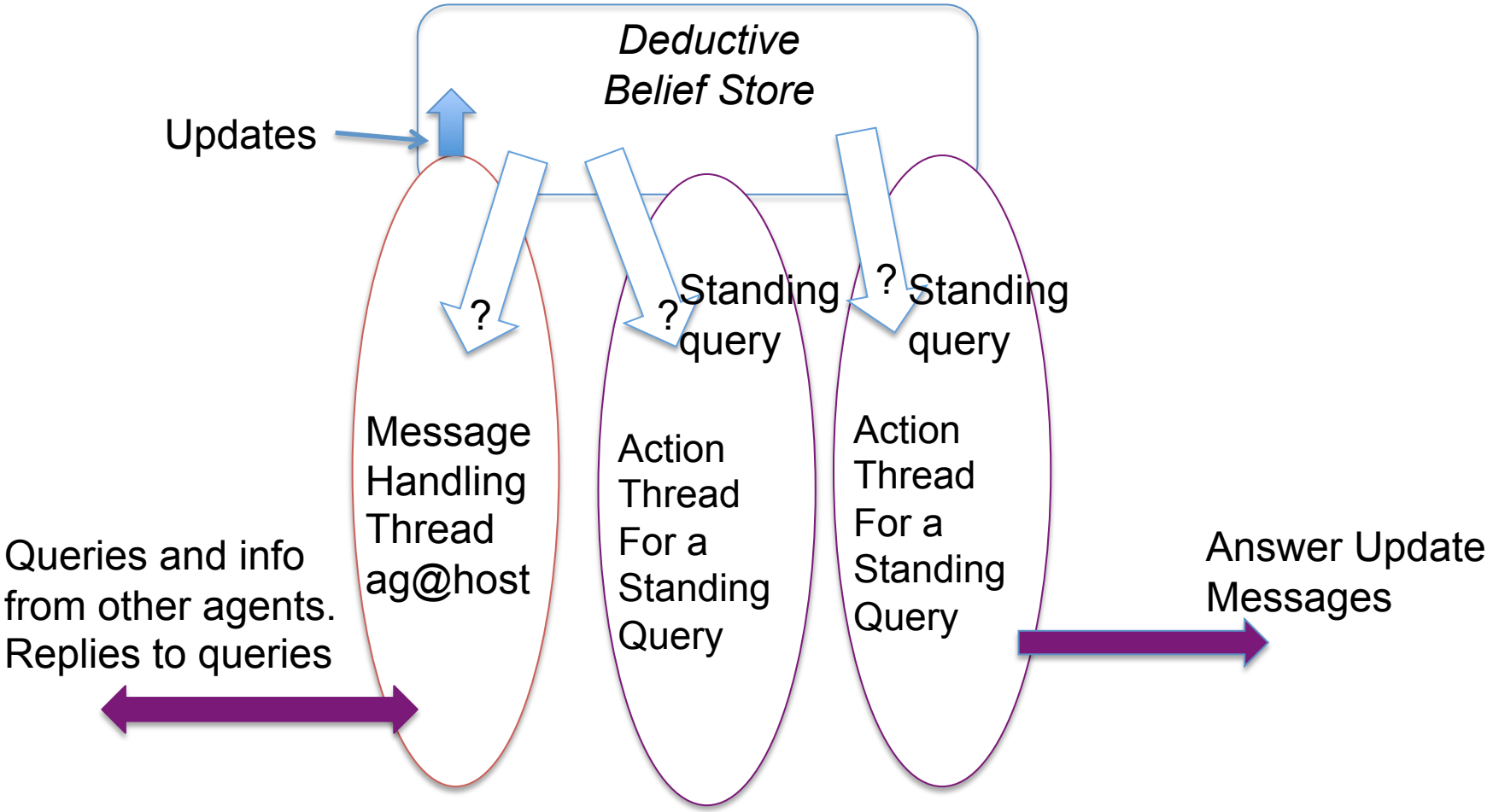
```
collector_actions_tr:()
collector_actions_tr(_) {
      stopped() ~> {}
      over_drop() & not holding() & very_close_bottle() ~> {}
      over_drop()  & holding() ~> open_gripper()
      holding() ~> get_to_thing_tr(drop)
      see(bottle,0,centre) & gripper_open() ~> close_gripper()
      see(bottle,0,Dir) & gripper_open() ~> turn(Dir,0.2)
      see(bottle,_,_) & gripper_open() ~> get_to_thing_tr(bottle)
      gripper_open() ~> turn(left,0.4)
      true  ~> open_gripper()   }
get_to_thing_tr: (thing)
get_to_thing_tr(Th){
   see(Th,D,centre) & D < 80 ~> move(0.8)
   see(Th,D,centre) ~> move(1.5)
   see(Th,D,Dir) & D< 80  ~> move(0.8), turn(Dir,0.2)
   see(Th,D,Dir) ~> move(1.5),turn(Dir,0.2)
   true ~> turn(left,1.0) }
```

