K. L. Clark
Dept. of Computing, Imperial College London
P. J. Robinson
School of ITEE, The University of Queensland

Programming Co-operating Robotic Agents:
A Teleo-Reactive Rule Based Approach

November 30, 2016

Springer
## Contents

1 **Introduction** ......................................................... 1  
  1.1 A simple agent architecture ................................. 2  
  1.2 Overview of **TeleoR** and **QuLog** .................... 3  
  1.3 Example **TeleoR** Procedure ............................ 3  
  1.4 The Evolution of **TeleoR**+**QuLog** ................. 5  
  1.5 Task Atomic Procedures for Multi-tasking ........... 7  
  1.6 Software Linked with the Book ......................... 8  

2 **Programming in Core **TeleoR** .......................... 9  
  2.1 A Bottle Collecting Robotic Agent ....................... 10  
    2.1.1 Percepts ............................................. 10  
    2.1.2 Robot actions ...................................... 11  
  2.2 The bottle collector’s **TeleoR** control program ..... 12  
  2.3 How the Program Behaves ................................ 14  
    2.3.1 Recovering from Setbacks ......................... 15  
    2.3.2 Repeating the bottle collection ................. 15  
  2.4 Universal Procedures and Regression .................. 16  
    2.4.1 Opportunism and robustness of universal procedures .... 17  
    2.4.2 Regression methodology for developing a **TeleoR** program ........................................ 17  
  2.5 **QuLog** for Defined Relations and Functions ........ 17  
    2.5.1 Using a speed function to slow the robot down ...... 20  
  2.6 Core **TeleoR** Procedure Syntax ....................... 21  

3 **Introduction to **QuLog** .................................... 23  
  3.1 Getting started ............................................. 24  
  3.2 Defining and Testing Rule Defined Relations .......... 27  
  3.3 Displaying the Program .................................. 29  
  3.4 Building Upon Tested Definitions ............... 30  
    3.4.1 Actions to update dynamic relations ............ 33  
  3.5 Query Evaluation and Backtracking .................. 33
Preface

Goal Directed Reactive Robotic Agents

This book is about programming robotic agents in a particular rule based language combination, TeleoR+QuLog. Such a robotic agent has one or more robotic resources that it can control using a repertoire of robotic actions to bring about changes in an environment. It does this to achieve the goal of some task by progressively achieving sub-goals of that goal. The task goal and sub-goals are future states of the environment that the agent can detect.

The agent may have several tasks. In that case it concurrently tries to achieve each task goal but in such a way that no task will undo an achieved sub-goal of another task. It will use its robotic resources in parallel whenever possible, but otherwise will fairly alternate the use of resources between tasks.

The environment may also be changed by exogenous events outside the control of the agent. These exogenous events may help or hinder the agent. If helped in some task the agent will skip sub-goals of the task. If hindered it will re-achieve sub-goals, or try to achieve the task goal by another sub-goal route.

To monitor its progress towards each task goal the agent gets feedback via perceptual data which are agent specific interpretations of sensor readings and camera images. These arrive frequently as a set of percept facts, resulting in frequent updates of the agent’s Belief Store. The Belief Store contains other dynamic beliefs that the agent wants to remember about what it has seen or done in the past, or has been told by another agent. It can also contain fixed facts about the environment and robotic resources and knowledge rules for querying the beliefs and facts.

The robotic agent may be working co-operatively with humans or other robotic agents. It may be able to perceive what its team mates have done, or it may need to be told what they have or are about to do. Such told information can affect its behaviour just as much as new perceptions.

After each new batch of percept facts arrive, or after it has update its Belief Store due to a communication, the agent reconsiders its last determined
actions. The re-consideration involves querying its updated Belief Store using the knowledge rules to determine the best action response for each task for its new view of the environment state. The agent should have an action response for each task for every environment state that can be described using its dynamic beliefs, even if that response is one or more actions for robotic resources for which it must wait to gain control.

**TeleoR**

TeleoR is a major extension of Nilsson’s T-R [63] Teleo-Reactive language for controlling robotic resources. Parameterised procedures in both languages comprises sequences of guarded action rules. Rule actions comprise one or more robotic resource actions, or a procedure call. Each guard is a QuLog query to the agent’s Belief Store which may use QuLog knowledge rules. Typically the rule guards can be mapped into a sub-goal tree routed at the guard of the first rule, the goal of calling the procedure. A task executes a procedure call the goal of which is, or implies, the task goal.

**QuLog**

QuLog is a flexibly typed logic and functional programming language with an imperative action rule layer on top of the declarative core. QuLog is the product of many years experience using and teaching Prolog and functional programming languages. It is more concise and more declarative than Prolog. It is also higher order in the functional programming sense. QuLog primitive actions enable the updating of the dynamic beliefs of an agent and communication with other agents. Declarative QuLog is a TeleoR agent’s Belief Store language. QuLog action definitions are used to configure the agent shell within which TeleoR tasks are executed.

**Mid-level control**

The programs one can write in TeleoR+QuLog are mid-level cognitive control programs. They are cognitive in that they use logical inference to determine actions. They are mid-level in that they assume that lower level programs to interpret camera images to provide percepts, and to implement actions, are coded in a lower level language such as C. An example of a perception program is one that will analyse a camera image and report that a bottle of known size lies in a particular relative direction, and is at a particular distance from a mobile robot with an onboard camera. An example of an action program is one that will maintain a given forward speed and direction for a wheeled mobile robot. The coding of such programs is outside the scope of this book.
TeleoR versus T-R

We experimented with teaching and using the Teleo-Reactive paradigm using a Prolog implementation of the T-R language as described in [63] for a decade or more. Before that we designed and implemented several logic based languages for programming communicating software agent applications, as well as interactive theorem provers.

Based on that experience, we progressively extended our Prolog implementation of T-R to an implementation of the TeleoR+QuLog language combination. We simultaneously developed the formal operational semantics of TeleoR, with useful ‘symbiosis’ between the two activities.

The differences between TeleoR and T-R are as follows:

- TeleoR is typed using the same flexible type system as QuLog. This allows compile time guarantees that the robotic resource actions will be correctly typed and fully instantiated before being dispatched.
- TeleoR uses the QuLog language for encoding knowledge rules to query the agent’s Belief Store. This is much more expressive than the simple rule language used for T-R.
- The use of QuLog attached actions enables an agent to update the non-percept beliefs in its Belief Store as it dispatches its robot resource actions. It also allows implementation of communicating robotic agent applications.
- TeleoR has extra forms of rules and actions giving more fine grained task control. Each extension enables the programming of applications that cannot be programmed, or are less transparently programmed in T-R.
- Using TeleoR’s task atomic procedure concept, one can readily program multi-tasking agents dynamically sharing multiple robotic resources, without deadlock and without starvation of any task, with each task occasionally helping but never hindering other tasks.
- TeleoR has a formally specified operational semantics.

Structure of the book

We introduce the TeleoR language in stages with each new feature motivated by a robotic agent programming example. The progressive introduction is interleaved with a progressive development of its operational semantics. We also give an example driven introduction to the main features of QuLog in Chapter 3, before it is used in earnest.

In Chapters 2 and 4 we present and illustrate the use of core TeleoR, which is very close to Nilsson's T-R language. In Chapter 5 we give the operational semantics for core TeleoR.

In Chapter 6 we present and exemplify the use of TeleoR’s forms of rules and actions not in the core language or T-R. In Chapter 7 we update the operational semantics of Chapter 5 to cover these extensions.

In Chapters 8 and 9 we describe and exemplify TeleoR’s high-level features for programming multi-tasking agents dynamically sharing one or more resources.
robotic resource, giving some details of how they are implemented. In Chapter 10 we give the operational semantics for the evaluation of each of a set of concurrent resource sharing tasks.

In Chapter 11 we give a brief history of the T-R language as we understand it. We then survey other rule based robotic agent languages and architectures, beginning with those using the Teleo-Reactive paradigm. We finish with pointers to work on formal methodologies for developing and verifying Teleo-Reactive programs.

In Chapter 12 we conclude with ideas for extensions of the TeleoR language and supporting agent architecture.

Linked software

There is free open source software linked with the book downloadable from [70]. It can be installed on any Linux/Unix or Windows based device, or under OS X. All the example programs of the book, or close variants, are included, each paired with an interactive Python graphical simulator. You can run the programs within the supplied agent shell, helping or hindering the agent via the simulator.

To use with any robotic hardware one interface process is needed written in C/C++, Java or Python. It communicates with the TeleoR agent via our Pedro [71] communications server, also used for inter-agent communication. A Python template for a ROS [67] interface process is provided.

On an Android phone located on the robot a TeleoR agent has been used to control a Lego Mindstorms robot with simple image processing using the phone’s camera software. It has also been used as the top level control of a Baxter robot engaged in two concurrent configuration tasks with dynamic allocation of the arm resources to the tasks. The arms are used in parallel whenever possible. That application has a ROS interface.

Pre-requisites

The book assumes familiarity with the notation and semantics of predicate logic for the semantics chapters. The first four chapters of each of the two volumes of the free electronic book [76] provide sufficient background, although we use a slightly different logic syntax.

Familiarity with logic programming concepts or Prolog would be useful but not essential. However, [54] is an excellent introduction to using declarative Prolog for computational ‘thinking’.

Acknowledgements

The first acknowledgement has to go to Nils Nilsson. Without his novel and elegant concept of Teleo-Reactive procedures, and particularly his triple-tower architecture paper [63] which added inference from a Belief Store, this book
would not have been written. We much appreciate his positive comments about our work and encouragement to complete the book.

Secondly, we acknowledge the support and contributions of former colleagues Ian Hayes and Brijesh Dongol, with whom we had very useful discussions regarding new rule forms and their operational semantics.

For their comments on draft chapters of the book we thank: Pedro Sanchez Palma and colleagues, John Staples, Kostas Stathis, Adam Stacy, Maarten van Emden, Robert Kowalski, Krysia Broda, Andrew Davison of PSU Thailand, Naranker Dulay, Marek Sergot and Frank McCabe.
Chapter 1
Introduction
1.1 A simple agent architecture

As mentioned in the Preface this book step by step introduces the programming of multi-task communicating robotic agents in two rule based programming languages, TeleoR and QuLog. We start with the programming of a single task non-communicating agent which has two threads as depicted in Figure 1.1.

![Fig. 1.1 Two Thread Agent Architecture](image)

The role of the *Percepts Handler* thread is to wait for batches of sensor data from sensors and cameras external to the agent. The data comes as a list of *percept* facts. On receipt of each list it does an atomic update of the agent’s *Belief Store*, replacing old percept facts with the new facts. The percept updates may be upwards of 10 times a second.

The *TeleoR Evaluator* thread evaluates a call to a TeleoR procedure. It is the task thread of the agent. Each time the *Belief Store* is updated it queries the *Belief Store*, possibly using *knowledge rules* expressed in QuLog, to determine the best action response to the new perceptual data. This may result in a change of control message sent to one or more robotic resources external to the agent. The evaluator thread then suspends until there is another update of the *Belief Store*.

The purpose of the action responses is to bring about changes in the environment of the robotic resources, which may include the location of any mobile resource, until the task goal is achieved. The goal will be detected by a query to some future *Belief Store* state.
1.2 Overview of TeleoR and QuLog

A TeleoR program comprises a set of typed parametered procedures. Each procedure comprises a sequence of guarded action rules the simplest of which has the form

\[ K \rightarrow \lambda \]

\( K \) is a QuLog query to the agent’s Belief Store which may access the percepts via QuLog knowledge rules. \( A \) is either a tuple of actions for the robotic resources, to be executed in parallel, or it is a TeleoR procedure call, which can be a recursive call.

The TeleoR language is a major extension of Nilsson’s Teleo-Reactive robotic agent language T-R [63]. *Teleo* means to bring to an end or to achieve a goal. In the case of T-R and TeleoR it is the goal of a task executing a procedure call. The query that detects this goal state is the guard of the first rule of this procedure call, which has the *do nothing action* \((\cdot)\) - the empty tuple of actions.

The rule guards of each procedure typically form a sub-goal tree routed at the guard of its first rule - the goal of the procedure. The programmer thinks in terms of goals and sub-goals to develop the program. The sub-goals on each branch of the sub-goal tree are possible stages on route to the procedure’s goal.

Often, initially called TeleoR procedures have rule guards that use inference to query the percept facts and have actions that call TeleoR procedures as sub-tasks. These sub-task procedures then typically use shallower inference. Eventually procedures are called that directly query the percept facts. So in TeleoR the interface between reason based and sensor reactive behaviour is a sequence of intermediary procedure calls.

QuLog is a flexibly typed multi-threaded logic and functional programming language with an imperative layer of action rules. Action rules can call relations and functions but not vice versa. The logic programming subset of QuLog is more declarative than Prolog with a syntax closer to that of predicate logic. Function call expressions, list and set comprehension expressions can be used as arguments to relation calls, and relation queries can be used as guards of function defining rules. QuLog action calls can be used to update facts for dynamic relations, do I/O, fork threads executing action calls, and send messages to other threads and processes.

1.3 Example TeleoR Procedure

A Teleo-Reactive program offers a simple and intuitive way of both programming and specifying robust, opportunistic, goal directed robotic agent
behaviour. To see this goal directed robust behaviour, download Nilsson’s Java application simulating a robot arm building a tower of blocks [64]. You can help or hinder by moving blocks around as it is building a block tower, which you give it as a task goal. The agent will immediately respond appropriately. In particular you can co-operatively build the block tower with the arm agent if you choose. Teleo-Reactive programs are well suited to programming robotic agents for human/robotic resource co-operative work.

We will give you a flavour of the TeleoR and T-R languages by giving the simplest procedure of a program that we fully discuss in Chapter 2. The robotic agent has just one task executing an initial TeleoR procedure call with a goal to collect and deliver a bottle to a drop area. It achieves this goal using sub-tasks that: get next to a seen bottle, get hold of it by closing its open gripper, deliver it by getting next to the drop area, then releasing the bottle by opening the gripper.

Below is the general purpose TeleoR approach procedure comprising just two action rules for getting the bottle collecting robot to approach something Th at a certain forward speed Fs, with a correctional rate of turn Ts if it wanders off target. It is used where Th is bottle or drop, where drop is a delivery area for a collected bottle. Th, Fs, Ts, Dir are variables representing any value of the correct type. We follow the Prolog convention and use alphanumeric strings beginning with an upper case letter or _ as variables.

```
def thing ::= bottle | drop
% Enumerative type definition for thing
tel approach(thing,num,num)
% Type declaration for approach TeleoR procedure
approach(Th,Fs,Ts)
{
  see(Th,centre) ~> move(Fs)           % ~> may be read as do
  see(Th,Dir)  ~> move(Fs),turn(Dir,Ts)
}
```

Suppose the procedure is invoked by a call approach(bottle,2.5,0.3) by the bottle collecting task, because a bottle has been sighted. Its rules become

```
see(bottle,centre) ~> move(2.5)
see(bottle,Dir)  ~> (move(2.5),turn(Dir,0.3))
```

Action rule guards of a procedure call are always checked starting at the first rule. The first rule found with a guard query to the Belief Store that succeeds - is inferable generating bindings for any variables - becomes the call’s fired rule instance and the rule’s instantiated action is executed. If the latest percept about the seen bottle is see(bottle,centre) indicating it is
seen more or less in the centre of the camera image, the first rule is fired. Its action is to have the robot move forward at a speed of $2.5$. It is a *durative* action that will continue until stopped or modified.

Suppose that a change of camera image causes \texttt{see(bottle,centre)} to be replaced by \texttt{see(bottle,left)} in the *Belief Store* because the robot has deviated slightly to the right. As soon as that happens, the second rule will be fired.

This will continue the move forward but with a parallel slight turn to the left. The combination of the continuing forward move and the turn should quickly re-achieve the guard of the rule above - resulting in a new percept \texttt{see(bottle,centre)} being received replacing \texttt{see(bottle,left)}. As soon as that happens the first rule will be fired again, to have the robot approach more or less in a straight line once more.

The above \texttt{approach} call was the result of firing of the middle rule of

\begin{verbatim}
  near(bottle,_) ~> approach(bottle,1.5,0.5)
  see(bottle,_) ~> approach(bottle,2.5,0.3)
  true ~> turn(left,0.5)
\end{verbatim}

which are the last three rules of the parent call \texttt{get_next_to(bottle)}. The `\_' underscore means any direction, we do not need to know which. The last rule is the default rule of the calling procedure that is used if no bottle is in view, or if it the bottle is moved out of view when the robot is approaching it. If that happens the call to \texttt{approach} is terminated.

That is another peculiarity of \texttt{T-R} and \texttt{TeleoR} procedures. The calling procedure remains *active* as do all the ancestor procedure calls. When an ancestor call fires a different rule, this causes termination of the latest procedure call as well as any intermediary procedure calls. There is no *return* or *exit* action in a \texttt{Teleo-Reactive} procedure.

If the default rule in the parent call is fired, the \texttt{move} action of the robot will be terminated and a \texttt{turn} to the left action started, or increased in speed if the robot was already turning to the left having previously fired the second \texttt{approach} rule with \texttt{Dir=left}. The robot is now made to turn to the left, on the spot, at a speed of 0.5. One of the two rules above the default rule will be fired when a bottle comes back into view. They invoke the \texttt{approach} procedure with different speeds depending upon how far away the seen bottle is.

### 1.4 The Evolution of \texttt{TeleoR+QuLog}

We have been experimenting with teaching and using the \texttt{Teleo-Reactive} paradigm for a decade or more. As a result of our test applications, we progressively extended the \texttt{T-R} language of \cite{63} to produce the \texttt{TeleoR+QuLog}
combination. For single task programming we extended our initial implementation of T-R to have:

- Typed declarative QuLog as the guard query language. Typed robotic actions and typed TeleoR procedures.
- The optional attaching of QuLog actions to any TeleoR action rule.
- Actions which are a sequence of time limited durative actions or time limited procedure calls.
- $K \text{ while } C \rightarrow A$ rules that give an alternative $C$ to the rule guard $K$ for use after the rule is fired. Assuming no earlier rule is fired, it allows the rule’s action to continue should the rule guard $K$ be no longer inferable from an updated Belief Store, providing $C$ is inferable.
- $K \text{ until } U \rightarrow A$ rules that inhibit re-firing of the rule and the firing of earlier rules of a procedure call until $U$ is inferable from the Belief Store, providing $K$ remains inferable.
- $K \text{ while } C \text{ until } U \rightarrow A$ rules with a combination of the control aspects of while and until rules.

Using declarative QuLog as the Belief Store query language means there is be no runtime failure of a guard evaluation due to a type error. It also means a compile time guarantee that all actions are fully instantiated and correctly typed when a rule is fired. The ability to attach QuLog actions to rules allows us to program multi-agent co-operative applications and agents that can remember key things about past actions or perceptions. The inter-agent communication uses our Pedro communications server [71]. Adding timed action sequence enables us to program behaviours where the switch from one action to another is a time lapsed condition rather than a Belief Store query. An example is wandering behaviour. The last three allow us to have more refined control over task behaviour.

We tested all of the extensions with an application involving two identical communicating robotic agents with a given joint goal, co-ordinating their actions using communication as well as sense data from a mobile robot resource, which each separately controls. Each agent keeps the other informed regarding progress towards the joint goal. They also communicate to avoid collision with minimal divergence from their current paths. The communication is used to overcome poor perception as only the presence of the other robot but not its direction of travel can be determined. This application is fully explained at the end of Chapter 6.

As we were extending our initial implementation of T-R we were developing the formal operational semantics. This turned out to be an essential concurrent activity as we found it necessary to alternatively revise both the language extensions, their semantics and their implementation.

One fundamental revision was regarding the semantics of until rules. They had been proposed by a colleague at the University of Queensland, Ian Hayes, as a useful concept for the transparent programming of a safety critical application test case. He had proposed a semantics that we could not
see how to implement efficiently. That led to weakening of the semantics and
the adoption of a rule form with a dual semantics, the \texttt{while} rule.