Support for meta-level programming

- System generated program dependent primitives
  - relcall(r(..)) r(..) is a type and mode correct call to a primitive

    or program defined relation, or a belief. Modecorrect if all the arguments of the term r(..) that are input moded arguments of the relation r are ground. relcall accesses program and system relation declaration.
  - actcall(a(..)) ditto for actions
  - belief(B) B is a term denoting a type correct call to a belief relation
  - ground_belief(B) B is a term denoting a belief fact
- call relcall?
  - call C converts relcal compound term r(..) l binding of variable C to a call to relation r and evaluates it using r's definition, may not ground r(..)
- do actcall?
  - do C similar for its a(..) actcall binding of variable C
- Both meta-calls must be preceded by a runtime relcall(C) or actcall(C) test to ensure the meta-call arg denotes a type and mode correct correct relation or action call
- CTerm =? P@ArgList @ as pattern decomposing compound terms
- P@ArgList @ as function for constructing compound terms

Information agent



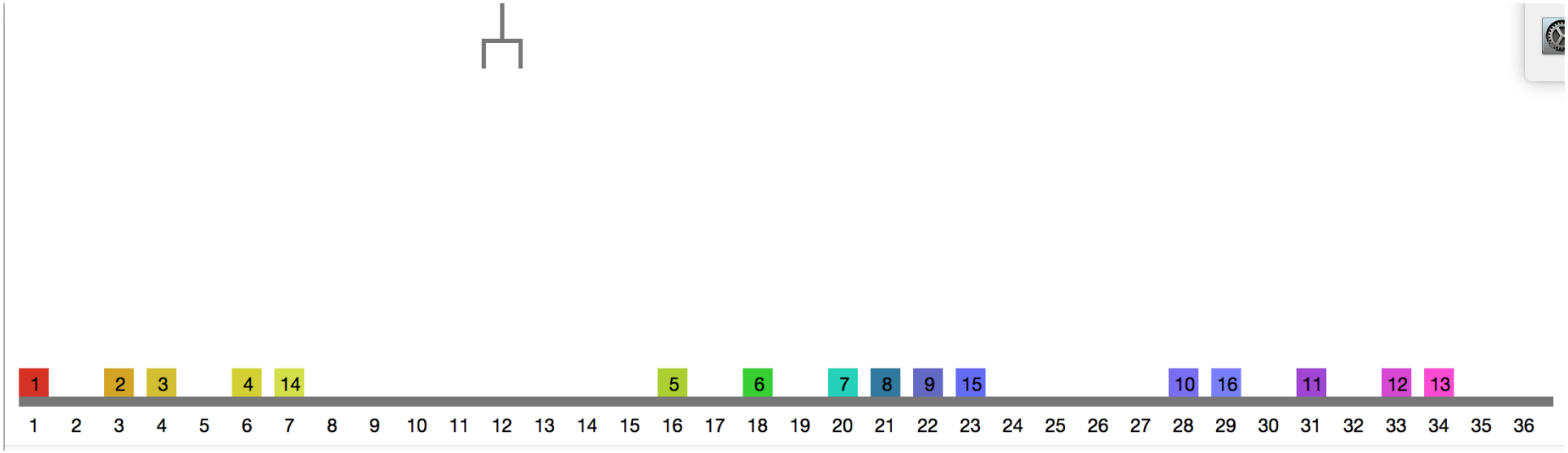
Simple message handling info agent

```
belief can_tell: (handle)
can_tell(keithsAg@...)
...                  %  agents that are allowed to give information
belief can_ask: (handle)
can_ask(keithsAg@...)
...                  %  agents that are allowed to ask queries
belief ....,...,...  % all the other fact forms that can be stored and queried
handle_messages() ~>>
    receive
      { tell(Bel) from Sndr ::
            can_tell(Ag) & same_handle(Sndr,Ag) & ground_belief(Bel) ~>
                                remember Bel
        tell(not(Bel)) from Sndr :: can_tell(Ag) & same_handle(Sndr,Ag) & belief(Bel) ~>
                                forget Bel
        ask(Ans,Qlist,Qid) from Ag ::
            can_ask(Ag) & type(Qlist,list(term)) & type(Qid,atom) ~>
                                answer_query(Ans,Qlist,Ag)
        M from Ag :: not can_tell(Ag) & not can_ask(Ag) ~> ignored(M) to Ag
        M from Ag ~> not_allowed_message_form(M) to Ag
      };
    handle_messages()
```

Meta-interpreter for query lists

```
answer_query: (atom, term?, list(term?), handle)

answer_query(Qid,Ans,[],Ag)  ~>>
                        reply(Qid,Ans) to Ag

answer_query(Qid,Ans,[Cond,..Conds],Ag) :: relcall(Cond) & call Cond ~>>
                        answer_query(Qid,Ans,Conds,Ag)

answer_query(Qid,_, [Cond,..],Ag) ~>>
                        failed_or_invalid_cond(Qid,Cond) to Ag
```

Robust tower building



Belief Store for tower building

```
block ::= "1" | "2" | "3" | "4" |  "5" | "6" | "7" | "8" | "9" | "10" | "11" |  ...
slot ::= 1 .. 36
place ::= block || slot

durative pickup : (slot,block),  putdown : (slot, block)

percept                    % TeleoR synonym for belief
        holding : (block),
        on : (block, block),
        in_slot: (block,slot)

clear: place
clear(Slot) :: type(Slot,slot) <= not in_slot(_, Slot)
clear(B) <= type(B,block) & not on(_,B)
```

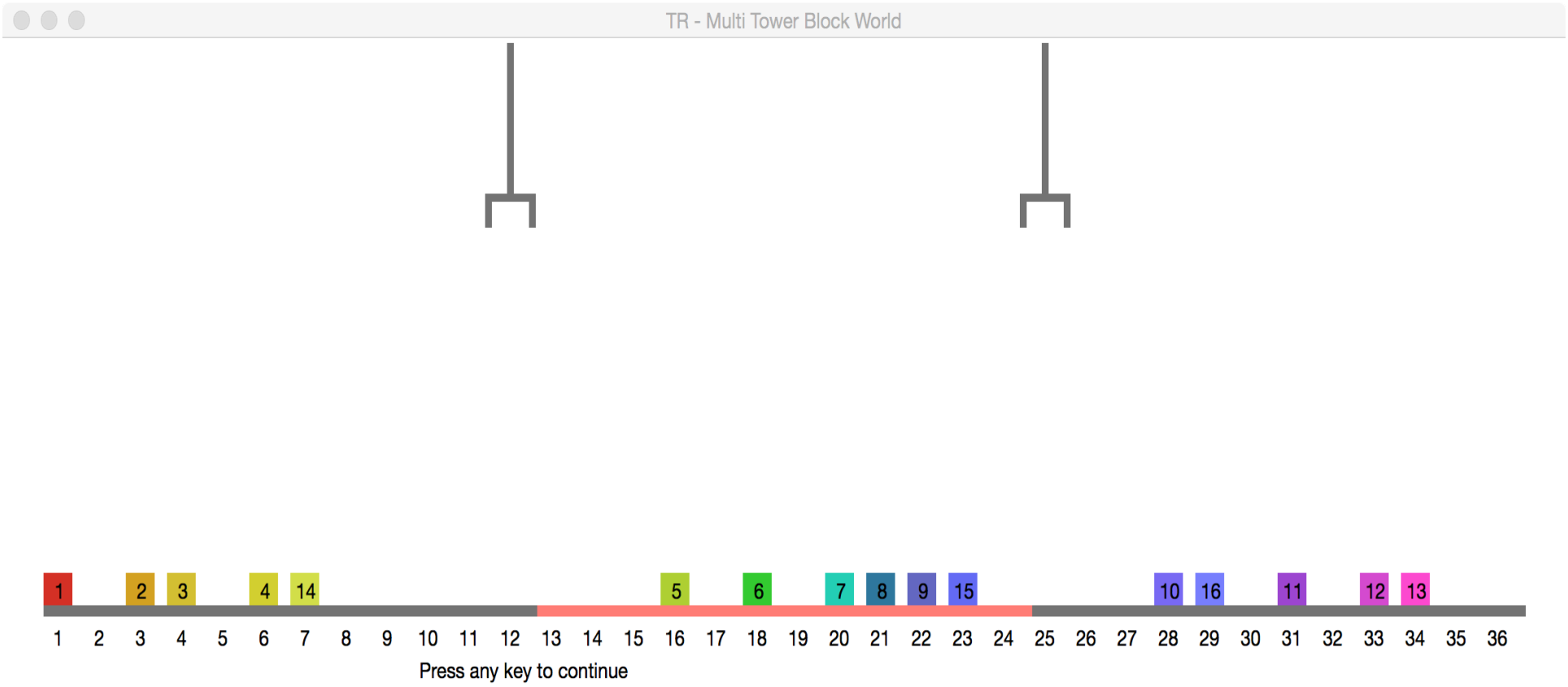
More relation defs

```
clear: place
clear(Slot) :: type(Slot,slot) <= not in_slot(_, Slot)
clear(B) <= type(B,block) & not on(_,B)

on_place: (?block,?place)
on_place(Bl,Pl) <= type(Pl,slot) & in_slot(Bl, Pl)
on_place(Bl,Pl) <= type(Pl,block) & on(Bl, Pl)

above: (?block,?place)
above(B, Pl) <= on_place(B,Pl)
above(B, Pl) <= on_place(OthrB, Pl) & above(B,OthrB)

...   % Other relation defs

tower: (list(block),?slot)
tower([B,..Bs],Slot) <= clear(B) & stack([B,..Bs],Slot)

stack:(list(block),?slot)
stack([B], Slot) :: in_slot(B, Slot)
stack([B1,B2|Bs],Slot) :: on(B1, B2) & stack([B2|Bs],Slot)
```