Defining the semantics also considerably helped with issues regarding opti-
mising the implementation. It gave us a clear description of the behaviour we
needed to optimise. The activity also exposed an ambiguity in the informal
operational semantics for T-R given by Nilsson in [63], which we discuss in
Chapter 5.

There are echoes in the \texttt{until} and \texttt{while} rules of TeleoR of the behaviour
inhibition concept of Brook's Subsumption Behaviours [11], [12], as both
forms of rules temporarily inhibit the firing of other rules of the procedure
call. However our motivation for adding them to TeleoR was to enable the
programming of certain behaviours using behaviour transparent programs
with a cleanly defined semantics.

1.5 Task Atomic Procedures for Multi-tasking

Our most radical extension of T-R is the concept of \texttt{task},\texttt{atomic} procedures
and the related concept of \texttt{task},\texttt{start} procedures. This was to allow multi-
tasking by our TeleoR robotic agents with the tasks alternating the use of
one or more robotic resources.

We started by investigating multi-tasking where the tasks had compatible
goals and must share a single robotic resource. This was tested by changing
Nilsson's block tower building program of [63], reproduced in Chapter 4, so
that it could safely interleave the use of one robotic arm to allow the building
of several towers comprising different blocks. The change required was the
adding of two \texttt{task},\texttt{atomic} procedure statements and one \texttt{task},\texttt{start} procedure
statement. Declaring a procedure as \texttt{task},\texttt{atomic} results in a quite different
compilation of the procedure.

The natural generalisation was to allow multi-task programming using
multiple robotic resources with dynamic allocation of subsets of the resources
to tasks. This required us to add a robotic resource use co-ordination policy
to the operational semantics. Our goal was to allow as much parallel use of the
robotic resources as possible, consistent with a 'fair' allocation of resources
so that there was no starvation of any task of the robotic resources it needed,
and there was no possibility of deadlock.

We tested this multi-tasking architecture by modifying Nilsson's tower
building program so that it could be used by an agent interleaving the build-
ing of several block towers using two robot arms. The blocks are distributed
over three tables with each arm only able to reach two tables, a home table
and a shared table. The arms and the tables are the agent's resources.

Short videos showing the two arm controlling agent in action, one using
a simulation in Python the other running on a Baxter robot, are accessible
from [18].

7
1.6 Software Linked with the Book

The entire software package for developing and running multi-threaded TeleoR agents comprises the following open source software:

- The QuLog interpreter that can be used to develop, test and debug the knowledge rules for an agent’s Belief Store as described in Chapter 3.
- The TeleoR extension of the QuLog interpreter which has a multi-threaded configurable agent shell and supports the incremental development, testing and debugging of TeleoR programs as described in Chapter 4.
- The multi-threaded Qu-Prolog system [17] on top of which QuLog and TeleoR are currently implemented.
- Example TeleoR+QuLog programs with linked Python simulations including all the programs of this book.
- The Pedro communications server [71] and its documentation. Pedro is used for inter-agent communication and to link the Python simulators with the agent shell.
- HTML and PDF documentation files for QuLog+TeleoR.

All are downloadable via [70].
Chapter 2
Programming in Core TeleoR
In this chapter we present our first TeleoR program, a bottle collecting program controlling one mobile robot. The program is written in core TeleoR, a subset of TeleoR that more or less corresponds to Nilsson’s T-R language. We discuss the behaviour of this first control program in some detail so that you will get a good understanding of the elastic properties of TeleoR procedures, and their very different behaviour compared with called procedures in other programming languages.

We also introduce Nilsson’s regression and completeness properties for his T-R procedures, and their relationship to program correctness. These properties also apply to TeleoR procedures. We finish with an introduction to the use of QuLog relation definitions to define program specific relations in terms of quite low level percept facts.

2.1 A Bottle Collecting Robotic Agent

A mobile robot is to be used to find, get hold of, and deliver an empty bottle of known size and colour to a drop area of known but different colour. The bottle and the drop area are on the floor. They have colours different from walls and other distant things that the robot might see. We assume that the space around the drop area in which bottles will be found is obstacle free.

The robot is able to independently turn as well as move forwards. It has a forward facing camera with simple image processing software. It has a gripper with sensors that may be used to grasp a bottle that the robot is next to, and to then keep hold of the bottle while it is delivered. The grasping may not always succeed, and the bottle may slip out of the gripper. Figure 2.1 depicts the environment.

2.1.1 Percepts

Assuming a fixed size for the bottles and only one drop area, with simple image processing the controlling robotic agent can receive camera percepts:

\[
\text{see(Th,Dir) near(Th,Dir) next_to(Th,Dir) close_to(Th,Dir)}
\]

where Th is bottle or drop, and Dir is left, right or centre, sent to the percept handling thread of the controlling robotic agent. These alternative symbol values for Th and Dir are called atoms.

The difference between the four percept relations is determined by the size of the area of Th’s colour in the camera image, and the direction is determined by the position of the centre line of a bounding rectangle of the coloured area in the camera image. centre is used when this centre line is positioned within +10 and - 10 degrees from the forward pointing direction of
the camera and robot. \texttt{left} is used when it is positioned somewhere between -10 and -25 degrees of this forward direction, and \texttt{right} when it is somewhere between 10 and 25 degrees.

We have controlled a Lego Mindstorms robot using an onboard Android mobile phone communicating with the Mindstorms brick by Bluetooth. The phone had software that computed the bounding rectangle of a blob of colour in the phone’s camera image, which we used to compute distance and relative direction of fixed size objects with distinct colours.

By calibration we assume \texttt{near(Th,Dir)} is a generated percept from the camera image processing software only if \texttt{Th} is within a metre. This should be generated in addition to \texttt{see(Th,Dir)} which also holds. \texttt{close_to(Th,Dir)} is generated when the robot is within 20 cms of \texttt{Th}, in addition to \texttt{see(Th,Dir)}. \texttt{next_to(bottle,Dir)} is generated only if the robot is within bottle grasping distance. \texttt{next_to(drop,Dir)} is generated only if the robot is at the drop area.

Using pressure sensors on the gripper, the agent receives percepts:

\begin{verbatim}
gripper_open() holding()
\end{verbatim}

\subsection{2.1.2 Robot actions}

The robot has durative actions:
move(Fs) turn(Dir,Ts)

where Fs and Ts are a forward speed and a turning speed respectively. It has discrete (aka ballistic) actions:

open_gripper() close_gripper()

with names that tell us what they do. If move and turn are executed together the effect is to move the robot in an arc swerving in the Dir direction. These actions may be used by the robotic agent to control the mobile robot. In addition there is the do nothing action ()

Durative actions are actions that may be stopped and modified whilst executing. For example, with move the speed argument can be changed, modifying the forward speed of the robot and our robot can be told to stop the move action.

Discrete actions are actions that cannot be modified or terminated when started, and which naturally terminate after a short time. Whether an action is classed as durative or discrete is device specific. If our robot was such that the gripper actions could be stopped they would to be declared as durative.

We assume that there is a bottle within visual distance of the robot when its controlling TeleoR robotic agent starts executing the collect_bottle procedure given below. Before this happens the agent’s Belief Store will have been updated with the latest percepts from the robot. Its task is to find and deliver the bottle to the drop area. The goal of the task will have been achieved when the robot is next to the drop, next to a bottle, with its gripper open, so that the bottle may be picked up by a human or another agent controlling a robot arm and placed in a bin.

2.2 The bottle collector’s TeleoR control program

```python
def dir::= left | centre | right % Definition of enumerated type
def thing::= bottle | drop
percept holding(), gripper_open(), next_to(thing,dir),
    close_to(thing,dir), near(thing,dir), see(thing,dir)
def discrete::= open_gripper() | close_gripper()
def durative::= move(num) | turn(dir,num)
```
tel collect_bottle()
collect_bottle(){
    next_to(drop,_) & next_to(bottle,_) & gripper_open() ~> ()
    % Goal holds, do nothing

    next_to(drop,_) & holding() ~> open_gripper()
    % holding() sub-goal holds, get the robot next to the drop

    holding() ~> get_next_to(drop)

    next_to(bottle,centre) & gripper_open() ~> close_gripper()

    gripper_open() ~> get_next_to(bottle)

    true ~> open_gripper()  % Open gripper if need be
}
tel get_next_to(thing)
get_next_to(Th){% To be used when Th is bottle or drop
    next_to(Th,centre) ~> ()  % Goal holds

    next_to(Th,Dir) ~> turn(Dir,0.3)
    % Turn slowly to get centre view, if need be

    close_to(Th,_) ~> approach(Th,0.5,0.5)
    % Very near to Th, approach very slowly

    near(Th,_) ~> approach(Th,1.5,0.5)
    % near(Th,_) holds, approach slowly to achieve close_to(Th,Dir)

    see(Th,_) ~> approach(Th,2.5,0.3)
    % Only see(Th,_) holds, approach Th quickly to achieve near(Th,_) 

    true ~> turn(left,0.5) % Th not in sight, turn hoping to see it
}
tel approach(thing,num,num)
approach(Th,Fs,Ts){% Only active whilst see(Th,_) holds
    see(Th,centre) ~> move(Fs)
    % whilst see(Th,centre), move forward

    see(Th,Dir) ~> move(Fs),turn(Dir,Ts)
    % else swerve in Dir direction to bring Th back into centre view
}
Each new statement of the program and each new procedure definition begins at the left end of a new line. Text beginning at the left end of the line is what signals the termination of the previous statement so no separator is needed between the statements and procedures. To continue a statement over several lines all but the first line are indented one or more spaces or tabs. The rules for each procedure are sandwiched between {...} braces, always indented.

Notice the type definitions and declarations at the beginning of the program. The TeleoR + QLog type system will be covered in more detail in Chapter 3. For now suffice it to say that for this program they constrain dir values to left, centre, right and thing values to bottle, drop. They also tell us the types of the arguments of the percept relations, of the procedures get_next_to, approach, and of the durative actions move and turn. The percepts declaration is both information for the percept handling thread and a declaration that the percept relations are dynamic relations.

2.3 How the Program Behaves

You may be thinking that as a conditional plan the above rules are back to front. However, by starting with a rule with a guard that is the goal of the collect_bottle procedure, and having the remaining rules have guards that are sub-goals of this goal, we achieve one of the elastic properties of a TeleoR procedure. The TeleoR evaluator skips actions whenever it can. The rule guards are tested in the order given and as soon as one is inferable its action is executed. The program is optimistic.

If the robot starts with its grippers closed the last rule collect_bottle() will be fired to open the grippers. Then, the first rule that may be fireable is its fourth rule should the robot happen to be next to and facing a bottle. Most likely this is not the case and the last but one rule will be fired. This rule invokes the procedure get_next_to with the call get_next_to(bottle).

When get_next_to(bottle) is called the procedure will evaluate its now partly instantiated rule guards (Th replaced by bottle), against the latest state of the Belief Store. Condition next_to(bottle,centre) of the first rule will not be inferable, else the rule of collect_bottle() that called the procedure would not have been fired. Most likely one of the guards of the third, fourth or fifth rules will be the first rule guard to be inferable if a bottle is in view. Depending on how far away the seen bottle is, the robot will approach it very slowly, slowly or quickly. It switches to slower forward speeds as it gets nearer.

The last default rule is used when no bottle is in sight. It has the robot turn to the left looking for a bottle. On our assumption there is a bottle somewhere within view, depending on how far away the seen bottle is, the third, fourth or fifth rule of the call get_next_to(bottle) will soon be fired.
After approaching the seen bottle for a while next_to(bottle, Dir), for some Dir, should eventually be received as a percept. If Dir is not centre the second rule of get_next_to(bottle) will be fired causing the robot to turn slowly in the Dir direction in order to get the bottle into centre view. As soon as Dir=centre the collect_bottle() call will now fire its fourth rule to close the grippers and the call to get_next_to will no longer be active.

Let us suppose that the attempt to get hold of the bottle is successful, and the holding() percept is received from the gripper touch sensors interface on the next percepts update. The third rule of collect_bottle() will be fired invoking get_next_to again with argument drop. Assuming this is successful, the percept next_to(drop, centre) will eventually be received causing the second rule of collect_bottle() to fire, terminating the get_next_to call again. open_gripper() will be executed allowing the first rule of collect_bottle() to fire when the percepts

gripper_open(), next_to(drop, centre), next_to(bottle, centre)

are in the Belief Store. The goal of collect_bottle() has now been achieved.

2.3.1 Recovering from Setbacks

get_next_to(drop) will remain active only while holding() remains inferable - while the holding() percept is in the Belief Store. Its goal is to deliver the held bottle to the drop area. Should the robot accidentally drop the bottle on route to the drop area, the next percepts update will not include holding(). collect_bottle() will now fire its last rule to open the grippers terminating the call to get_next_to(drop) and stopping the robot. The dropped bottle will probably still be in view and close, so a new call to get_next_to(bottle) on the firing of the fifth rule of collect_bottle(), when the open_gripper() percept is received, should have less to do.

2.3.2 Repeating the bottle collection

When it has released the bottle in the drop area the robot will sit there looking at the delivered bottle doing nothing. That is what the empty () action means. If a human removes the bottle, the goal of the collect_bottle procedure is no longer achieved and the procedure will begin to execute actions again, looking for, getting hold of and delivering another bottle if there is one within sight as the robot turns around at the drop. If there are no other bottles it will just keep turning to the left, having fired the default rule of
get_next_to(bottle), hoping that a bottle will eventually be put down on the floor and be in sight.

2.4 Universal Procedures and Regression

All the above procedures satisfy what Nilsson calls the regression property. The first rule of each procedure $P$ has a guard $K_1$ and an action $A_1$ such that if this rule is the fired rule of some call $P'$ of $P$ the execution of the firing instance of the action $A_1$ will normally maintain the firing instance of the guard $K_1$. Each inferable instance of $K_1$ for the call $P'$ is a goal of the call. Usually $A_1$ is the do-nothing action $()$.

Every other rule $K_j \rightarrow A_j$ of $P$ is such that if this rule is the fired rule of a call $P'$ of $P$ then the execution of the firing instance of the action $A_j$ will normally result in an earlier rule $K_i \rightarrow A_i, i < j$ becoming the next fired rule of the call $P'$ when or before the firing instance of the guard $K_i$ is no longer inferable. $K_j$ is the regression of $K_i$ through $A_j$. This is the goal seeking behaviour of the procedure. The guards of the rules of a procedure satisfying the regression property form a sub-goal tree rooted at the guard of the first rule of the procedure. It is a key property of T-R and TeleoR procedures.

If

    next_to(bottle,centre) & gripper_open()

is inferable and close_gripper() is executed then holding() should soon become inferable, Similarly,

    next_to(drop,_) & holding()

should eventually become inferable when get_next_to(drop) is called when and while holding() holds. Showing that each procedure satisfies the regression property is an informal proof of partial correctness of the program. Whenever a rule is fired some progress should be made towards a goal of the initial procedure call, which is a task goal.

Whether we will always be able to make progress is another matter. It requires that the procedures satisfy another condition - always having a rule that can be fired in the context that the procedure is called. For all the procedures except approach this is trivially the case as all are have default last rules with guard true. The approach(Th,..) procedure always has a rule that can be fired on the assumption that while it is active the robot can see Th. This will be the case when called from the three rules in get_next_to as we have assumed the see percept is received if near or close_to is received. So this is an informal argument that providing: a bottle will be in view when the robot bottle collector starts, the robotic agent is not purposely and repeatedly thwarted by having the bottle moved, and the robotic actions do not fail, a bottle will be collected and delivered to the drop area.
Procedures satisfying the regression property and such that there is always a rule that can be fired are universal procedures for the goals of the calls of the procedure. There is always a rule that can be fired and its action should always make progress towards a procedure call goal.

2.4.1 Opportunism and robustness of universal procedures

Universal TeleoR and T-R procedures immediately respond to opportunities skipping actions that are not needed by jumping up the guard sub-goal tree. They also automatically recover from setbacks redoing actions when needed by dropping down the guard sub-goal tree. The procedure `get_next_to(Th)` will automatically jump to the highest rule with an inferable condition after the `turn` action. If, when the robot is approaching a bottle, some other agent (say a human) moves the bottle to be near to the robot, it will immediately switch to executing the `approach` procedure with a slower move speed. Conversely, if that other agent then moves the bottle to be out of sight of the robot, the procedure will automatically drop to firing the last default rule to turn to look for a bottle, should no other bottle be in camera view. This is in order to re-achieve `see(bottle,_)`, `near(bottle,_)` or `close_to(bottle,_)`.

2.4.2 Regression methodology for developing a TeleoR program

As Nilsson says on page 8 of [62]

The backward-from-the-goal approach to writing T-R programs makes them relatively easy to write and understand, as experience has shown.

2.5 QuLog for Defined Relations and Functions

The guards of the TeleoR rules are not restricted to percepts. They can use relations defined by a set of facts and simple rules in our typed logic based language QuLog.

For example, the relations `next_to`, `close_to`, `near` and `see` can either be sensor percept relations, or defined in terms of a more low level sensor percept relation `see_patch(Col,Size,Dir)`. This gives the colour `Col` of a dense patch of pixels in the camera image, a measure of its size, `Size`, and
the relative direction $\text{Dir}$, which is left, right or centre, of the patch in the image on the forward facing camera. The higher level concepts would then be defined by implication rules such as those below.

By having the colour argument in the size facts we can have bottles of different colours being different sizes. Blue bottles are slightly smaller than green bottles.
def col ::= green | blue | red
% new enumerated type of the possible patch colours
percept see_patch(colour,num,dir), holding(), gripper_open()

rel next_to(?thing,?dir), close_to(?thing,?dir),
   near(?thing,?dir), see(?thing,?dir)
% Type declarations for rule defined relations

next_to(Th,Dir) <=
   see_patch(Col,Size,Dir) &
   next_to_size(Th,Col,NtoSize) & Size>=NtoSize

close_to(Th,Dir) <=
   see_patch(Col,Size,Dir) &
   close_to_size(Th,Col,CtoSize) & Size>=CtoSize

near(Th,Dir) <=
   see_patch(Col,Size,Dir) &
   near_to_size(Th,Col,NSize) & Size>=NSize

see(Th,Dir) <=
   see_patch(Col,Size,Dir) &
   see_size(Th,Col,SSize) & Size>=SSize

rel next_to_size(?thing,?colour,?num),
   close_to_size(?thing,?colour,?num), see_size(?thing,?num)

next_to_size(bottle,green,200)
next_to_size(bottle,blue,180)
next_to_size(drop,red,450)

close_to_size(bottle,green,150)
....

see_size(bottle,green,50)
see_size(bottle,blue,45)
see_size(drop,red,110)

The <= in a rule should be read as ‘if’. The left hand side of the <= is the rule head, the right hand side the rule body.

The rules are implicitly universally quantified with respect to all their variables - the names beginning with an upper case letter. The argument
types in the declarations for all but the percept relations are annotated with a prefix `?`. This indicates that the relation’s rule definition can be used both to test or to find instances of the relation. Argument types for dynamic relations such as `see_patch`, being defined solely by facts, do not need to be annotated. They are all implicitly `?` annotated.

In general, an agent’s Belief Store will contain a whole hierarchy of rules, progressively defining higher level concepts in terms of lower level concepts, but with most ultimately rooted in a set of received percepts such as `see_patch`.

Making use of an extra relation definition, we could replace the guard of the first rule of `collect_bottle` by the single condition `delivered()`.

```
rel delivered()

delivered() <=
   next_to(drop,_) & next_to(bottle,_) & gripper_open()
```

### 2.5.1 Using a speed function to slow the robot down

We now slow our robot down in stages at particular distances from its target by having defined proximity relations and different rules for each proximity condition. Another approach, especially when the robot is getting close, is to slow it down as it gets closer by having the forward speed defined by a function with argument the size of the colour patch for the approached thing. This is particularly useful when the robot is approaching a bottle to prevent the bottle being pushed away by the forward momentum of the robot. Using a function also saves having to define `very_close_to` etc.