Top level TeleoR procedure

```
makeTower: (list(block), slot)
makeTower(Bs,Slot){

  tower(Bs,Slot)  ~> ()

  stack(Bs,Slot) & Bs=[B,..] ~>  make_clear_block(B, Slot)

  Bs=[B,B1,..Bs1] & tower([B1,..Bs1],Slot)  ~> move(B, Slot)

  Bs=[B] & clear(Slot) ~> move(B, Slot)

  Bs = [B]  ~> make_clear_slot(Slot)

  Bs=[B,..RestOfBs] ~> makeTower(RestOfBs, Slot)
  }
```

Multi-resources tower building

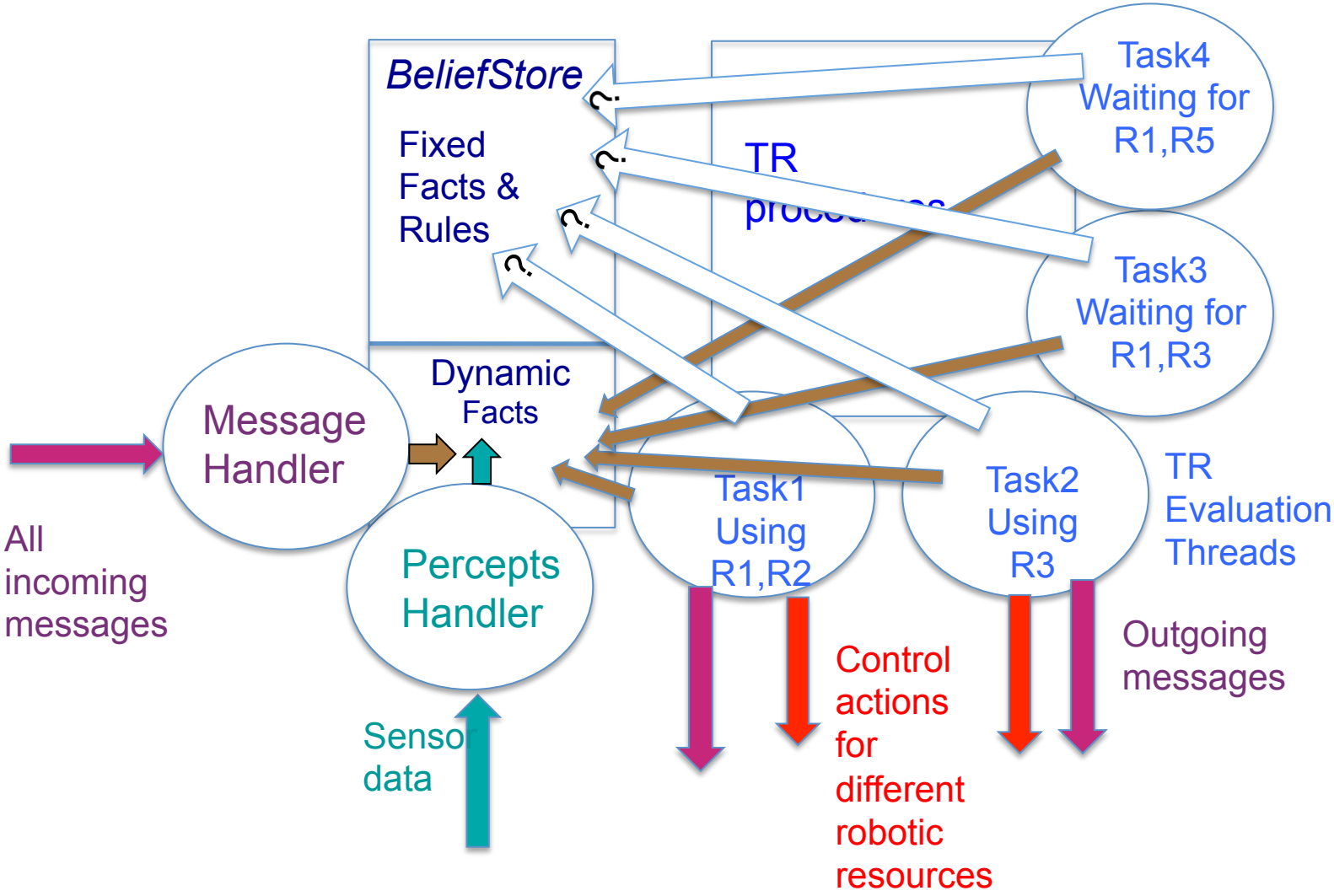


arm1 only

both arms

arm2 only

Multi-tasking architecture



Baxter two arm control

QuLog system

- Downloadable from:

  http://staff.itee.uq.edu.au/pjr/HomePages/QulogHome.html

  New release in December 2015

- Two books available late next year:

  QuLog: Engineering Agent Applications,
  Clark & Robinson, Springer

  Programming Robotic Agents: *a Multi-tasking Teleo-Reactive Approach,*
  Clark & Robinson, Springer

  First 7 chapters of TeleoR book downloadable from
  http://teleoreactiveprograms.net

Informal syntax of declarative rules

- **Relation facts**, variable free: $p(arg_1,..,arg_k)$
- **Relation rules**

$p(ptn_1,..,ptn_k)$ :: *Conj* <= C_1 & C_2 & … & C_n n≥0

Universally quantified with respect to all variables in rule *head.*
No function calls in any head arguments $ptn_1,..,ptn_k$

Each C_i is a:
relation call: $RExp(exp_1,..,exp_k)$
negated test: not $RExp(exp_1,..,exp_k)$