Using a `speed` function we can change the third rule of `get_next_to` making use of a modified `close_to` relation that also gives the size and the colour of the close thing.

```
close_to(Th,_,Col,Size) ~> 
   approach(Th,speed(Th,Col,Size),0.5)

rel close_to(?thing,?dir,?colour,?num)

close_to(Th,Dir,Col,Size) <=
   see_patch(Col,Size,Dir) & 
   close_to_size(Th,Col,CtoSize) & Size>CtoSize
```
fun speed(thing, colour, num) -> num
speed(bottle, green, Size) :: 150=<Size<170 -> 0.5
speed(bottle, blue, Size) :: 140=<Size<155 -> 0.5
speed(bottle, green, Size) :: 170=<Size<190 -> 0.3
speed(bottle, blue, Size) :: 155=<Size<170 -> 0.3
speed(bottle, green, Size) :: Size>=190 -> 0.1
speed(bottle, blue, Size) :: Size>=170 -> 0.1
speed(drop, _, _) -> 0.5

Function rules are committed choice rules with an optional commit test given between the :: and the ->. The first rule with a left hand side that matches the call with a commit test that succeeds is used to evaluate the call.

Now the close_to rule of the get_next_to procedure will be re-fired each time the percepts are updated and a different forward speed is computed. When approaching a bottle this means the speed is decreased in two stages from 0.5 to 0.1 as the robot gets very close to the bottle. There is no change as the robot gets very close to drop.

2.6 Core TelesoR Procedure Syntax

Now that we have looked at several TelesoR procedures, we can give a more general description of their syntax. Core TelesoR procedures more or less correspond to T-R procedures.

The overall form of a core TelesoR procedure is:

tel p(t₁,...,tₖ) % declaration of the argument types of p
p(X₁,...,Xₖ) {  
  K₁ ~> A₁
  .
  .
  Kₙ ~> Aₙ
}

p must be a name uniquely identifying the procedure.

The tᵢ are the types of the procedure’s k parameters. Each tᵢ may be a primitive type such as num, or a program defined type such as thing, or a type expression for a relation, function or TelesoR procedure. That a TelesoR procedure can be passed code when called is a difference from T-R procedures.

Each guard Kᵢ is a restricted QuLog query comprising an & conjunction of unnegated and negated conditions, with restrictions on the negated conditions. The restrictions will be explained in Chapter 3 when we describe QuLog and the guard evaluation process. We shall see in that chapter that
we can indirectly include quite complex conditions in guards using rule defined relations.

Each action $A_i$ is either a tuple $(a_1, ..., a_j), j \geq 0$ of robotic actions to be executed in parallel, or a single call to a TeleoR procedure. Except for the empty tuple $()$, used to indicate the do nothing action, we can drop the $(...)\text{ }$ brackets.

Allowing parallel execution of robotic actions is an extension of T-R procedures as described by Nilsson in [61],[63], although parallel robotic actions and even concurrent T-R procedure calls are allowed in an earlier research report [60]. The robotic actions are either discrete (aka ballistic) or they are durative. Discrete actions naturally terminate usually after a short time, for example $\text{close_gripper()}$ or $\text{open_gripper()}$, or $\text{beep()}$ for a horn. They cannot be prematurely terminated once started.

Durative actions may naturally terminate, for example a pickup action for a robot arm as will be used in the block tower building program of Chapter 4. More often they are open ended and continue unless explicitly stopped, or modified. For example, $\text{move(1.5)}$ is modified to $\text{move(0.5)}$ when a robot gets close to the thing it is approaching. It is stopped and replaced by a $\text{turn}$ action if the robot loses sight of the thing.

This is the key characteristic of durative actions. They can be stopped at any time even if they would eventually terminate. If parameterised, they can be modified whilst executing. The ability to modify durative actions allows for a smooth transition from one set of robotic actions to another. A durative robotic action is terminated only if a rule is fired which has robotic actions that neither include the durative action nor a parameter modification of the action.
Chapter 3
Introduction to QuLog
This introduction to QuLog is more than sufficient to understand all its uses in the rest of the book. Hopefully it will also convey to the reader the potential of its use for enhancing both the reasoning and behavioural aspects of TeleoR agents. For a more complete description of QuLog see the research report [19].

In Chapter 4 we will give a TeleoR program which is a modification of Nilsson’s block tower building program of [63]. As the program makes essential use of Belief Store knowledge rules, even recursive rules, we begin by describing the use of the QuLog interpreter to develop and test knowledge rules independently of their use by a TeleoR program.

Step by step we will develop and test the type definitions and Belief Store facts and rules for the tower building program. We explain how QuLog evaluates interpreter entered queries and hence TeleoR rule guards, and how the use of rules can be monitored.

The tower program definitions use lists, which are generic recursive structures in QuLog. After developing the rules we need for the tower builder, we introduce other list processing features of QuLog such as pattern match splitting and list comprehension expressions. Strings in QuLog are not lists but we show how they can also be concatenated and split in a similar way to lists.

QuLog has sets which do not contain duplicates. We show how they can be generated using set comprehensions and manipulated using the usual mathematical set operations. They can be converted into lists and vice versa. The same infix operator in can be used to access elements of lists, single character sub-strings of strings, and elements of sets.

The multi-threaded TeleoR agent shell, the use of which we shall introduce in Chapter 4, has percepts handling and message handling threads with default behaviours. The default behaviours can be modified by including with the TeleoR program certain QuLog action rule definitions. The functionality of the agent shell can also be enhanced by adding extra threads that execute QuLog actions. As we shall see in Chapter 6, a TeleoR rule action can have an attached sequence of QuLog actions to execute, such as Belief Store updates and message sends. We therefore finish the chapter by introducing key action primitives of QuLog and action rule definitions.

3.1 Getting started

The robotic arm and the blocks that must be re-configured to build a particular block tower might start as depicted in Figure 3.1. We assume that there is a camera that has a side view of the blocks with image processing software. This software can detect that the arm is holding block 7, that blocks 9, 1 and 2 are on the table, that block 6 is directly on top of block 1 etc.

There are three towers in the figure. The list [8,5,4,9] are labels of the blocks of the tallest tower. [3,6,1] and [2] are also towers. One task
we might want our tower builder to do is reconfigure the blocks so that \([4, 6, 2, 7]\) is a tower. This will involve un-piling blocks on top of blocks 6 and 4 in order to pick them up.

Unless we are absolutely confident we can correctly define tower and any auxiliary relations we may need, we should first test separately our definitions using the QuLog interpreter. We load into the interpreter just the relation and function definitions we will be using together with a set of facts representing some configuration of blocks. When we use these relation definitions with our TeleoR program these facts will not be in the program file. They will be supplied and frequently updated by the agent’s percept handling thread.

We could represent the configuration of blocks of Figure 3.1 as the facts:

```prolog
holding(7)
on(8,5)
on(5,4)
on(4,9)
on(3,6)
on(6,1)
on_table(9)
on_table(1)
on_table(2)
```

These are the facts that we will be included in the test file for our relation definitions.
There is no need to give fact and rule terminating fullstops, as are needed in Prolog. In QuLog and TeleoR the termination of a fact, rule or procedure is indicated by the the next line of text starting at the left end of the line. All facts, QuLog rules and TeleoR procedures must start at the left end of a new line. They can be continued over several lines providing the text on each continuation line is indented slightly to the right. This idea we borrowed from Python. Unlike Python the indentation does not need to be a fixed number of spaces, a single space will do. For readability we shall use several indentation spaces for the programs in the book.

If you have a version of the Emacs editor there is a qulog.el plug in file supplied with the TeleoR+QuLog software. This does auto-indenting as you are entering both QuLog rules and TeleoR procedure definitions, to aid program layout. It also uses colour to highlight key syntactic features. Lack of expected indentation, or an unexpected colour, often indicates a syntax error. You will have far less syntax errors if you use Emacs and this plug in.

For our bottle collector program we defined the thing, dir and colour types, and declared the types of its percepts, actions and TeleoR procedures using these defined types. Defining and declaring types for our Belief Store relations by the tower builder program is a good starting point.

In the figure the blocks are labelled 1 to 9. We can use this range of natural numbers as our block type. Only block things can be an argument to holding and on_table, and on has two block arguments.

```
def block::= 1..9 % A range type definition
dyn holding(block), on_table(block), on(block,block)
```

holding, on_table and on are declared as dyn (short for dynamic) relations rather than as percept relations as we want to be able to give facts for them in a program file and to update them in the interpreter. In Chapter 4 they will be declared as percept relations. If facts for declared percept relations happen to be included in the program file they will be ignored.

A file containing the above type definitions, declarations and the on, on_table and holding facts can now be loaded into the QuLog interpreter to check. Suppose the file is called towerBS.qlg, where .qlg is the extension for a QuLog or combined QuLog+TeleoR program file. After starting up the QuLog interpreter using the terminal command:

```
quilog
```
we load and compile the contents of the file using a consult action:

| ?? consult(towerBS).<return> 
Consulting towerBS...
... towerBS consulted success

| ?? is the QuLog prompt for a new query, expression or action command. It can be spread over several lines. <full stop><return> signals its termination.

If there are any syntax errors in the consulted program file, quite helpful error messages will be given. The program file can be edited and consulted again. Only one file can be consulted, but that file can contain commands to consult other files. Details are in the reference manual.

3.2 Defining and Testing Rule Defined Relations

Let us now address the issue of defining what a tower is. Our QuLog definition has to test whether a list of blocks is configured as a tower.

To define a tower we use two concepts. The concept of a stack of blocks, and the concept of an uncovered block. A list of blocks is a stack if for each adjacent pair \( B_1, B_2 \) on the list \( B_1 \) is on \( B_2 \), and the last block on the list is on the table. A block is uncovered if there is not a block on top of it. A list of blocks is then configured as a tower if it starts with an uncovered block and is configured as a stack.

In QuLog the most straightforward way to define a stack is recursively. Suppose we have a list of blocks of the form \([B_1,B_2,... Blocks]\) where \( Blocks \) may be the empty list of blocks [], or it could contain one or more blocks \([B_3,...]\). If we know that the list \([B_2,... Blocks]\) is a stack, then we can infer that \([B_1,B_2,... Blocks]\) is a stack if on\((B_1,B_2)\). This gives us our recursive rule. The simplest stack is a list of one block \([Block]\), such that we have on_table\((Block)\).

We need two QuLog rules to capture this recursive definition of a stack:

---

1 We cannot just enclose the file name inside list brackets as in Prolog, as the QuLog interpreter sees that as an expression. Expression queries, relation queries and action sequences can all be entered in the QuLog interpreter.
rel_stack(?list(block))
% The ? means list of blocks argument may be given or generated
stack([Block]) <=
on_table(Block)
stack([Block1, Block2, ..Blocks]) <=
on(Block1, Block2) & stack([Block2, ..Blocks])

,.. should be read as followed by list. It is the same as the Prolog | which
may be used instead of ,...
We can add these two relation rules to the towerBs.qlg file and consult
the file again. We can now test this definition by entering various stack
queries.
| ?? stack([3,6,1]).
yes
| ?? stack([6,1]).
yes
| ?? stack([6,3,1]).
no
| ?? stack([B1,6,B3]).
B1 = 3 : block
B2 = 1 : block
| ?? stack([8,..Blocks]).
Blocks = [5,4,9] : list(block)

The first three queries show we have a correct definition for stack for test-
ing or completing partial lists of blocks. Notice the answer bindings for any
variables in the query are given with their type.
We can find all eight stacks in Figure 3.1 with the query:
| ?? stack(S).
S = [9] : list(block)
...
S = [4,9] : list(block)
...
S = [5,4,9] : list(block)
...
S = [8,5,4,9] : list(block)
S = [1] : list(block)
..%..is entered to get more answers
S = [6,1] : list(block)
...
S = [3,6,1] : list(block)
...
S = [2] : list(block)

The...lines are displayed by the interpreter to separate the different answers. By default the interpreter displays up to 5 answers to a relation query. If we want to see more...needs to be entered.

We can also specify how many answers we want to see in the query. We could have used the query:

| ?? 8 of stack(S).

to get all eight answers displayed straight away.

### 3.3 Displaying the Program

At any point you can see the entire consulted program by entering the command

| ?? show.

You can just see the rules and type declaration for a particular relation, such as stack, by entering

| ?? show stack.
rel stack(?list(block))
stack([Block]) <=
on_table(Block)
stack([Block1,Block2,...Blocks]) <=
on(Block1,Block2) &
   stack([Block2|Blocks])

Notice that it retains the variable names used in the program file even though the rules have been compiled.

You can even enter show st instead of show stack. The QuLog interpreter will extend this partial name to each of the names of relations, functions or actions that start with that stub and display their rules preceded by their type declarations. This is useful if you have used a long name.

We can also just see type declarations and definitions using the types command.
def block ::= 1..9
dyn holding(block)
dyn on(block, block)
dyn on_table(block)
rel stack(?list(block))

To see the types of one or all the QuLog primitives stypes can be used. Each type will be displayed with a brief comment describing the primitive.

fun random_int(M : int, N : int) -> int
"Returns a random integer between M and N inclusive."

fun random_num() -> num
"Returns a random number between 0 and 1 inclusive."

Notice that in the first type declaration there are type annotated variables $M : \text{int}, N : \text{int}$ and these variables are used in the associated string comment. Type annotated variables can also be used in type declarations of user programs. Moreover, if any type declaration is immediately followed by a string comment, this comment will be displayed along with the type declaration when show or types is used. We give an example of this later.

### 3.4 Building Upon Tested Definitions

Now that we have stack recursively defined and tested we can define tower, the configuration that we want our TeleoR program to achieve. $[\text{Block,..Blocks}]$ is a tower if Block is uncovered and $[\text{Block,..Blocks}]$ is a stack. A first definition, for checking only, could be:

```
rel tower(list(block))  % test use only with arg. a ground list of blocks
tower([Block,..Blocks]) <=
   not on(_,Block) & stack([Block,..Blocks])
```

which we add to the QuLog program file.

The first condition is a negated condition read as not there exists something on Block. When this rule is used to test if a given list of blocks is
configured as a tower the not condition will be evaluated first to check that there is no other block recorded as being on Block in our \textit{Belief Store}. That is, there is an attempt to show that there is such a block on Block. If that succeeds the negated condition \textit{fails}, but if the attempt to find such a block \textit{fails}, the negated condition succeeds.

This way of confirming that negated test conditions hold is called \textit{negation as failure}. We can validate the use of such an inference rule by showing that it is implicitly a normal proof of the negated condition from the \textit{completion} of the program and a set of inequalities \cite{16}. The completion is a set of if \textit{and only if} definitions formed from the program rules and the inequalities tell us that different names for things denote different things, e.g. 2 \neq 3. It is the only way that negated conditions are checked in QuLog.

A constraint on its use is that by the time a negated condition is checked, in the left to right order of evaluating conditions in queries and rule bodies, all but underscore _ variables, or explicitly existentially quantified variables\textsuperscript{2}, must be bound to ground terms. An underscore variable is implicitly existentially quantified inside the not. For the tower rule the constraint will be satisfied. Block will be a ground term as the definition of tower will only be used for testing lists of blocks.

Two test queries are

\begin{verbatim}
| ?? tower([3,6,1]).
  yes |

| ?? tower([6,1]).
  no |
\end{verbatim}

For our use in guards of the \texttt{TeleoR} program for controlling a robot arm to build block towers we shall only need test use of our tower and stack definitions. However, let us see what we need to do to the tower definition if we wanted to use it to find towers as well as to check.

Just changing the mode to test or generate by changing the ! to ? on the type declaration will result in a mode error telling us that we have a negated condition \texttt{not on(_,Block)} where Block might be an unbound variable. One way to avoid this is to put it at the end of the rule:

\begin{verbatim}
rel tower(?list(block))  % test or generate use
tower([Block,..Blocks]) <=
    stack([Block,..Blocks]) & not on(_,Block)
\end{verbatim}

When used to find or to instantiate a template list this uses the stack rules to find candidate stacks and then rejects those that have a block on top of

\textsuperscript{2} So the negated condition could be written as \texttt{not exists B on(B,Block)}.  

31
the top Block. The trouble is that this is inefficient both for generating and testing as the crucial requirement that the first block is uncovered is only done after all the blocks on Blocks have been found.

Fortunately, there is another solution that uses a cheap extra test or generate condition as the first condition of the original rule. Where a program defined type T has a finite number of instances, i.e. is a range type such as block, or a union of such types, we can use a QuLog primitive isa(V,T) to find or test instances of the type. If V has a value when the isa is evaluated, it checks that the value belongs to type T. If V is an unbound variable, it will generate each of the possible values of type T in turn, as possible solutions to the isa condition. For example,

<table>
<thead>
<tr>
<th>?? 3 of isa(N,block).</th>
</tr>
</thead>
<tbody>
<tr>
<td>N=0 : block</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>N=1 : block</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>N=3 : block</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>N=4 : block</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>N=5 : block</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>N=6 : block</td>
</tr>
</tbody>
</table>

% .. is request for another 3 answers

A definition of tower that is good for testing or generating is:

```
rel tower(?list(block)) % test or generate use
tower([Block,..Blocks]) <=
    isa(Block,block) &
    not on(_,Block) & stack([Block,..Blocks])
```

If we add the definition:

```
rel clear(block)
clear(Block) <=
    not on(_,Block)
```

we can replace the not on(_,Block) condition by clear(Block).
### 3.4.1 Actions to update dynamic relations

Suppose we would like to check what tower answers we would get after an action to put the held block 7 on top of the currently clear block 2. One way is to edit our program file and to consult it again. However, because we have declared holding and on as dynamic relations, we can do the update as a command.

```prolog
| ?? forget holding(_); remember on(7,2).
| success
```

will do the update for us. Note that we have separated the update actions of the command using ;. This is the connective that must be used to separate QuLog actions. If we did this update tower([7,2]) would be inferable.

### 3.5 Query Evaluation and Backtracking

We have already mentioned that queries and rules have their conditions/calls evaluated left to right. But where a condition has multiple solutions all its solutions are not generated at once. In evaluating the rule body

\[
\text{isa(Block,block)} \& \text{not on(_,Block)} \& \text{stack([Block,..Blocks])}
\]

the QuLog evaluator finds the first solution binding for Block using the isa condition, which will be Block=1, then passes this on to the rest of the conditions in the rule body, which become

\[
\text{not on(_,1)} \& \text{stack([1,..Blocks])}
\]

For the facts given in Section 3.1, the test not on(_,1) will fail, as we have the fact on(6,1).

At this point the next solution, Block=2 to the isa condition will be found. Going back to a previously evaluated condition to find the next solution, if there is one, is called backtracking. Left to right evaluation of conditions with backtracking, to re-try and find another solution to a previously solved condition, is how QuLog finds solutions to queries and evaluates rule bodies. It is the Prolog evaluation method.

In this case the next solution binding Block=2 is a block that has no other block covering it. It is also directly on the table as we have the fact on(2,table). The conclusion is that [2] is a tower.

To find more towers using the facts of Section 3.1, the QuLog evaluator will first see if there are more solutions to the last condition stack([2,..Blocks]). As there are none, it will skip over the not test and look for the next solution to isa(Block,block). This will give the candidate binding Block=3. As this does not have a covering block, that [3,6,1] is a tower will be inferred.
Finally, after five more backtracking retries of the `isa` condition, the last solution `tower([8,5,4,9])` will be found.

### 3.6 Watching the Evaluation of One or More Relations

There is a `watch Rel` command that will display all calls to the relation `Rel`, say, which rule is being used, give bindings for variables in the call as the rule is used, give the instantiated rule preconditions, log whether the call succeeds or fails, give solution instantiations, and log when the call is retried on backtracking. `unwatch Rel` turns off the logging.

Any number of non-dynamic relations can be watched at the same time, but doing just one or two at one time is the most useful. Here is part of the log for a `tower` query using the test or generate version of its definition with a watch on `tower` and `stack`.

```
| ?? watch tower, stack.
success

| ?? tower([B1,6,B2]).
1: tower([B1, 6, B2])
  Call 1 unifies rule 1 output none
  Rule body is:
    isa(B1, block) & clear(B1) & stack([B1, 6, B2])
2: stack([2, 6, B2])
  Call 2 unifies rule 2 output none
  Rule body is:
    on(2, 6) & stack([6, B2])
  no (more) proofs using rule 2 trying next rule for call 2
2: stack([2, 6, B2]) no (more) proofs
3: stack([3, 6, B2])
  Call 3 unifies rule 2 output none
  Rule body is:
    on(3, 6) & stack([6, B2])
4: stack([6, B2])
  Call 4 unifies rule 2 output none
  Rule body is:
    on(6, B2) & stack([B2])
5: stack([1])
  Call 5 unifies rule 1 output none
  Rule body is:
    on_table(1)
5: stack([1]) succeeded
4: stack([6, 1]) succeeded
3: stack([3, 6, 1]) succeeded
```
1: tower([3, 6, 1]) succeeded
B1 = 3 : block
B2 = 1 : block

| ?? unwatch tower, stack. success

We can also watch the evaluation of functions and actions.

3.7 Syntax of Terms, Mode Annotated Types, and Code Types

Ground and template terms

- A term is a variable, a constant, aka atomic literal, such as a number, atom or string, a list structure or a set, or a compound term of the form \( c(trm_1, ..., trm_k) \) where \( trm_1, ..., trm_k \) are terms and \( c \) is an atom. \( c \) is the functor of the compound term.
- Terms containing no variables are ground terms.
- Lists or compound terms that contain variables are template terms.
- A variable is an alphanumeric sequence beginning within an upper case letter or underscore _ on its own is the anonymous variable. Different occurrences of _ denote different un-named variables.
- An atom is an alphanumeric sequence beginning within an lower case letter or any sequence of characters surrounded with single quote marks, e.g. 'Keith L. Clark'.
- A string is any sequence of characters surrounded with double quote marks, e.g. "Hi Peter!".
- A number has the usual numeric representations, for details see the manual.

We will introduce sets in Section 3.9.

Primitive type expressions

These are: nat, int, num, atom, string, atomic, list(TypeExp), set(TypeExp).
All numbers, atoms and strings are sub-types of the type atomic.

Mode annotations

The three basic mode annotations that can be attached to any non-code

\(^3\) That \( k \) can be 0 allowing compound terms such as empty() is a difference between QuLog and Prolog.
type expression in the type declaration for a QuLog relation are: !, ?, ??.
Only the first two can be attached to a code type expression. There is one
other mode annotation, @ on its own, which is implicitly @term.

- A moded type !t indicates that the argument must be given as a ground
term when the relation is called - that the argument is input only - and
must be a value of type t or a sub-type of t. If there is a call for which this
might not be the case a mode error will be signalled. No prefix is an implicit
! prefix, but show will always display the relation type with an ! prefix
if none was attached in the program file. As an example, !list(num), or
just list(num), indicates the argument must be given as a list of mixed
num, int or nat values.

- A moded type ?t means that in any call that argument may be an unbound
variable, a template term, or a ground term. If not a variable it must be a
term of type t or a subtype of t. If not a ground term it will be instantiated
to a ground term of type t or a subtype of t if the call succeeds. If a relation
rule is such that a ? moded argument might not be instantiated to a ground
term a mode error is signalled. As an example, ?list(num), indicates the
argument may be an unbound variable, a template list of num values such as
[2,3,8,U,...L], or a ground list of num values. If not ground it will be
a ground list of num, int or nat values after a successful call.

- A prefix ?? is a relaxation of the ? mode that does not require a call to
the relation to generate a ground value for the given argument. It may be
left as a template term, even as an unbound variable if that was given in
the call.

- @ means that the given argument, any term, will not be changed in any
way by a call to the relation.

Argument type expressions for a QuLog action procedure can be mode
annotated using the same annotations as for relations. No annotation is also
an implicit ! annotation.

The argument type expressions for TeleoR procedures and QuLog functions
are not mode annotated as they are all implicitly ! ground input only mode.

The argument type expressions for dynamic relations are also not mode
annotated because there is an implicit ? annotation for every argument.

list type expressions, and program defined structure types, can be given a
mixed mode using two mode annotations. For example, !list(?nat), or the
equivalent list(?nat), indicates that the argument must be a complete list
of natural numbers or variables, and that any variables will have been bound
to natural numbers after a successful call. list(?!nat) indicates that the
argument must be a complete list of natural numbers or variables, and may
still contain variables after a successful call. Finally, ?list(?!nat) indicates
the argument may be a variable, template or ground list of natural numbers,
but that it will be a complete list of numbers or variables after a successful
call. [3,X,7,Y] is such a list, [3,X,Y,...L] is not, as the list template is of
unknown length. It is not a complete list.
Code type expressions

- \( \text{rel}(m_1, \ldots, m_k) \), where each \( m_i \) is a moded type expression, denotes an argument that is a relation. As an example, \( !\text{rel}(\text{atom}, ?\text{num}) \), or just \( \text{rel}(\text{atom}, ?\text{num}) \), is a relation that must be given and must allow the uses in which the first argument will always be given as an \( \text{atom} \), and the second argument may or may not be given, but if given must be a \( \text{num} \) or a sub-type of \( \text{num} \), and if not given it will be bound to a \( \text{num} \) or a sub-type of \( \text{num} \).
- \( \text{act}(m_1, \ldots, m_k) \), where each \( m_i \) is a moded type expression, denotes an argument that is a action.
- \( (t_1, \ldots, t_k) \rightarrow t \), where \( t \) and each \( t_i \) is an un-moded type expression, denotes an argument that must be a function mapping a tuple of ground arguments of type \( (t_1, \ldots, t_k) \), or of sub-types of these types, into a ground value of type \( t \), or a sub-type of \( t \).

If a \( \text{rel} \) or \( \text{act} \) code type expression for an argument for a relation \( R \) is prefixed by \( ? \) this means that the code value - nearly always the atom name of a program defined or primitive relation or action - will be returned by a call to \( R \) if that call argument is an unbound variable. This enables us to index relations and actions using some term description and to retrieve appropriate code names given a description.

3.8 List and String Processing

The QuLog definition for the relation \( \text{app}(L_1, L_2, L_3) \), which holds when \( L_3 \) is \( L_1 \) with all the members of the list \( L_2 \) inserted after the last member of \( L_1 \), e.g. \( \text{app}([1,2],[3,4],[1,2,3,4]) \) holds, is:

\[
\begin{align*}
\text{rel} \ &\text{app}(\text{list}(T), \text{list}(T), ?\text{list}(T)), \\
&\text{app}(?\text{list}(T), ?\text{list}(T), \text{list}(T)), \\
&\text{app}(??\text{list}(T), ??\text{list}(T), ??\text{list}(T)) \\
&\text{app}([], L, L) \\
&\text{app}([U, \ldots], L_2, [U, \ldots]) \leftarrow \text{app}(L_1, L_2, L_3)
\end{align*}
\]

\( T \) is a type variable telling us that the relation is \textit{polymorphic}, able to be used with any type of list arguments. Its type declaration is a \textit{conjunctive} or \textit{intersection} declaration as all three apply and represent different possible uses of the relation.
The first moded type tells us that when the first two arguments are ground lists of \( T \) data values, and the last is a variable or template list term containing only values of type \( T \), then the last argument will be a ground list of \( T \) values if the call succeeds. The second tells us that if the last argument is a ground list of \( T \) values, and the first two are variables or template list terms containing only values of type \( T \), that they will be ground if the call succeeds. The last covers the case when each argument may be a template term. Its tells us that even after the call succeeds the arguments may still be template terms even though some of their variables have been bound to ground values.

Here are example uses.

| ?? app([1,hello],[-7,there,5],L).
L=[1,hello,-7,there,5] : list(int\|atom)
% int\|atom is a union type expression

| ?? app(L1,L2,[1,2,3]).
L1=[] : list(nat)
L2=[1,2,3] : list(nat)
...
L1=[1] : list(nat)
L2=[2,3] : list(nat)
...
L1=[1,2] : list(nat)
L2=[3] : list(nat)
...
L1=[1,2,3] : list(nat)
L3=[] : list(nat)

| ?? app([X1,2],[X2,4,..L2],[1,X3,3,..L3]).
X1=1 : nat
X2=3 : nat
X3=2 : nat
L3=[4,..L2] : list(nat\|Ty1)

The first query shows we can append lists of mixed type giving an answer that is a list of the union of the element types. For the last query the first argument has become the ground term \([1,2]\), the second argument the template term \([3,4,..L2]\) and the last argument the template term \([1,2,3,4,..L2]\). The type of the binding \([4,..L2]\) is given as a list of values from the union of \( \text{nat} \) (the number 4), and the unspecified type \( \text{Ty1} \) of the elements of \( L2 \).

Here is a log of the last query:

| ?? watch app.
success

| ?? app([X1,2],[X2,4,..L2],[1,X3,3,..L3]).
1:app([X1, 2], [X2, 4,..L2], [1, X3, 3,..L3])

38
Call 1 unifies rule 2 output $X_1 = 1$
Rule body is:
\[
\text{app}([2], [X_2, 4,..L_2], [X_3, 3,..L_3])
\]
2: $\text{app}([2], [X_2, 4,..L_2], [X_3, 3,..L_3])$
Call 2 unifies rule 2 output $X_3 = 2$
Rule body is:
\[
\text{app}([], [X_2, 4,..L_2], [3,..L_3])
\]
3: $\text{app}([], [X_2, 4,..L_2], [3,..L_3])$
Call 3 unifies rule 1 output $L_3 = [4,..L_2]$ $X_2 = 3$
No rule body
3: $\text{app}([], [3, 4,..L_2], [3, 4,..L_2])$ succeeded
2: $\text{app}([2], [3, 4,..L_2], [2, 3, 4,..L_2])$ succeeded
1: $\text{app}([1, 2], [3, 4,..L_2], [1, 2, 3, 4,..L_2])$ succeeded

\[
X_1 = 1 : \text{nat}
\]
\[
X_2 = 3 : \text{nat}
\]
\[
X_3 = 2 : \text{nat}
\]
\[
L_3 = [4,..L_2] : \text{list(nat)}|Ty_1
\]

There is a QuLog primitive \texttt{append} with exactly the same rule structure but two more modes of use which are refinements of the first two modes for \texttt{app}. 

\[3.8.1 \textbf{The <> operator and list pattern matching}\]

For the appending of ground lists QuLog has a primitive function $\langle \rangle$ used as an infix operator. Its type declaration is

\[
\text{fun } \langle \rangle: (\text{list}(T), \text{list}(T)) \rightarrow \text{list}(T)
\]

Our first use of the \texttt{app} relation for appending two lists $[1,3]$, $[-7,4,5]$ is achieved with the expression $[1,3] \langle \rangle [-7,4,5]$. This can be entered in the QuLog interpreter as an expression query:

\[
| ?? [1,\text{hello}] \langle \rangle [-7,\text{there},5].
\]
\[
[1,\text{hello},-7,\text{there},5] : \text{list(int)}|\text{atom}
\]

Another example use is the expression $L_1 \langle \rangle [3] \langle \rangle L_2$ which will append ground list values of $L_1$ and $L_2$ inserting 3 between them. $L_1$ and $L_2$ do not have to be lists of integers or even numbers. If $L_1=[a,b]$ and $L_2=[4.5,c]$ the expression value is $[a,b,3,4.5,c]$ of type $\text{list(\text{atom} \mid \text{num})}$.

We can use $\langle \rangle$ to recursively define a function that will reverse a ground list of values of any type:
fun rev(list(T)) -> list(T)
   rev([]) -> []
   rev([Hd,..Tl]) -> rev(Tl)<>[Hd]

A better definition is:

fun rev(list(T)) -> list(T)
   rev(L) -> revOnto(L,[])

fun revOnto(L:list(T),Onto:list(T)) -> list(T)
"Returns the reverse of L appended to Onto"
   revOnto([],Onto) -> Onto
   revOnto([Hd,..Tl],Onto) -> revOnto(Tl,[Hd,..Onto])

which constructs the reverse of the list as it recurses down the list. It has
linear, as against the quadratic complexity of the first definition.

The following polymorphic function finds the list of root labels of a binary
tree. We need a recursive type definition for the generic tree type.

def tree(T) ::= emp() | tr(tree(T),T,tree(T))

fun tree2list(tree(T)) -> list(T)
   tree2list(emp()) -> []
   tree2list(tr(Left,Lab,Right)) ->
       tree2list(Left)<>[Lab]<tree2list(Right)

An example call is
| ?? tree2list(tr( tr(emp(),2,emp()), 5, tr(emp(), -7, 
  tr(emp(),3,emp())))).
[2,5,-7,3] : list(int)

Pattern match splitting of lists

<> can also be used in a list pattern expression for splitting ground lists.
Our second example use of app can be achieved using the query:

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

with the same four answers. To exclude the empty list answers, we can use:
where the inequalities are constraints on the bindings for the variables of the pattern.

is QuLog's pattern match operator that will also evaluate its left hand argument which has to return a ground list. For list splitting use, the right hand side can have any number of occurrences of list patterns separated by . The pattern match

\[ [1,2]<\>[3,4,5] =? \text{Before}<\>[3]<\>\text{After} \]

will produce the bindings \( \text{Before}=[1,2], \text{After}=[4,5] \).

\[ \text{L1}<\>\text{L2}<\>\text{L3} =? \text{Before}<\>[3]<\>\text{After} \]

will append the three ground lists \( \text{L1}, \text{L2} \) and \( \text{L3} \). It will then find all the occurrences of \( 3 \) in the resulting list generating bindings for \( \text{Before} \) and \( \text{After} \), the sub-lists before and after each occurrence of \( 3 \). For example:

\[ \text{L1}=[1,3] \& \text{L2}=[4,2] \& \text{L3}=[3,9] & \]
\[ \text{L1}<\>\text{L2}<\>\text{L3} =? \text{Before}<\>[3]<\>\text{After}. \]
\[ \text{Before}=[1] : \text{list(nat)} \]
\[ \text{After}=[4,2,3,9] : \text{list(nat)} \]

We can add any single condition constraints to the splittings of a list achieved using patterns. These are relation calls that appear after the list pattern with a separating . The constraints are checked left to right as candidate bindings for the variables of the pattern are found and when not satisfied will result in a different splitting being generated, if any remain.

\[ \text{L1}=[-1,2] \& \text{L2}=[4,6] \& \text{L3}=[-3,9] & \]
\[ \text{L1}<\>\text{L2}<\>\text{L3} =? \text{Before}::#\text{Before}>1 <> \text{E}::(E<0) <> \text{After}. \]
\[ \text{Before}=[-1,2,4,6] : \text{list(int)} \]
\[ \text{E}=-3 : \text{int} \]
\[ \text{After}=[9] : \text{list(nat)} \]

is the QuLog prefix operator function for finding the length of a ground list, the length of a string, or the size of a set. The first constraint \( \#\text{Before}>1 \) therefore requires \( \text{Before} \) to be bound to a list of at least two members.
The constrained pattern \([E]::(E<0)\) means \([E]\) must be single element list containing a negative number.

### 3.8.2 The ++ operator and string pattern matching

String concatenation, and splitting subject to constraints, can be done using the ++ operator. This allows quite complex string processing.

A simple example is the relation \(\text{words}(S: \text{string}, L: ?\text{list(string)})\) such that \(L\) is the list of the ‘word’ sub-strings appearing in string \(S\). A word sub-string does not contain any space character, and any number of spaces may separate each word. Its recursive definition is:

```prolog
rel words(S: string, L: ?list(string))  
"L is list of word substrings of S"

words(Str,[Str]) :: word(Str)  
words(Str,[Word,..Words]) <=  
    Str =? Word::word(Word) ++ Sps::allSpaces(Sps) ++  
    RemStr::words(RemStr,Words)

rel word(string), allSpaces(string)

word(Str) <=  
    #Str>0 &  
    not exists CharStr (CharStr in Str & CharStr = " ")

allSpaces(Str) <=  
    #Str>0 &  
    not exists CharStr (CharStr in Str & CharStr \= " ")

| ?? words("Hello Bill how are you",Words).  
Words=[["Hello","Bill","how","are","you"] : list(string)
```

In the rules for \(\text{word}\) and \(\text{allSpaces}\) the existential quantification of \(\text{CharStr}\) is necessary to avoid a mode error, as it will not be ground when the negated conjunction is tested.
3.9 List and Set Comprehension Expressions

Like Prolog, QuLog has both deductive data base and list manipulation capabilities. The interface between the two is provided by QuLog’s list comprehension expressions and the primitive \texttt{in} relation for accessing the members of a ground list one at a time.

\begin{verbatim}
| ?? X in [1,2,3] & X in [2,3,4] & 0 = X mod 2.
X=2 : nat
\end{verbatim}

This query finds the even numbers in both [1,2,3] and [2,3,4]. Here, the QuLog expression value unification primitive = is being used as an arithmetic expression evaluator.

We can create lists from relational queries using list comprehension expressions.

\begin{verbatim}
| ?? [X :: X in [1,2,3] & X in [2,3,4] & 0 = X mod 2].
[2] : list(nat)
| ?? \{B :: isa(B,block) & not on(_,B)\}. % List of uncovered blocks
[8,3,2] : list(block)
| ?? \{B :: isa(B,block) & not on(_,B)\}. % Set of uncovered blocks
{2,3,8} : set(block)
\end{verbatim}

The difference between the use of [...] brackets and {...} braces is that the latter will create a set which has no duplicates. A set is always displayed surrounded with braces and with its elements ordered.

\begin{verbatim}
| ?? \{Tower :: tower(Tower)\}.
{[2], [3,6,1], [8,5,4,9]} : set(list(block))
\end{verbatim}

The system type declaration for \texttt{in} is:

\begin{verbatim}
rel in(?T,list(T)), in(?string,string), in(?T,set(T))
\end{verbatim}

It can only be used for accessing elements of ground lists. There is another primitive list membership relation with type declaration:

\begin{verbatim}
rel member(?T,list(T)), member(?T,list(T))
\end{verbatim}

This can be used both for accessing elements and binding variables of template lists.

\begin{verbatim}
| ?? member(2, [1,X,3]).
X=2 : nat
| ?? 3 of member(2,L).
L = [2,...A] : list(Ty1||nat)
\end{verbatim}
L = [A, B, 2, ..C] : list(Ty1||Ty2||Ty3||nat)

If we wanted to convert from a set to a list we apply the to list function denoted symbolically as the \[ \] pair of square brackets.

| ?? | {3,-2,8,3,-1,8}. | {-1,-2,3,8} : set(int) |
| ?? | \[\]({3,-2,8,3,-1,8}). | {-1,-2,3,8} : list(int) |

| ?? | \[\]({3,-2,8,3,-1,8}). | {-1,-2,3,8} : list(int) |

The last expression query has converted an unordered list with duplicates into an ordered list without duplicates. We can use the double conversion to define a polymorphic sortViaSet function that drops duplicates.

fun sortViaSet(list(T))-> list(T)
sortViaSet(L) -> \[\]({}(L))

However, a more general ordering function is the following higher order function that takes the ordering relation for the elements as an argument. It does not drop duplicates.

fun bsortR(L:list(T),OrdRel:rel(T,T)) -> list(T)
"Sorts L using order relation OrdRel via bubble sort method"

bsortR(L,OrdRel) ->
    bubbleUntilOrdered(bubble(L,OrdRel), OrdRel)

fun bubbleUntilOrdered((nat,list(T)),rel(T,T)) -> list(T)

bubbleUntilOrdered((0,OrdL),_) -> OrdL
% The 0 means there were no exchanges when bubble last called
bubbleUntilOrdered((_,BubbledL),OrdRel) ->
    bubbleUntilOrdered(bubble(BubbledL,OrdRel), OrdRel)

fun bubble(L:list(T), OrdRel:rel(T,T)) -> (nat,list(T))
"Recurses down L exchanging out of order pairs returning number of exchanges and new list"

bubble([],_) -> (0,[])
bubble([E],_) -> (0,[E])
bubble([E1,E2,..L],OrdRel) ::
    OrdRel(E1,E2) & not OrdRel(E2,E1) ->
    update(bubble([E2,..L],OrdRel), (0,E1))
bubble([E1,E2,..L],OrdRel) -> % E1=E2 or OrdRel(E2,E1)
    update(bubble([E1,..L],OrdRel), (1,E2))

fun update((nat,list(T)),(nat,T)) -> (nat,list(T))
"Utility function for updating returned pair from
a recursive bubble call"

update((Swaps,List),(ZeroOrOne,E)) ->
(Swaps+ZeroOrOne, [E,..List])

rel(T,T) is the type expression for a binary test only relation with arguments
of type T, the type of the elements of the list argument of bsortR. This does
not mean that the Ord relation should only compare terms of type T. It can
be a more general relation able to take arguments of a more inclusive type. It
also need not be a test only relation. The relation argument must only cover
the testing of pairs of the list elements. So if T=int, Ord could be given as a
relation of type rel(num,?num).