quantified test: forall *Vars* ($SimpleConj_1$ => exists *EVars* $SimpleConj_2$)
single solution condition: once $RExp(exp_1,..,exp_k)$

RExp is an expression returning a k-adic relation of correct type and use mode for the values of:
$exp_1,..,exp_k$

SimpleConj is a conjunction of conditions with no forall

- F**unction rules** are committed choice even if no explicit *Conj commit test*:

$f(argptn_1,..,argptn_k)$ :: *Conj -> Exp*

Informal syntax of action rules

Action rules

a(*ptn*$_1$,..,*ptn*$_k$) :: *SimpleConj* ~>> A_1 ; A_2 ; … ; A_n n≥0

Each A_i is a:

call *AExp*(*exp*$_1$,..,*exp*$_k$), *AExp* is *action valued expression* of correct type and mode

thread fork: fork *AExp*(*exp*$_1$,..,*exp*$_k$) as *Name*

suspending relation call:

wait *RExp*(…) watch *UpdateEvents* for *Secs* timeout *Action*

non-blocking message send: *MessExp* to *Handle*

blocking message receive discarding other messages: *MessPtn* from *HandlePtn*

choice message receive

receive {*MessPtn*$_1$ from *HandlePtn*$_1$ :: *Test*$_1$ ~> A_1

| …

| *MessPtn*$_n$ from *HandlePtn*$_n$:: *Test*$_n$ ~> A_n

| timeout T ~> A }

fact update actions, e.g. remember Fact replace FactPtn by Fact

iterated action:

forall Vars (*SimpleConj* ~>> exists EVars *SimpleActSeq*)

atomic action sequence: atomic {A_1; .. ; A_k}

SimpleActSeq is action sequence with no iterated actions