There is no need to have the test not Ord(E1,E2) in the last rule as the rule
will only be used if the second rule is not used, hence if the test
Ord(E1,E2) fails. This is because all function rules are committed choice.
If the rule head pattern matches the function call, and any test such as
Ord(E1,E2) holds, only that rule is used to evaluate the call. Action rules
are also committed choice. A relation rule can be made committed choice by
attaching a :: Test condition to the head of the rule. We give an example
of this in Section 3.12.

Here are three expression queries to bsortR.

| ?? bsortR([2,8,-3,7,1,8],=<).
[-3, 1, 2, 7, 8] : list(int) |

| ?? bsortR([2,-3,7,1,8,8],>=).
[8, 7, 2, 1, -3] : list(int) |

| ?? bsortR([john,23,g(b),55,g(2),"hello",23,apple],@=<).
[23, 23, 55, apple, john, "hello", g(2), g(b)] : list(term) |

@=< is the QuLog term order relation. QuLog has two primitive sort relations
both based on merge sort.

sort(list(??T),?list(??T)) sorts any list of terms, including variables
or template terms, using @=<. It removes duplicates.

45
sortr(list(T),?list(T),rel(T,T)) sorts any ground list using a given order relation. Like bsortR it does not remove duplicates.

3.9.1 Set operations

QuLog has functions

- union, inter, diff

all of type (set(T),set(T))->set(T), for manipulating sets. They can be used as infix operators.

| ?? {2,4,8} union {4,9,11}. |
| {2,4,8,9} : set(nat) |

| ?? {2,4,8} diff {4,9,11}. |
| {2,8} : set(nat) |

| ?? {2,4,8} inter {4,9,11}. |
| {4} : set(nat) |

| ?? {1,2,3} union {3,4,5,6} inter {3,5}. |
| {1, 2, 3, 5} : set(nat) |

The QuLog type checker will check that these operations are only used with set arguments.

3.10 Path finding in a graph

In Section 3.16 we discuss the implementation of an information agent that holds a plan of a network of doorway connected rooms some of which contain charging stations. The doorways in each room all have a different colour. The plan data is updated whenever the information agent receives information about a doorway in a room being closed or open. A robotic agent can send a request to the information agent to advise it on the shortest path of open doors from its current room to a charger room not currently being used.

Finding such a doorway path is a special case of the finding of a shortest path in a graph of bi-directional labelled edges, as depicted in Figure 3.2, where all the edges emanating from each node have different labels. Such a graph can be represented as a symmetric linked relation that queries a non-symmetric link relation. The end nodes of the shortest paths that we might want to find can also be defined by a test relation such as end.
Fig. 3.2 Graph with bi-directional edges

rel link(Lab:nat,Nd1:atom,Nd2:atom)
"Graph nodes Nd1 and Nd2 are linked by edge labelled Lab"
link(1,a,b)
link(2,a,c)
link(3,a,g)
link(3,c,b)
link(4,f,b)
link(3,f,h)
link(1,f,g)
link(1,d,c)
link(2,d,f)

rel linked(?nat,?atom,?atom)
linked(E,N1,N2)<= link(E,N1,N2)
linked(E,N1,N2)<= link(E,N2,N1)

rel end(atom)
end(a)
end(h)
Below is a general purpose `graph_path` higher order relation definition. The evaluation of a call to the relation is an algorithm that will uniformly construct all the loop free paths in a graph defined by the relation argument `GraphR`, starting at the given `First` node, one node at a time. The step by step growing of the paths emanating from `First` is done by the recursive evaluation of the call to the subsidiary `bfs` relation (breadth first search).

The `bfs` recursion terminates when a next node `Last` that extends one of the paths satisfies the test given by the `EndR` relation argument. The call returns a minimal length sequence of nodes connecting `First` to `Last` in reverse order. The `graph_path` relation then uses a call to `tuple_path` to map this node list into a sequence of tuple instances of the `GraphR` relation that is a reverse path from `First` to `Last`, with both nodes and edges of the path identified. For example:

```prolog
| ?? graph_path(6, Last, Path, linked, end).  
| Last = 8 : block  
| Path = [(f, 6, 8)] : list((atom, nat, nat))  
| ?? graph_path(4, Last, Path, linked, end).  
| Last = 1 : block  
| Path = [(b, 3, 1), (h, 4, 3)] : list((atom, nat, nat))
```

```
rel graph_path(First:NT, Last:?NT, TpPath:?list((ET,NT,NT)), GraphR:rel(?ET,?NT,?NT), EndR:rel(NT))  
"Last is last node of TpPath which is a shortest reverse path of tuples of the GraphR relation from First to an EndR node"  
graph_path(First,First,[],_GraphR,EndR) ::  
EndR(First)  
graph_path(First,Last,TpPath,GraphR,EndR) <=  
bfs([First],[],[[First]],NdPath,GraphR,EndR) &  
NdPath=[Last,..] &  
tuple_path(NdPath,GraphR,[],TpPath)
```

```
rel node2tuple_path(NdPath:list(NT),GraphR:rel(?ET,?NT,?NT), TpPath:?list((ET,NT,NT)))  
"TpPath is a list of GraphR tuples connecting the sequence of nodes in NdPath. It is an edge labelled reverse of path from the last node of NdPath to its first node"  
node2tuple_path([],[],[])  
node2tuple_path([ToNd,FromNd,...Nds],GraphR,  
[(EdgeNm,FromNd,ToNd),..TpPath]) <=  
GraphR(EdgeNm,FromNd,ToNd) &  
node2tuple_path([FromNd,...Nds],GraphR,TpPath)
```
rel bfs(NdsSoFar:list(NT), ENdPaths:list(list(NT)),
    NdPaths:list(list(NT)), NdPath:?list(NT),
    GraphR:rel(?_,?NT,?NT), EndR:rel(NT))

"NdPath is a reverse order loop free GraphR path extension of
some path in ENdPaths<>NdPaths to an EndR node. NdsSoFar is
list of all nodes in loop free paths of ENdPaths<>NdPaths
that all start at the same node."

bfs(NdsSoFar,ENdPaths,\[],NdPath,GraphR,EndR) ::
  ENdPaths \= \[] <=
      bfs(NdsSoFar,\[],ENdPaths,NdPath,GraphR,EndR)

bfs(_,ENdPaths,[[Nd,..RmNds],..],NdPath,GraphR,EndR) ::
  GraphR(_,FromNd,ToNd) & EndR(ToNd) <=
      NdPath=[ToNd,FromNd,..RmNds]

bfs(NdsSoFar,ENdPaths,[[FromNd,..RemNdNs],..RemNdPaths],
    NdPath,GraphR,EndR) <=

  NewENdPaths = [[NewNd,Nd,..RemNdNs] ::
    GraphR(_,FromNd,ToNd) &
    not ToNd in NdsSoFar] &
  bfs([ToNd::[ToNd,..] in NewENdPaths]<>NdsSoFar,
     NewENdPaths<>ENdPaths,RemNdPaths,NdPath,GraphR,EndR)

The following partial log of a watched call to \texttt{bfs} illustrates how the path [1, 3, 4] is found which is then mapped into the tuple path [(b, 3, 1), (h, 4, 3)].

| ?? graph_path(4,Last,Path,linked,end).

1:bfs([4], \[], [[4]], A, linked, end)
  Call 1 unifies rule 2 output none
  Rule body is:
    :: linked(_, 4, NewNd_0) & end(NewNd_0) <=
      A = [NewNd_0, 4]
  no (more) proofs using rule 2 trying next rule for call 1
  Call 1 unifies rule 3 output none
  Rule body is:
    NewENdPaths_0 = [[NewNd_0, 4]::linked(_, 4, NewNd_0) &
      not(NewNd_0 in [4])] &
  bfs([ToNd_0::[ToNd_0,..] in NewENdPaths_0] <> [4],
      NewENdPaths_0 <> \[], \[], A, linked, end)
2:bfs([3, 6, 4], [[3, 4], [6, 4]], \[], A, linked, end)
  Call 2 unifies rule 1 output none
  Rule body is:
    :: ([[3, 4], [6, 4]], \[]) <=
bfs([3, 6, 4], [], [[3, 4], [6, 4]], A, linked, end)
3:bfs([3, 6, 4], [], [[3, 4], [6, 4]], A, linked, end)
Call 3 unifies rule 2 output none
Rule body is:
:: linked(_, 3, NewNd_1) & end(NewNd_1) <=
A = [NewNd_1, 3, 4]
3:bfs([3,6,4], [], [[3,4], [6,4]], [1,3,4],linked,end) succeeded
2:bfs([3,6,4], [[3,4], [6,4]], [], [1,3,4],linked,end) succeeded
1:bfs([4], [], [[4]], [1,3,4], linked, end) succeeded
Last = 1 : block
Path = [(b, 3, 1), (h, 4, 3)] : list((atom, nat, nat))

3.11 Iterating over all Solution Bindings for a Query using forall

We can wrap all solutions to a query as a list using a list comprehension expressions. However, sometimes all we want to do is check that all elements of such a list satisfy some test condition.

An inefficient way of doing this is to first construct the comprehension list and then use the higher order testAll relation.

```
rel testAll(L:list(T), Test:rel(T))
"All elements of L satisfy monadic relation Test"

    testAll([],_)
    testAll([Hd,..Tl],Test) <=
        Test(Hd) & testAll(Tl,Test)

rel even(int)
even(N) <= 0 = N mod 2
```

| ?? testAll([#Bs :: tower(Bs)], even). no |

We get the answer no because, as depicted in Figure 3.1, there is a tower of length 3 and one of length 1.

To do the test on all the elements of the list comprehension without constructing the list we can use forall.

| ?? forall Bs (tower(Bs) => even(#Bs)). no |
If there are local variables in either the antecedent or consequent of the implication these must be explicitly existential quantified as would be the case in predicate logic. For example, to test that all the nodes of the \texttt{linked} relation graph other than \texttt{h} are connected to at least two nodes we would use the query:

$$| \forall Fr, To1 \ (\exists E1 \ \text{linked}(E1, Fr, To1) \ & \ Fr\neq h \implies \exists E2, To2 \ \text{linked}(E2, Fr, To2) \ & \ To2\neq To1) |$$

\textbf{yes}

In predicate logic this would be expressed as:

$$\forall Fr \exists To1 \ (\exists E1 \ (\text{linked}(E1, Fr, To1) \ \& \ Fr\neq h) \implies \exists E2 \ (\text{linked}(E2, Fr, To2) \ \& \ To2\neq To1))$$

In QuLog the existential quantifications for \texttt{E1} and \texttt{E2} can be dropped and both variables replaced by the anonymous underscore variable as each only appears in one condition. Underscore variables in a \texttt{forall}, \texttt{not} or a \texttt{once} (find at most one solution) condition are implicitly existentially quantified in front of the predication in which they appear. The above query can therefore be re-expressed as:

$$| \forall Fr, To1 \ (\text{linked}(\_, Fr, To1) \ & \ Fr\neq h \implies \exists To2 \ \text{linked}(\_, Fr, To2) \ & \ To2\neq To1) |$$

\textbf{yes}

Using \texttt{forall} and \texttt{in} we can give a non-recursive definition of \texttt{testAll}.

\begin{verbatim}
rel testAll2(list(T), rel(T))
testAll2(List, Test) <=
    forall E (E in List => Test(E))
\end{verbatim}

\subsection*{3.12 Type and Mode Checking in QuLog}

The type and mode checking done by the QuLog compiler is a form of abstract interpretation [23]. The details are beyond the scope of this chapter and are given in [19]. We shall only need to use the modes \texttt{!} and \texttt{?} for an agent’s \texttt{Belief Store} relations.

Suffice it to say that when an argument of a relation or action is moded \texttt{!}, that argument in every call to the relation needs to be a ground value of the specified type, or a sub-type of that type. When it is moded \texttt{?}, the argument may be a ground value of the specified type or a sub-type, or a template term of the specified type or a sub-type, or an unbound variable. After a successful
evaluation of the call that argument will be ground value of the required type or a sub-type.

As an example of sub-types, nat is a sub-type of int which is a sub-type of num. All are sub-types of atomic which also has atom and string as sub-types. list(T1) is a sub-type of list(T2) if T1 is a sub-type of T2. All are sub-types of term. term covers all QuLog data values.

Every use of a ground input value in the body of a rule should normally be in a call that can handle that type or a super-type. If some use requires a sub-type value there must be an earlier runtime type test to select out that sub-type. The program below to sum numbers of a list of any terms is an example of the need for such a runtime type test.

Any variable in a ? moded argument position in the head of the rule should normally be in some call in the rule body that will generate a ground value of the required type or a super-type. If the test or generate call in which it appears in the body may generate a super-type value there must be a later runtime type test to filter out all but the required sub-type bindings for the variable.

We may also need to insert type tests for local variables of a rule body if some later use requires a sub-type of the type of value that may be generated by an earlier call. The runtime type test must be between the two calls.

### 3.12.1 Type test primitives

type(Term,TypeExp) can be used to test for any primitive, program defined type or code type. Its second argument TypeExp is a mode annotated type expression using either the ! or ? modes, with no annotation being equivalent to having used ! as in a relation type declaration. A ! moded type test checks that Term is a ground term of type TypeExp. A ? moded TypeExp checks that Term is a variable of the given type, or a template or ground term of that type. If Term is the first occurrence of a variable in a rule or query the type checker will give it type TypeExp.

isa(Term,FiniteType) can be used to test or generate members of any finite type. Term is either a variable which will be progressively bound all the ground instances of the unannotated FiniteType, or a value to be tested as an instance of FiniteType.

**Example use of type**

Suppose we wanted to sum all the numbers that appear on a list of any terms, including variables. The relation could be defined as:
The type expression \texttt{list(@)} means that the first argument is a complete list of possibly non-ground terms and that the terms will not be changed in any way by a call to the relation.

The second rule is a 	extit{committed choice} rule, \texttt{type(N,num)} being the commit test. If the commit test succeeds the evaluation of a \texttt{sumNumsOnList} call will just use the second rule at that point in the recursive evaluation and not try to use the last rule. Without the \texttt{type} test we would get a type error.

A possible query to this relation is:

\begin{verbatim}
| ?? sumNumsOnList([apples,4,U,[1,2,k],g(a,X),-2,9],Sum).
Sum=11 : nat
\end{verbatim}

where all but numeric values at the top level of the list are ignored.

### 3.13 Action rule programming

In Section 3.4.1 we illustrated the use of the \texttt{remember}, \texttt{forget} action primitives for updating dynamic relation facts, and the use of an action sequence command in the \texttt{QuLog} interpreter. The actions of the sequence are separated by semi-colons to emphasise that they were ‘doing’ something rather than just querying.

We can however embed a conjunctive query in an action sequence by prefixing the query with \texttt{?}. For example:

\begin{verbatim}
| ?? ? loc(robot1,Rm) & open_connected(blue,Rm,NewRm) & not loc(_,NewRm);
   forget loc(_,NewRm); remember loc(robot1,NewRm).
Rm = 0 : room
NewRm = 1 : room
\end{verbatim}

The relations being queried are:

\begin{verbatim}
def room ::= 0..15
def robot ::= robot1 | robot2 | robot3
\end{verbatim}
def door ::= blue | green | red | yellow
def compass_dir ::= 0..359
dyn loc(robot,room)
loc(robot1, 0)
loc(robot2, 3)
dyn closed(door)
closed(yellow,5)
closed(yellow,4)

rel connects(?door, ?room, ?room, ?compass_dir)
"Relational map of the room connections in environment."
connects(blue,0,1,90)
connects(blue,2,3,90)
... 
connects(green,1,5,180)
connects(green,2,6,180)
...

rel connected(?door, ?room,?room, ?compass_dir)
"Relation defining the two way door connection relation between rooms"
connected(Door,Rm1,Rm2,Dir) <=
    connects(Door,Rm1,Rm2, Dir)
connected(Door,Rm1,Rm2,Dir) <=
    connects(Door,Rm2,Rm1,RevDir) &
    Dir = rev_dir(RevDir)

rel open_connected(?door,room,room)
open_connected(Door,Rm1,Rm2) <=
    connected(Door,Rm1,Rm2,_) & not closed(Door,Rm1)
% Only need to make sure door seen from one room is closed
% as closed facts for both rooms will be added if door seen
% as shut from either side.

fun rev_dir(compass_dir) -> compass_dir
rev_dir(Degrees) -> Degrees+180 mod 360

The connected relation defines a static map of a set of connected square rooms in a rectangular configuration with sides running north/south or east/west. Coloured doors connecting rooms are always located in the cen-
tre of a wall and straight metal lines, all passing through the room centre, connect the centres of the doorways. In each room the different doors have different colours.

The robots travel through open doorways, always following the metal lines. They navigate using a path of terms of the form

\[
\text{through}(\text{Door:door}, \text{Rm1:room}, \text{Rm2:room}, \text{Dir:compass\_dir})
\]

The compass\_dir argument is used to tell the robots on which side of the room \text{Rm1} their exit door \text{Door} to the next room \text{Rm2} is located. They have a compass that will give them their current direction. This and the compass direction of the exit door can be used to determine the better way to turn (left or right), if at all, when reaching the room centre. They turn until they are pointing in the compass direction of their exit door. They can then lock onto and follow the metal line to the exit door. When the robots are moving forwards they are following a metal line.

The open\_connected relation defines a dynamic map of the rooms that are connected with open doorways. It does not have the fourth compass\_dir argument so can be used as the graph defining relation in calls to \text{graph\_path} of Section 3.10.

The example command queries the dynamic \text{loc} relation and the relation open\_connected before updating the \text{loc} fact for \text{robot1}. The assumption is that it is moving through a doorway coloured blue.

Below is a defined QuLog action for updating any robot’s location given the robot’s name and the colour of the doorway it is passing through.

```qulog
act update_loc(robot)
update_loc(Rob,Door) ~>
  ? loc(Rob,Rm) & open_connected(Door,Rm,NewRm) &
  not loc(_,NewRm);
  forget loc(Rob,_); remember loc(Rob,NewRm)
```

QuLog action rules use the same \text{~>} operator as TeleoR action rules.

Defined and primitive actions must always succeed when called. If they do not there is a runtime error. \text{forget} always succeeds even if there is no matching dynamic fact to forget. \text{remember} always succeeds and will only update the dynamic facts if the fact being remembered is not already a stored fact. It adds the new fact to the end of the sequence of facts for the dynamic relation.

| ?? update_loc(robot3,blue,NewRm).

QuLog exception - exception term: relation\_call\_failure\_in\_action(loc(robot3, _95))
This exception is reported as there is no \texttt{loc} fact for \texttt{robot4}. A normal \texttt{loc} query will just report \texttt{no}.

\begin{verbatim}
| ?? loc(robot3,Rm).
no
| ?? update_loc(robot3,blue,NewRm).
robot3 current location unknown or its room does not have an open blue door
\end{verbatim}

Failures are not tolerated in action sequence commands or action rule bodies because there is no backtracking in an action sequence evaluation, except local backtracking inside a conjunctive query \texttt{(?C1&...&Cn)} within the action sequence. There may be backtracking to find the first solution to the \texttt{C1&...&Cn}, but thereafter there will be no further backtracking to find another solution should a following action or query fail.

An alternative definition of \texttt{update_loc} puts the query to \texttt{loc} and \texttt{connected} in a guard, and has two rules:

\begin{verbatim}
update_loc(Rob,Door) ::
    loc(Rob,Rm) & open_connected(Door,Rm,NewRm) ~> 
        forget loc(Rob,_); remember loc(Rob,NewRm)
update_loc(Rob,Door) ~> 
        write_list([Rob," robot's location unknown",nl_,
            "or its room does not have an open ",
            Door," door",nl_])
\end{verbatim}

Since all action rules are 'committed choice' like function rules, the new second rule is only used if the guard of the first rule does not hold. \texttt{write_list} is an action primitive for writing messages to the terminal. Its single argument is a list of terms with special terms, such as the atom \texttt{nl} (newline), used for formatting. Strings are displayed without the string quotes.

### 3.14 Threads and suspending queries

The agent shell supplied with the \texttt{TeleoR+QuLog} software is multi-threaded. \texttt{QuLog} threads execute actions. A \texttt{QuLog} process can have many action threads. However a \texttt{QuLog} agent process usually has just a handful of key threads that persist until the agent terminates. These are augmented with short lived threads, say to answer queries. The threads are time shared by
the underlying Qu_Prolog system, with each thread being given a 1 millisecond time slice. This is sufficient for about 6000 inference steps and 120 forget/remember updates of dynamic facts on a high end laptop of 2014. The number of updates is so low because at the moment the ground dynamic facts are compiled in the same way as static facts and rules. However, if a quite complex sequence of queries to and updates of the dynamic facts must be completed in order to leave the Belief Store in a consistent state for other threads, the action code $S$ that will execute the sequence can be executed as atomic \{S\}. If need be the thread’s time slice will be extended until the sequence $S$ is exited. $S$ should not execute any action, such as the wait action described in Section 3.14.1, that will make the thread suspend.

Threads do not share variables. When an action call containing unbound variables is forked to be evaluated in a new thread the call is copied and given a fresh set of variables local to the new thread. Threads within the same QuLog process have two ways of communicating. They can send messages to each other, since each thread has a unique local name and a message queue of received and unread messages, or they can communicate by manipulating the dynamic facts of the process. All threads access the same dynamic facts, and the same rule defined relations, QuLog actions and TeleoR procedures. Threads in different QuLog processes can only communicate using messages.

Our TeleoR robotic agents have one thread, the message handling thread, as the message portal for all messages sent to the agent from threads in other agents. Communication between the standard threads of a TeleoR agent is by manipulation of its dynamic beliefs. The standard threads are the percept handler, the message handler and any threads evaluating TeleoR procedure calls.

Any thread in a QuLog process can fork a new thread executing one or a sequence of action calls using one of the following

fork_light ActCalls as \textcolor{red}{Id}

fork ActCalls as \textcolor{red}{Id}

fork ActCalls using SpaceReqs as \textcolor{red}{Id}

thread forking primitives. \textcolor{red}{Id} is the name of the thread. If given as a variable the runtime system will assign it an unused name of the form thread$n$, where $n$ is a natural number.

The difference between the first two is the size of the evaluation stack, heap and other data areas allocated to the thread. The fork_light primitive allocates sizes to these data areas sufficient for threads that will not do long query evaluations where many conditions have multiple solutions. fork allocates more space to these data areas sufficient in size for most thread evaluations. The last form of fork allows the programmer to specify the size of some or all these data areas. Details are in the TeleoR+QuLog manual.
The QuLog query and command evaluator is a thread automatically started when the QuLog interpreter process is entered with specified sizes for the data areas. Other threads can be forked as interpreter commands, or as actions of previously forked threads. The percepts and messages handling threads of the TeleoR agent shell are light threads.

3.14.1 Internal thread communication using dynamic facts

A thread can be made to wait for a query to the dynamic facts to succeed using the wait action. Here is a simple example of its use.

```
| ?? fork_light wait loc(robot3,Rm) retry [+loc];
    write_list([loc(robot3,Rm),
                    " has just been remembered",nl_])
    as loc_watch.
success
| ?? ....
% other queries and commands
| ?? remember loc(robot3,6).
success

loc(robot3,6) has just been remembered
```

The fork_light action starts a new thread named loc_watch that executes independently of the main interpreter thread. As there is no loc(robot3,..) fact the new thread immediately suspends. It has to wait until another thread remembers such a fact.

Each time a new loc fact is remembered the query loc(robot3,Rm) is re-tried. It is not retried if a loc fact is forgotten, or if a dynamic fact for any other relation is changed. The list [+loc] is the dynamic relation update event list for re-trying the query. A +p is the event of remembering a fact for p, a -p is the event of forgetting such a fact. Both can appear in the event list and any number of update events can be given. If one or more occur, the query is retried. The list should include all the dynamic fact update events that may make the query succeed when it has previously failed. retry all will cause the query to be retried if there is any change to a dynamic relation.

The thread loc_watch will persist until its wait query succeeds. At any stage we can terminate the thread using thread_exit(loc_watch) as an interpreter command, or as an action in any thread. A forked thread can terminate itself with the exit() action. An exit() command entered at the interpreter prompt will terminate the entire interpreter process as the
query/command evaluator is its initial thread. Terminating the initial thread of a QuLog process terminates the process.

We can also have the forked thread automatically terminate itself after a specified number of seconds by forking the thread using:

```prolog
| ?? fork_light {wait loc(robot3,Rm) retry [+loc]
  timeout 300 ~> exit();
  write_list([loc(robot3,Rm),
    " has just been remembered",nl_])}
```

as Id.
Id = thread1 : atom

thread1 is the name given to the thread by the QuLog system. The allocated name is always different from the names of running threads.

Below is a more elaborate forked Belief Store monitoring thread that persists unless terminated by another thread executing the action `exit(loc_watch)`. It writes a message each time the location of a particular robot changes.

```prolog
act report_loc_changes(robot,room)
report_loc_changes(Rob,Rm) ~>
  wait loc(Rob,NewRm) & NewRm \= Rm retry [+loc];
  write_list([[(Rob,NewRm)," new current location"],nl_]);
  report_loc_changes(Rob,NewRm)
```

```prolog
| ?? ?loc(robot1,Rm);
  fork_light report_loc_changes(robot1,Rm) as loc_watch.
Rm = 1 : room

| ?? update_loc(robot1,green).
success

(robot1,5) new current location

| ?? update_loc(robot1,blue).
success

(robot1,6) new current location
```

The messages are displayed by the background `loc_watch` thread.
3.15 Inter-process Communication using Pedro

Communication between two QuLog agent processes makes use of our Pedro [71] publish/subscribe and addressed message server. It runs as a separate process on any accessible host. For our example uses we shall assume it is running on the same host as all the communicating QuLog processes, but a Pedro server running in Australia can be used to allow agent processes running in Australia, the USA and the UK to communicate. We just have to name the host on which Pedro is running with a command line option when the QuLog processes are launched, or when a connection is made later using the connect_to_pedro system action.

Pedro can also be used to allow communication of QuLog term messages between QuLog processes and processes in C/C++, Java and Python. This is how we connect TeleoR agent processes with the Python simulations distributed with the software. Using an interface process in one of these languages is also the most convenient way of connecting our agents with real robots. A template for a ROS interface process which uses both the Python ROS interface library, and our Python Pedro API, is included with the software.

Routing all messages via a Pedro server does slow down the communication between a robot and a TeleoR agent. However a Pedro server can transfer around 18,000 messages per second between two processes, where each message is a 20 element list and all three processes are on the same host. Since the interface between a TeleoR agent and a robot is at the mid level of percepts and actions, that communication speed is more than adequate. Raw pixel arrays will not be being communicated to a TeleoR agent, and durative actions are really lower level perceive/act controller processes. These are usually written in an imperative language where tighter integration and faster communication can be achieved.

3.15.1 Communication between two invocations of the QuLog interpreter

We will illustrate inter process communication via Pedro using two invocations of the QuLog interpreter. We assume Pedro has been installed and is running on the same host. Pedro is launched by entering the OS command pedro

in a terminal or command window.

In one terminal window we can then enter

qulog -A p1
and in the other

```
quelog -A p2
```

The `-A` flag tells the QuLog interpreter to connect with the Pedro server on the same host on its default port, and to register with the server the atom name following the `-A` as the process name. If there is no Pedro server running, or that name has already been registered by another process on this host, the launch of the interpreter will fail with an error message. Other command line options allow registration with a Pedro server on another host, and/or using another connection port. Processes on different hosts can register the same name with Pedro. Names just have to be host unique.

The other option for connecting and name registering with a Pedro server is to use the action primitives `connect_to_pedro, register_with_pedro` after the interpreter has been entered. If a QuLog or TeleoR process is only going to make use of the Pedro publish/subscribe services, only connection is needed. No name needs to be registered.

In the QuLog or TeleoR interpreter, the main query/command handling thread is the default message handling thread. This can be changed to another thread using the action primitive `set_default_message_thread(atom)`. We shall start by illustrating communication between main interpreter threads.

In the `p1` terminal enter:

```
| ?? started(p1) to p2; Msg1 from Sndr1.
```

In the `p2` terminal enter:

```
| ?? started(p2) to p1; Msg2 from Sndr2.
```

You should immediately see the answers:

```
Msg1 = started(p2) : term
Sndr1 = p2@localhost : process_handle

Msg2 = started(p1) : term
Sndr2 = p1@localhost : process_handle
```

displayed in the respective terminals.

The terms `p2@localhost, p1@localhost` have `process_handle` type. An `agent_handle` is a `process_handle` or an `atom`. In the message sends the destination agent process is identified just using the `atom` names `p2` and `p1`. This reduced form of handle can only be used if the receiving process is on the same host as the sender. `Msg to Nm` is shorthand for `Msg to Nm@localhost`.

Message sending is asynchronous. The sending thread immediately moves on to the next action. Sent messages are not stored. The sending thread can optionally remember ground sent messages in dynamic facts.

To send a message to an agent with Pedro registered name `Nm` on another host `HostName`, we must use `Msg to Nm@HostName`. `HostName`
can be a singly quoted IP address, or a singly quoted host name such as 'zeus.doc.ic.ac.uk'. This can be simplified to zeus if the sender's host is also in the doc.ic.ac.uk domain.

Sending a message to _@localhost, where the process name is given as the anonymous variable _, will send it to the default message handling thread of all processes running on localhost that have registered a name with the sender's Pedro server. Sending a message to robot_agent@, will send it to the default message handling threads of all processes, on any host, that have registered the name robot_agent with the sender's Pedro server. They must all be running on different hosts.

The type of the to primitive action is

\[
\text{_MsgTerm:}@ \text{ to RecrPtn:}??\text{agent\_handle}
\]

with the constraint that the agent handle argument cannot just be an unbound variable. Like the message term first argument, it will not be instantiated by the action.

The type of the message receive primitive is

\[
\text{_MsgPtn:}??\text{term from SndrPtn:?agent\_handle}
\]

A thread executing a from call will suspend until a message appears in the thread's message queue that is an instance of the _MsgPtn template term, such that the sender's handle is an instance of the _SndrPtn template term. It is then removed from the queue. All messages that preceded it in the queue, and which were skipped whilst searching for the matched message, remain. They can be matched against and removed at a later time.

The _SndrPtn pattern can take the form Nm@localhost to make sure the sender is a local agent process, or it could be bottle_collector@Host to make sure the sender has registered the name bottle_collector with Pedro, but may not be on the same host as the receiving process. To use just bottle_collector as the _SndrPtn is to insist the sender is running on the same host with Pedro registered name bottle_collector. It is the same as using bottle_collector@localhost.

### 3.15.2 Dedicated message handling thread

The TeleoR agent shell that we will introduce in Chapter 4 forks a thread to handle messages. This is made the default message handling thread. We shall do the same using the following handle_messages() action.
act handle_messages(), handle_message(??term,agent_handle)
handle_messages() ~> 
  Msg from Sndr; 
  handle_message(Msg,Sndr); 
  handle_messages()

| ?? fork_light handle_messages() as message_handler; 
  set_default_message_thread(message_handler).

handle_message has argument types (??term,agent_handle) because from 
has type ??term from ?agent_handle. Any QuLog ground or template might 
be received as a message. If the message term contains variables it is copied 
as part of the communication process\(^4\) so will contain a fresh set of local 
variables of the receiving thread.

Suppose that the agents controlling the four robots inform a location track-
ing QuLog agent whenever their robot moves through a coloured doorway. 
The tracking agent starts knowing the location of each robot, which it up-
dates whenever it receives a passing_through(robot,door) message from 
the robot’s agent. It replies by telling the robotic agent the new room location 
of its robot.

This can be achieved using two rules to define handle_message for the 
tracking agent:

handle_message(passing_through(Rob,Door),RobAg) ::
  type(Rob,robot) & type(Door,door) ~> 
  update_loc(Rob,Door,NewRm); 
  new_loc(NewRm) to RobAg

handle_message(Msg,Ag) ~> 
  write_list([Msg," from ",Ag," ignored", _nl]).
  % Ignore all other messages

The type tests for Rob and Door in the guard of the first rule are needed 
as the message argument of handle_message has type ??term. Rob and Door 
need to be a valid robot name and door colour for the call to update_loc. 
Without the ground value type tests the QuLog checker can only infer that 
Rob and Door both have type term, and may not be ground. The call to 
update_col would then be flagged as both type and mode incorrect.

\(^4\) The message term is actually converted to a string, with all occurrences of each variable 
replaced by the same variable name, for routing via Pedro. On receipt, it is converted back 
into a term with a new local variable replacing each occurrence of a variable name.
There is an alternative to having such ground value type tests in a guard of each handle_message rule. We can define a new type message_term that covers all the expected message terms. We then have handle_message first check that the term message argument is a ground message of that type. It ignores all other messages and calls a handle_message_term that deals with the expected messages. The program structure for this approach is

```
def message_term::= passing_through(robot,door) | ....

act handle_message(??term,agent_handle)
handle_message(Term,RobAg) :: type(Term,message_term) ~> 
    handle_typed_message(Term,RobAg)
% Check that Term is a ground message of type message_term
% and handle the message knowing this is the case
handle_message(Msg,Ag) ~> 
    write_list([Msg," from ",Ag," ignored", _nl]).
% Ignore all other messages

act handle_message_term(message_term,agent_handle)
handle_message_term(passing_through(Rob,Door),RobAg) ~> 
    update_loc(Rob,Door,NewRm);
    new_loc(NewRm) to RobAg
...
```

We shall usually uses explicit type checks in each of a sequence of action rules for handle_message. It is the more general method as it allows us to have rules dealing with both ground and non-ground messages.

### 3.16 Information agents

The tracking agent is acting as an information resource for the robotic agents. It has the static map of the environment and keeps track of the location of all the robots providing their agents keep it informed about their robot’s movements through coloured doorways. We can build on this to allow the robotic agents, or a human, to query the location tracker about the current room location of any robot. It could also be asked to give a path to get a robot from its current location to a specified room, or a room with a battery charger not currently being used. The path could be a list of the rooms and doorway colours through which the robot has to move.

To be able to return such a doorway path, the tracker needs static information about charger room locations, and dynamic information about their
current use. So, it might have facts for a static relation \texttt{charger\_room(room)}, and maintain facts for a dynamic relation \texttt{using\_charger(robot,room)}.

Assume that as well as \texttt{passing\_through} messages the robotic agents also send \texttt{reserve(robot)} and \texttt{release(robot)} messages. The first will cause the tracker agent to find a nearest room \texttt{CgrRm} with a charger not in use, reserve it for the requesting agent, and return a shortest path to the charger room. The tracker needs three more \texttt{handle\_message} rules before the ignore message rule.

```prolog
handle\_message(reserve(Rob),RobAg) ::
type(Rob,robot) &
path\_to\_nearest\_free\_charger(Rob,CgrRm,Path) ~> 
  remember using(Rob,CgrRm);
  use\_charger(CgrRm,Path) to RobAg

handle\_message(reserve(Rob),RobAg) ::
type(Rob,robot) ~> 
% All chargers currently in use. Fork light thread that
% waits for one to become free in order to reserve
fork\_light
  {wait charger\_room(CgrRm) & not using(_,CgrRm)
    retry [-using];
    ? path\_to\_nearest\_free\_charger(Rob,CgrRm,Path);
    remember using(Rob,CgrRm);
    use\_charger(CgrRm,Path) to RobAg}
as _

handle\_message(release(Rob),_) :: type(Rob,robot) ~> 
  forget using(Rob,_) 
% The forget will succeed even if there is no using\_charger fact to forget
```

The rooms are nodes in a graph and the door colours are the names of the links in the graph. We can therefore define \texttt{path\_to\_nearest\_free\_charger} using the general purpose \texttt{graph\_path} relation of Section 3.10. The Section 3.13 \texttt{open\_connected} relation defines the graph. \texttt{free\_charger(room)} is the test relation that defines a terminal node for a path to be found using \texttt{graph\_path}.

```prolog
dyn using(robot,room)
def through\_term ::= through(door,room,room,compass\_dir)
```

65
Suppose that a robot that requests a charger reservation is currently in room 6, that the nearest free charger is in room 15. Using fixed facts such as the connects facts of Section 3.13, and the latest dynamic beliefs about closed doors, the reply will be something like

```
use_charger(15, [through(red,11,15, south),
  through(green,10,11,east),
  through(red,6,10,south)])
```

containing the reverse path from room 6 to room 15. In Section 4.5 we shall give a recursive TeleoR procedure for following such a path. We shall then see that we need it to be returned as a reverse order path.

### 3.16.1 Updating the facts about closed doors

We will assume that the robotic agents update the tracker regarding which doorways of a room their robot has entered are ‘seen’ to be shut or open by sending status(Door,Rm,St) messages, where St is shut or open.
Suppose that doorways reported as shut are recorded by the tracker as two 
closed(door,room) facts, one for each room the door connects. We need two 
extra handle_message rules.

<table>
<thead>
<tr>
<th>dyn closed(door,room)</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_message(status(Door,Rm,shut),_) ::</td>
</tr>
<tr>
<td>type(Door,door) &amp; type(Rm,room) &amp;</td>
</tr>
<tr>
<td>connected(Door,Rm,OthrRm,_) ~&gt;</td>
</tr>
<tr>
<td>remember closed(Door,Rm)</td>
</tr>
<tr>
<td>remember closed(Door,OthrRm)</td>
</tr>
<tr>
<td>handle_message(status(Door,Rm,open),_) ::</td>
</tr>
<tr>
<td>type(Door,door) &amp; type(Rm,room) &amp;</td>
</tr>
<tr>
<td>connected(Door,Rm,OthrRm,_) ~&gt;</td>
</tr>
<tr>
<td>forget closed(Door,Rm);</td>
</tr>
<tr>
<td>forget closed(Door,OthrRm)</td>
</tr>
<tr>
<td>% Any remembered closed facts need to be forgotten</td>
</tr>
</tbody>
</table>

3.17 Answering queries sent as lists of call terms

The tracker agent sends replies to moving_through and reserve messages. 
These are queries to the agent sent as ground messages. More generally, we 
might want to query an information agent using an open set of queries that 
are non-ground ‘conjunctions’ of calls to a some set of relations that other 
agents are allowed to query. Such queries can be sent as lists of relcall 
terms.

relcall is the QuLog compiler generated meta-type, a sub-type of term, 
that includes every term with a functor that is the name of a primitive or 
program defined relation, with argument terms of types that are any allowed 
argument types for a call to the relation. For the tracker agent program the 
relcall type includes terms of the form

loc(robot,room), connects(door,room,room,compass_dir),
connected(door,room,room,compass_dir),
open_connected(door,room,room),...

and all terms denoting type correct calls to QuLog built-in relations.

To query the tracker agent to find the names of all the robots currently 
located in a room with at least two open doors, together with the identity of 
the room, we might send a query term
ask_all(loc_query, (Rob,Rm),
[location(Rob,Rm), open_connected(D1,Rm,1),
open_connected(D2,Rm,2)],\=(D1,D2))

where the variables in the non-ground list of relcall terms will the given values when the query list is evaluated. loc_query is a query identifier that will be sent with the answer.

A relcall term is evaluated as a relation call using the QuLog call primitive, similar to but in one key feature different from the Prolog call primitive. The latter accepts any term as argument but will give a run-time error if the term is not compound, its functor does not name a program or built-in relation, and where the relation is built-in, if it does not have the arguments that must be given of the required type. In QuLog such run-time errors cannot occur as the compiler will check that for any call C use C has type relcall and that call C is preceded by a runtime test ground_inputs(C).

A type test type(C,relcall), which is the same as type(C, !relcall), will check that C is a ground relcall term. A type test type(C, ?relcall) will check that C is either an unbound variable or a compound term that is a template or ground relcall term. What we need to check before using call C is that C is a template or ground relcall term which is mode correct - that all its unput arguments - those that must be given as ground terms - are indeed ground. That is what the primitive ground_inputs(??relcall) checks.

Below is a rule for the tracker's handle_message(??term, agent_handle) action that will accept ask_all query terms. We also give a definition for a relation eval(list(??relcall)) that recursively evaluates a list of terms that are variables or type correct relcall terms, and which are mode correct relation calls at the point that they are evaluated using the call ??relcall primitive.

```
handle_message(ask_all(QId,Ans,QList),Agent) ::
type(QList,list(?relcall)) & type(QId,atom) ~> 
fork,light ans(QId,[Ans::eval(QList)]) to Agent as Th;
remember query_thread(QId,Agent,Th,now())
% Remember query thread name and when it was forked

dyn query_thread(QId:atom,Ag:agent_handle, Th:atom,Time:num)
"Th is the light weight query thread forked at time Time
to answer the query with identifier QId from Ag"

rel eval(QList:list(??relcall))
"QList must be a complete list of variables or relcall terms
and may not be ground after the eval call"
```
If the tracker is sent the above ask all query about robot locations it will pass the type test type list(?relcall) of the message handling rule. At the point that the call D1\=D2 is to be evaluated there is an extra run-time check that D1 and D2 have values since \= requires its arguments to be ground. More generally, if the query list of an ask all query does not pass the extra check during evaluation, the call to eval inside the list comprehension will fail and ans([[]]) will be sent as an answer to the query.

The above approach can be extended to allow query lists containing neg(relcall) terms to indicate a negated condition where at the time of evaluation the relcall argument is ground.

def negCall ::= neg(relcall)
def queryCond ::= relcall || negCall

handle_message(ask_all(QId,Ans,QList),Agent) :::
  type(QList,list(?queryCond)) & type(QId,atom) \->
  fork_light ans(QId,[Ans::evalN(QList)]) to Agent as Th;
  remember query_thread(QId,Agent,Th,now())

rel evalN(QList:list(??queryCond))
"QList must be a complete list of variables or queryCond terms and may not be ground after the evalNaF call"

evalN([[]])
evalN(neg(C)) :: type(C,relcall) <=
  not call C &
  evalN(Calls)
% neg(C) holds of call C fails - the negation-as-failure
% evaluation of negated calls, C a ground relcall term.
evalN([C,..Calls]) :: ground_inputs(C) <=
  call C &
  evalN(Calls)

This allows the query about robot locations to be sent as

ask_all((Rob,Rm),[loc(Rob,Rm),open_connected(D1,Rm,_),
  open_connected(D2,Rm,_),neg(=(D1,D2))])
The calls to `eval` and `evalN` are done inside forked light weight threads to insulate the message handling thread from a possible non-terminating evaluation. That is also why the thread name and the time of forking (`now()` returns the current time) are remembered as `query_thread` facts. To ‘garbage collect’ any potentially non-terminating query threads we can fork a call to the following `monitor_query_threads()` action. This kills any query thread that has not terminated naturally inside 1.1 seconds. This time is more than enough for any query thread to find its required answer. It sends a message informing the agent who sent the query, identified by its `QId`, that this query has been terminated.

\[
\text{act monitor_query_threads()}
\]
\[
\text{monitor_query_threads() :: query_thread(_,_,_,_) } \rightarrow
\]
\[
? \text{ Now=} \text{now();}
\]
\[
\text{forall QId,Ag,Th,ForkT (}
\]
\[
\text{query_thread(QId,Ag,Th,ForkT) } \& \text{ Now}>\text{ForkT+1 } \rightarrow
\]
\[
\text{exit_thread(Th); query_terminated(QId) to Ag;}
\]
\[
\text{forget query_thread(QId,Ag,Th,ForkT));}
\]
\[
\text{sleep(0.1)};
\]
\[
\text{monitor_query_threads()}
\]
\[
\text{monitor_query_threads() } \rightarrow
\]
\[
\% \text{ No current query_thread facts}
\]
\[
\text{sleep(1)};
\]
\[
\text{monitor_query_threads()}
\]

The above `forall` is an iterated action. That is why the action operator `~>` is used between the generating query and the action sequence. `sleep(T)` will cause the thread that executes the action to become inactive for `T` seconds.

### 3.18 Publish/Subscribe Pedro Communication

As well as addressed message forwarding using registered names a Pedro server supports message forwarding using subscriptions for messages of interest to an agent. Agents do not have to register a name to make use of this service, they just have to connect to a Pedro server using the `connect_to_pedro` system action. However, using both forms of communication is useful. An agent can lodge a subscriptions for notifications containing requests for and advertisements for services that contain the notifiers agent handle. When a such a notification is received by the subscriber, the message interaction between notifier and subscriber can then continue using addressed communication.
The tracker agent we described in the last section offers a centralised service for a set of robotic agents each of which must inform and query the tracker agent. A more distributed approach would have each robotic agent able to construct a path to a charger using a map of the environment and the latest information about charger use and shut and open doors. To keep each agent up to date we can use Pedro subscriptions and notifications.

3.18.1 Pedro subscriptions

A Pedro subscription is a string of the form

"MsgPtn" or "MsgPtn::PrologQuery"

(MsgPtn is a Prolog term. The PrologQuery uses comma instead of & and can use the Prolog disjunction operator ;. The conditions in the conjunction must use a restricted set of predicates supported by Pedro. They typically use variable names appearing in MsgPtn to specify constraints on the values inside the message instance of MsgPtn notified to Pedro.

The allowed predicates include = unification, the <, >, =<, >= arithmetic inequalities where the arguments can be arithmetic expressions, Prolog type tests number(N), atom(A), string(S) and list(L), and the list membership relation member(E,L). In addition not and once can be applied to a (...) bracketed call or Prolog query. The full list of the allowed predicates, and the syntax of the allowed subscription conditions, are in the Pedro manual.

Here are some example subscription strings:

"new_loc(Rob,_)::member(Rob,[robot1,robot4])"
"using(_,_)
"status(_,_,closed)"
"temperature(room1,T)::T>25;T<10"

The first subscription string would be used in order to receive any notification term that gives the location of the robot1 or robot2 robot. The second would be used to receive all notifications about use of a charger room, and the third to receive all notifications giving the open status of a door in a room. The last one is for notifications about the temperature of room1 that are greater than 25 or less than 10.

To lodge a subscription with the connected Pedro server the action primitive subscribe string as ?nat is used. For example

subscribe "status(Door,Rm,Status)" as Id

---

5 This is the same as a QuLog term except that no argument compound terms of the form f() are not allowed.
Here \texttt{Id} must be an unbound variable. It will be given a unique natural number value if the subscription string has the correct syntax and is accepted by the Pedro server. If not, an error is raised.

The \texttt{Id} value returned can be used to remove the subscription with a later unsubscribe \texttt{Id} action. An agent can update its subscriptions as its concerns change.

The string of the subscription action can be determined by a ++ string expression that uses the \texttt{QuLog str(term)} primitive function. \texttt{str} will map any ground term into the string denoting the term. Suppose that the names of the robots for which some agent wants to receive all notifications concerning location are recorded in a dynamic relation \texttt{monitor(robot)}. The subscription action would be:

\begin{verbatim}
subscribe "new_loc(Rob,_) :: member(Rob," ++ str([R :: monitor(R)]) ++")" as Id1
\end{verbatim}

which would become

\begin{verbatim}
subscribe "new_loc(Rob,_) :: member(Rob,[robot1,robot4])" as Id1
\end{verbatim}

if these were the two robots for which location tracking was wanted.

\subsection*{3.18.2 Pedro notifications}

Notifications are sent as normal message terms, unlike subscriptions they are not sent as strings.\textsuperscript{6} The normal message send action of QuLog is used with receiver handle \texttt{pedro}. Examples are:

\begin{verbatim}
new_loc(robot4,room3) to pedro
status(green,room5,shut) to pedro
\end{verbatim}

Towards the end of Section 3.14.1 we had an example of a forked non-terminating thread within the tracker agent that wrote a terminal message every time the tracker’s fact recording the location of a particular robot was replaced by a new one recording a different room location. We can now modify that thread so that a notification is sent to Pedro instead. The action that is forked is:

\textsuperscript{6} As with addressed messages, they will be converted to strings before being sent to Pedro and converted back to QuLog terms on receipt by a subscriber.
This way all the robotic agents are kept up to date regarding the room location of each robot.

3.18.3 Receiving a notification

All notifications covered by a lodged subscription appear as message terms in the message queue of the thread of the QuLog agent process that lodged the subscription. The sender handle of the message is pedro. This tells the receiver it is a forwarded anonymous notification. Only if the message term includes the sender’s agent handle will subscribers that receive the notification know the Pedro identity of the notifier. Indeed, since notifiers do not need to register a name with Pedro, the notifier may not have a Pedro identity.

For a QuLog agent we can have all the notifications go to the forked message handling thread where this thread lodges subscriptions before it enters its message processing recursive loop. The structure of the message handling action that is forked and made the default message handler will therefore be:

```prolog
act notify_loc_changes(robot,room) 
notify_loc_changes(Rob,Rm) ~> 
    wait loc(Rob,NewRm) & NewRm \= Rm retry [+loc]; 
    new_loc(Rob,NewRm) to pedro; 
    notify_loc_changes(Rob,NewRm)
```

The returned subscription identifier is remembered, either associated with the subscription string or some other symbolic identification, so that it can
be retrieved at a later stage if the subscription needs to be removed. This can be done as part of the handle_message response to some message. New subscriptions can also be lodged in such an action.

In Section 6.9 we shall illustrate the use of Pedro’s publish/subscribe feature by re-doing path finding to a free charger room and charger room reservation without a centralised information agent. Each robotic agent will have its own relation map of the environment which it uses to keep track of the room in which its robot is located. The agents continually revise the connectively of their map as a result of their own observations of open and shut doors and notifications they receive about seen doors from other agents. They also know which charger rooms are in use, when they become free, and the room locations of each robot and the path they are currently following, if they are following a path, because of notifications.

When a robot needs to be re-charged, its agent uses the path following program of Section 4.5 to find the shortest path of open doors to a free charging room. It then bids for the use of the charger room, and informs all the other agents of its intended path to the charger, by sending a

\texttt{path(robot,door\_path)}

notification. If two agents try to bid for the same charger room at more or less the same time, one bid will get to all the agents before the other as Pedro serialises the forwarding of notifications. The first path notification to arrive with destination a particular charger room wins. The other agent will look for a shortest path to another free charger room when it receives its own path notification after that of the winning agent.
Chapter 4
Recursive Procedures, Relation and Procedure Hierarchies
We will introduce the concept of rule and procedure hierarchies, as Nilsson did, by first giving his robot arm block tower building program from [63], which we have slightly modified.

- It makes essential use of Belief Store rules as developed in Chapter 3.
- It is a very good example of the robustness and task specific ‘intelligence’ of Teleo-Reactive programs.
- It is an example of a TeleoR program that uses recursive and mutually recursive procedures.

The program is given, and its behaviour explained, in Sections 4.1 and 4.2. In Section 4.3 we describe how you can start an agent using the TeleoR agent shell performing a tower building task using this program, supplied with the software, which interacts with a Python graphical simulation. You can interact with the agent by pausing the simulation and moving blocks around to either help or hinder the agent. If you help by moving the next block that it would be placing onto a partially built tower onto the tower, it will immediately move to pick up the next but one block, or suspend its activity if the tower is now complete. Similarly, if you remove a block from its partially or completely built tower, it will abandon whatever is was doing to pick up and replace the displaced block, uncovering it if need be. This robust behaviour is a consequence of the testing of the rule guards of all the active TeleoR procedures on each significant change in the perceived state of the agent’s environment, with the guards of the rules of each parent TeleoR call always re-evaluated before those of its child call, to check if the child call should be aborted. In Section 4.4 we show how the agent shell can be extended so that the tower building agent can be sent requests to build a tower, or to abandon the building of a tower, from another agent or process.

We follow this by giving another recursive TeleoR program. This is a program that will control a mobile robot following a path through doorway connecting rooms to get to a destination. The path is a list of (door, room, room) tuples as returned by the path_to_nearest_free_charger relation we defined in Section 3.16. Recall that this returns the path in reverse order and this is what enables us to give a recursive TeleoR program. The agent shell is then extended to use this program with message handling, the lodging of Pedro subscriptions when it is launched, and with enhanced percept handling.

We conclude the Chapter with a discussion of memory architectures for Teleo-Reactive agents starting with Nilsson’s triple tower architecture of [63].

**Agent architecture**

The agent architecture we assume for this chapter is as depicted in Figure 4.1. It comprises at least three threads. The role of the precepts handling and TeleoR evaluator threads are as in the two thread architecture of Figure 1.1 of Chapter 2 except that the evaluator thread will also react to updates of the Belief Store made by the new message handling thread. The role of
this thread is to respond to each of a stream of messages. If the message is an information message the response is usually an atomic update of the Belief Store. If it is an enquiry message the response is usually a query of the Belief Store and the sending of a response. If it is a task request to evaluate a TeleoR procedure call, and no task is currently executing, the evaluator thread is started executing that procedure call.

In addition there may be any number of Belief Store monitoring threads there send out messages when certain percepts or other dynamic beliefs appear in the store, or which add extra dynamic facts. Our path following agent will have two such extra threads.

The agent is implemented by configuring a generic agent shell as we shall describe in Sections 4.3 and 4.4.

4.1 Block Tower Building Agent

We will use the type declarations of Chapter 3, its definitions of stack and clear, and its test only definition of tower. The top level TeleoR procedure of the tower building task is the recursive procedure:
percept holding(), on(block,block), on_table(block)

tel makeTower(list(block))
makeTower(Blocks){

tower(Blocks) ~> () % Blocks are configured as a tower

stack(Blocks) & Blocks=[Block,... ] ~> unpile(Block)
% Blocks will be a tower if Block is cleared

Blocks=[Block] ~> move_to_table(Block)
% Move Block to the table to make a one block tower

Blocks=[Block,TopBlock,...MoreBlocks] &
tower([TopBlock,...MoreBlocks]) ~> 
    move_to_block(Block,TopBlock)
% Blocks one block short of being a tower move Block onto TopBlock

Blocks=[_,...MoreBlocks] ~> makeTower(MoreBlocks)
% Recursively configure MoreBlocks as a tower
}

which only uses defined relations. The tower to be built is specified by list of integer block values Blocks where adjacent blocks on the list Blocks must be adjacent blocks in the tower, with the first block on Blocks being the top of the tower. unpile, move_to_block and move_to_table are auxiliary TeleoR procedures. Firing of the last rule will normally eventually result in the guard of the fourth rule being inferable. Firing of the second, third or fourth rules will normally eventually result in the guard of the first rule being inferable. So the procedure satisfies the regression property and there is always a rule that can be fired.

As durative actions we shall use pickup(Block) which will pick up a clear block, put_on_block(Block) which will put down a held block on top of Block if it is clear, and put_on_table() which will put down a held block on a free space on the table. We assume there is always a free space.

def durative::= pickup(block) | put_on_block(block) | put_on_table()
The procedure `unpile(Block)` moves all the blocks on top of `Block` onto the table. Note the mutual recursion between `unpile` and `move_to_table`. 
Our rules for move_to_block and move_to_table are in a slightly different order than as given in [63] so that they satisfy the regression property. For example, in move_to_block we have the rule that puts any other block that happens to be being held on the table as the last rule whereas Nilsson has it as the third rule. We have the explicit not holding(,) test in the two rules above the last rule, the sub-goal that is achieved by the put_on_table action of the last rule.

4.2 Brief Description of How the Program Behaves

Suppose the start call is makeTower([2,4,1,8]). If the agent is really lucky, the blocks are already configured as a tower and the first rule of makeTower can be fired. If it is just lucky, these blocks are on top of one another in sequence with 8 on the table, but there are blocks on top of 2. The second rule of makeTower then applies, and the blocks on top of 2 will be one by one moved onto the table. This is done by calling unpile(2). If block b1 is immediately on top of 2, move_to_table(b1) is called. If b1 is clear - no block on top - it is picked up. However, if there is a block b2 on top of b1, there is a call to unpile(b1), which in turn calls move_to_table(b2).

Suppose b2 is clear. It will be picked up by the firing of the third rule of the call move_to_table(b2). Before this call can fire its second rule to put b2 of the table, the ancestor call move_to_table(b1) will switch from having fired its fourth rule to firing its last rule (remember all ancestor calls remain active) as the guard not holding(,) of the fourth rule no longer holds. So the call move_to_table(b1) will put b2 onto the table. As b1 is now clear, move_to_table(b1) will next fire its third rule to pick up block b1, assuming it has not been covered again by outside interference. Before move_to_table(b1) can fire its second rule to put b1 on the table, makeTower([2,4,1,8]) will fire its first rule as [2,4,1,8] will now be inferably a tower from the latest batch of percepts as 2 will now be clear. The arm will be left holding block b1.

In Chapter 6 we shall see how small changes to the makeTower and unpile procedures will ensure that b1 is always put down on the table before the makeTower call can fire its first rule. We shall also see that another small change to the move_to_table procedure will ensure that it is the unpile(b1) call, hence the move_to_table(b2) call, that puts b2 on the table rather that its ancestor call move_to_table(b1). Where there are many blocks above the block that needs to be made clear this saves the repeated firing of the fourth rule

not holding(,) ~> unpile(Block)

of move_to_table, and the repeated exit and re-entry of its unpile(Block) call.
Let us now examine the other extreme, that no tail subsequence of the list \([2, 4, 1, 8]\) is already a tower, or a stack. The \texttt{makeTower} call recurses down using its last rule to the call \texttt{makeTower([8])}. As we have assumed no tail subsequence is a tower or stack, 8 must be buried inside some tower on the table or be the top block of a tower. The third rule of this last \texttt{makeTower([8])} call will fire, calling \texttt{move_to_table(8)}. The behaviour of this is similar to the unpiling behaviour discussed above. All the blocks on top of 8, if there are any, will be recursively moved to the table until 8 is clear, when it will be moved to the table.

Immediately 8 is on the table, parent call \texttt{makeTower([1,8])} can fire its fourth rule. This has the action \texttt{move_to_block(1,8)}. If need be 1 is made clear by a call to \texttt{unpile(1)}. It is then picked up and put down on 8 by \texttt{move_to_block(1,8)}. Two more uses of rule 4, in calls \texttt{makeTower([4,1,8])} and \texttt{makeTower([2,4,1,8])}, possibly requiring calls to \texttt{unpile} if blocks 4 and 8 are not clear, will complete the tower resulting in the firing of rule 1 of the initial \texttt{makeTower} call.

The intermediary situation is where a tail sub-list of \([2,4,1,8]\) is already a tower or is a stack with blocks on top. In that case the initial \texttt{makeTower} call recurses until a \texttt{makeTower} call can fire its rule 4 or rule 2. If it is rule 4, the tail, say \([1,8]\), is a tower. The behaviour is then the recursion exit behaviour just discussed to move blocks 4 and 2 to complete this partial tower. If rule 2 is fired, the tail, again say \([1,8]\), is a stack. It has all blocks covering 1 moved to the table as in our first scenario. Then rule 4 of the \texttt{makeTower([4,1,8])} call will be fired, followed by a rule 4 firing of \texttt{makeTower([2,4,1,8])}. This is all assuming there is no interference or help.

4.3 Incremental Development and Testing of TeleoR Programs

In Chapter 3 we exemplified the use of the QuLog interpreter for developing and testing Belief Store defined relations. The next stage is the adding of TeleoR procedure definitions to the program file with incremental testing of these procedures using real robotic resources or a simulator.

To test a program file that contains TeleoR procedure definitions you need to first launch the Pedro communications server using the terminal command

\texttt{pedro}

Then we launch the TeleoR extension of the QuLog interpreter using the terminal command

\texttt{teleor -Agent_name}

81
agent_name is the name that will be registered with the Pedro server for
the agent which you will launch to test your TeleoR program. Use a name
such as builder. The command line prompt of the teleor extension is \|
?~>. The extension has all the capabilities of the QuLog interpreter as
described in Chapter 3, and extra functionality specific to launching and testing
agent control programs comprising TeleoR procedures.

Let us suppose that the TeleoR procedures given in this chapter have been
added to the program file towerBS.qlg of Chapter 3. Such a file with the
name bookTowerChap4.qlg is provided with the QuLog+TeleoR software in
the towers sub-directory of the examples directory. The only difference is
that it has 16 block labels not 9. There is also a Python simulation of the
robotic arm and blocks environment bookSimChap4.py in the towers direc-
tory. In a separate terminal launch this with a command

python bookSimChap4.py

The Python process will register the name bookSimChap4 with the Pedro
server. This is used for communication from the agent to the simulator via
Pedro.

Inside the teleor interpreter consult the file bookTowerChap4 and launch
an agent linked with the simulator process with the commands:

Consulting bookTowersChap4...
.... bookTowersChap4 consulted
success

Starting messages thread
Starting percepts thread
Waiting for initial percepts from bookSimChap4
Received initial percepts from bookSimChap4
success

The agent starts by sending an initialise, message to the simulator to
get the initial set of percept facts. updates is the communication convention
used to send the percepts from the simulator. It tells the agent shell that only
the changes to percepts will be sent each time there is a change. So, if the
arm is used to pick up a block 2 from on top of block 4, what is received by
the agent is the list \[r\_holding(arm,2), f\_on(2,4)\] where the r_ signals
remember and the f_ forget. All the simulators provided with the software
use this convention. The other convention is signalled by all. In this case all
the percept facts are sent each time there is a change, or at set intervals. The
agent then forgets any percept fact not in the list sent, and remembers each
new percept not already believed. It is the simpler convention to implement
on a real robot but requires more to be communicated when most percepts stay the same.

To see the current percepts in the agent’s Belief Store enter the command

| ?~> bs.

You will have displayed a set of on table facts reporting that every block is on the table. By hitting the space bar when the simulator window is selected you can move blocks around using your mouse or trackpad. You can build towers. Hitting space bar again will cause the simulator to send an update to the agent recording the new configuration. Another bs command will display the agent’s new percept beliefs.

You can now directly enter actions to be sent to the simulator using an action command. This is particularly useful if you are interacting with a real robot via an interface process, which like the simulator must be connected to the same Pedro server as the agent. Documentation with the Pedro server will tell you how to do this for an interface process written in C/C++, Java or Python.

| ?~> action(pickup(3)).

will pick up block 3 if it is clear.

| ?~> action(put_on_table()).

will put down the held block on a free space on the table.

When testing a simulation or a robot interface process it is a good strategy to test the action repertoire first with the action command. The next stage is to work bottom up on the TeleoR procedures. Stack some blocks on top of block 2 by interacting with the simulator and then try:

| ?~> start_task(t, unpile(2)).

All the blocks on top of 2 will be picked up and all except the last block will be put down on the table. The last block will just be held by the arm because unpile will fire its first rule clear(2)~>() as soon as block 2 is clear. To get this block on the table enter:

| ?~> kill_task(t).

| ?~> action(put_on_table()).

Now try a command such as:

| ?~> start_task(t, makeTower([2,5,1,9])).

By pressing the space bar in the simulator window suspend the activity and interact with the tower builder by helping or hindering. It will always respond appropriately.

If we wanted to have extra threads inside a started agent we just fork them using fork .. as .. action commands at the TeleoR interpreter prompt.

83
4.3.1 Logging an agent’s behaviour

You can get a complete log of the the behaviour of any launched task by using the logger.py Python tool. This is in the /bin directory of the QuLog software distribution. Launch the logger visualisation as a stand-alone process with a command:

```bash
logger.py log
```

This will cause a Logger:log window to be displayed. Now, in the terminal in which you started the agent enter:

```
| ?~> logging log.
```

Kill task `t` and enter a new `start_task` command to build another tower. In the logger window you will see a complete log of the behaviour of the agent. You will see the percept messages it receives, the state of its Belief Store immediately afterwards, which rules have been fired as a result of the percepts update, what action is determined, and which action control messages are sent to the simulator. It is a very useful tool for learning about and monitoring a TeleoR agent application.

4.4 Message Handling Configuration of the Agent Shell

As an alternative to starting and terminating tasks inside the Teleor interpreter we can send requests to the agent to start and to kill task threads. In a multi-agent application this is how agents would initiate tasks in other agents. One of the configuration options for the agent shell is the supply of a QuLog action procedure that is an application specific way of handling messages sent to the agent. All we need do is add a QuLog action procedure called `handle_message` to the program file. This has the same role for a TeleoR agent as the example `handle_message` action we defined in Section 3.16 for the QuLog tracker information agent. It has the same two arguments: a `term` argument which is the message received, which may not be ground, and an `agent handle` argument which is the Pedro identity of the sending agent or process to which replies can be sent. If `handle_message` is not defined the default agent behaviour is to remove each message that arrives from the message buffer and to ignore it.

There is also a facility for adding extra processing to the agent’s percept handler, which we shall use in Section 4.6. The extra processing is done after the default percept updates performed by the percept handling thread.

A definition that allows the starting and termination of tower building tasks, one at a time is:
handle_message_(makeTower(Blocks),Agent) ::
  type(Blocks,list(block)) & not task_(makeT,_) ~> 
  start_task(makeT,makeTower(Blocks));
  task_started(makeTower(Blocks)) to Agent
handle_message_(makeTower(_),Agent) :: task_(makeT,_) ~> 
  a_task_is_running to Agent
handle_message_(kill_last_task,Agent) :: task_(makeT,_) ~> 
  kill_task(makeT);
  task_killed() to Agent
handle_message_(Message,Agent) ~> 
  cannot_handle(Message) to Agent

Task_ is a dynamic relation maintained by the agent shell. Each time a task is started a new task_ dynamic relation fact is added of type (atom,telcall). telcall is a program specific type generated by the TeleoR compiler that includes every term that denotes a type correct call to a TeleoR procedure. So, if the first handle_message_ rule processes a makeTower([3,5,1,8]) message task_(makeT,makeTower([3,5,1,8])) will be in the Belief Store. The type(Blocks,list(block)) test in the first action rule is to ensure that the makeTower(Blocks) message term denotes a ground type correct call to the makeTower procedure, as required by the start_task primitive. The from action in the agent’s message handling thread executed just before handle_message_ is call will already have checked that the Blocks argument of the makeTower(Blocks) message, if given, is at least a partial list of block values.

The above definition is included in the bookTowersChap4.qlg file. There is no need to declare type of handle_message_ as its type is specified in the agent shell program.

In a separate terminal launch a QuLog interpreter:

qulog -Arequestor

registering the name requestor for the process with the Pedro server. We can now communicate with our agent builder from that qulog terminal.

| ?? makeTower([2,5,1,9]) to builder; Reply from builder. 
Reply = task_started(makeTower([2,5,1,9])): term |

| ?? makeTower([7,6,2,8]) to builder; Reply from builder. 
Reply = a_task_is_running: atom |

1 We can just use the test ground(Blocks) as the message receive primitive will have checked that the argument of makeTower is type correct. But it will not also check that it is ground.
| ?? hello to builder; Reply from builder.
Reply = cannot_handle(hello): term

| ?? kill_last_task to builder; Reply from builder.
Reply = task_killed: atom

| ?? makeTower([7,6,2,8]) to builder; Reply from builder.
Reply = task_started(makeTower([7,6,2,8])): term

In Chapter 8, when we modify the makeTower program to allow the interleaving use of the arm to build several towers of different blocks, we shall give an alternative handle_message definition that allows several makeTower tasks to be running at the same time.

There is another configuration option for the message handler of the agent shell. In Section 3.18 the handle_messages action we defined started by lodging subscriptions with the Pedro server before entering the message processing loop. We can have the agent shell message handler do the same thing by defining a QuLog action init_agent() in the program file. This will then be called by the agent’s message handling thread before it starts accepting messages. As well as lodging subscriptions this initialisation action can remember dynamic beliefs and send out initial messages and notifications. We will use an init_agent action definition in Section 6.9.

4.5 Recursive Path Following

In Section 3.16 we described how the tracker agent for a group of robots moving through door connected rooms could use its logical map of the room connections to find the shortest path from a robot’s current room location to a free charger room. A robot Rob sends a reserve(Rob) message to the tracker and receives a reply such as

use_charger(15,[through(red,11,15, south),
through(green,10,11,east),
through(red,6,10,south)])

if the charger in room 15 is free and none of the doors on the path are currently believed to be closed. The door path to charger room 15 is given in reverse order.

Below is a recursive procedure for generating the sequence of actions to take a robot through a sequence of open doors given in reverse order as a door_path list. The destination room of the path is the second room of the first tuple of the list of through terms. As mentioned in Section 3.13, all doors are in the centre of room walls, all rooms are square, and straight metal lines
connect each door with the room centre. The doors are always due north, east, south or west of the room centre, i.e. at compass directions 0, 90, 180 or 270.

`follow_line_forwards()` action will guide a robot along one of these tracks. Action `turn_at_centre(turn_dir)`, if executed when the robot detects it is at the room centre, will position it exactly over the intersection of the metal lines and turn it on the spot to the left or right.

As we want the robotic agent guiding a robot along a doorway path to report to the tracker the open/shut status of all seen doors, we assume the robots have a fisheye camera pointing vertically. This has range of just over half the width of a room so that as a robot nears the centre of a room it will be able to see all doors in the room and detect whether they are open or closed.

The controlling agent gets percepts `see_door(door,door_status)`, where `door_status` is `open` or `closed`, for all the doors in its current room just before it reaches the centre of the room. However, as the robot approaches its exit door from a room, another door being opened or closed will not detected by the fisheye lens, as the door will be out of range of the camera. For this reason, and because certain rooms not have been entered by a robot for some time, the finder agent will not have completely accurate information about door status.

As the robotic agent has a path which tells it which room is on the other side of each exit door, providing it starts knowing the robot’s initial room location, stored in a dynamic fact `my_loc(room)`, it can update its own location as it moves through doors. It does not need to rely on the tracker agent to tell it into which room it is entering. Each agent will therefore send a `new_loc(robot,room)` notification, with the `robot` argument being its name, when their robot has just passed through the doorway to the next room on its path.

We can do this by tailoring the agent’s percept handler. Just as the message handler will call `handle_message` if defined in the `TeleoR` program file, so the percepts handler will call one or other of two specially named `QuLog` actions before it loops to wait for the next batch of percepts. This allows us to ‘infer’ and update extra perceptions such as `my_loc`, but also to communicate that there has been such an update with messages such as `new_loc`.

The `new_loc` notifications will be received by the tracker agent, and all the robotic agents including the notifier, if each has lodged a subscription for such a message with Pedro. There is no need for the robotic agents to tell the tracker they are passing through doorways, as was assumed in Section 3.15.2.

As well as percepts about open and closed doors, each robotic agent also receives percepts that tell it that: its robot is facing towards the centre of a room, is at the centre of a room, is in a doorway of a particular colour, is pointing in a certain 360 degree compass direction. The first three percepts are obtained by image processing of a forward facing regular camera, the last via a digital compass on the robot.
def room ::= 0..15
def compass_dir ::= 0..359
def door ::= red | green | blue | yellow
def robot ::= robot1 | robot2 | robot3
def door_status ::= open | shut
def turn_dir ::= left | right
def through_term ::= through(door,room,room,compass_dir)
def door_path == list(through_term)

percept see_door(door,door_status), at_room_centre(),
     see_centre_ahead(), in_doorway(door),
     pointing(compass_dir)

dyn my_name(robot)
dyn my_loc(room)
dyn path_to_charger(door_path)
dyn loc(robot,room)
% The first two are initialised when the agent is started. loc facts
% are updated each time the agent receives a new loc message
% The my_path fact is remembered when the tracker agent gives
% this robotic agent is shortest path to a charger room.
durative follow_line_forwards(), turn_at_centre(turn_dir)

tel follow(door_path)
follow(Path){
    Path=([through(_,_,OutRm,_)],[..]) &
         my_loc(OutRm) & at_room_centre() ~> ()
    % Robot has reached the centre of the destination room of Path
    Path=([through(_,_,OutRm,_)],[..]) & my_loc(OutRm) ~>
         follow_line_forwards()
    % In destination room OutRm of Path, but not at its centre
    Path=([through(Door,InRm,OutRm,Dir),[..]) & my_loc(InRm)~>
           find_and_go_through_doorway(Door,Dir,OutRm)
    % Go to centre of InRm, turn towards direction Dir of the
    % exit door of InRm. Go through Out door if not shut.
    Path=([..PathToInRm]) ~> follow(PathToInRm)
    % Go through the doors that will get the robot into InRm
}
tel find_and_go_through_doorway(Door, compass_dir, Room)

find_and_go_through_doorway(Out, OutDir, OutRm)
{
    pointing(OutDir) & see_door(Out, open) &
    not loc(_, OutRm) ~>
    follow_line_forwards()

    at_room_centre() & pointing(CurrDir) ~>
    turn_at_centre(turn_dir(CurrDir, OutDir))
    % When and while at room centre keep turning towards OutDir

    see_centre_ahead() ~> follow_line_forwards()
    % Follow the line from entry door to room centre

    true ~> ()
    % Pause if Out door is closed or there is a robot in OutRm
}

fun turn_dir(compass_dir, compass_dir) -> turn_dir

    turn_dir(From, 0) :: From < 180 -> left
    turn_dir(From, 90) :: From > 90 & From < 270 -> left
    turn_dir(From, 180) :: From > 180 & From < 360 -> left
    turn_dir(From, 270) :: From > 270 -> left
    turn_dir(From, 270) :: From < 90 -> left
    turn_dir(_, _) -> right

You may be wondering why the find_and_go_through_doorway auxiliary procedure has a last rule that will get the robot to the centre of the room it is in when the main follow procedure's second rule does the same thing. The reason is that the first two rules of the follow procedure are only fired after the robot has passed through the last door of the path it is following into its destination room.

For all earlier rooms in the path, including the last but one room, as soon as the robot passes through the open Door into OutRm, the controlling agent will send a new_loc notification and update its my_loc belief. Immediately after that update, the parent follow call will fire its third rule to find and go through the exit doorway of OutRm, since this is not the destination room of the path. That is why the doorway procedure must start by getting the robot to the centre of the room just entered.

We assume there is an initial TeleoR procedure call that controls the robot until it detects its battery is low. In Section 6.9 we shall give such an initial TeleoR procedure, but for a fully distributed control of the path following robots in which there is no central tracking agent that contains
all the information about closed and open doors and connectivity of rooms. Through communication, the robotic agents maintain a shared model of their environment: which doors are closed, where each robot is currently located and what path each is following, if any. Each constructs its own path to a free charger room and back again to its home room.

The path following program of these robotic agents will also overcome a major drawback of the above program. `find_and_go_through_doorway` will stop the robot when the exit door is closed, or when there is robot in the room that it wants to enter. If the door is not re-opened, or the robot in the next room wants to pass through the same doorway, the robot will stay put for ever.

The control procedures of Section 6.9 will compute a new path when a door is seen to be closed, or another door that needs to be passed through is notified as closed by another robotic agent. In addition, because each knows the paths of the other robots, the deadlock of two robots needing to pass through the same room can be detected. Using priority rules one agent gets its robot to back away from the centre of the room so that the other robot may pass through unobstructed.

### 4.6 Adding extra actions to the percepts handler

In Section 3.16 we suggested that the robotic agents inform the tracker agent of the open or closed status of any doors that they see as they follow a path through rooms. This is agent shared perception and the most straightforward way of doing this is to tailor the percepts handler of the agent shell so that when a `see_door` percept has been added to a robotic agent’s Belief Store this is also sent as a message to the tracker agent.

We can do this by defining an action called `post_percepts_updates_` if the updates convention for percept communication is being used, and by defining an action called `post_percepts_all_` if the all convention for percept communication is being used. If the appropriate procedure is defined in the agent’s program file it will be called immediately after the percepts handler has done its standard updates, including the updates for any rule defined percepts.

`post_percepts_updates_` is called with argument the list `r_(percept), f_(percept)` terms just used to update the percepts by the percepts handler. `post_percepts_all_` is called with two lists of percept arguments. The first list contains all the percept facts just forgotten by the percepts handler, the second all those it has just remembered.

Below is a definition of `post_percepts_updates_` that will send a message `status(door,room,door_status)` to the tracker each time it has just remembered a `see_door(door, door_status)` percept. Such a `status` message is handled by the tracker agent of Section 3.16 to update its be-
lies about which room doors are closed. The call to the subsidiary action check_for_room_change then checks to see if f_(in_doorway(Door)) is in the list of percept updates just received. If it is, the robot has just entered the next room on the path being followed. The action queries my_loc to find last room that the robot was in, it then queries the path_to_charger belief remembered by the agent’s message handling thread to find the room that will have just been entered. It then sends a new_loc Pedro notification to tell the tracker agent and each other robotic agent its new room location.

```
post_percepts_updates_(R_F_List) ~> 
  ? my_loc(Rm);
  forall Door, Status 
    (r_(see_door(Door,Status) in R_F_List ~> status(Door,Rm,Status) to tracker));
  check_for_room_change(R_F_List)

act check_for_room_change(list(percept_update_))
check_for_room_change(R_F_List) ::
  f_(in_doorway(Door)) in R_F_List ~> 
    ? my_name(Me) & loc(Me,InRm) & path_to_charger(Path) & 
      through(Door,InRm,OutRm,_) in Path;
    forget my_loc(InRm);
    remember my_loc(OutRm);
    new_loc(Me,OutRm) to pedro
% f_(in_doorway(Door)) will be received in percepts update list
% immediately after the robot has entered OutRm through Door
check_for_room_change(_ _) ~> {} 
```
Remember that init_agent will be called as the first action of the agent’s message handling thread before it enters the loop removing successive messages from its buffer and calling handle_message to handle each one. The use_charger message will have sender the tracker agent as it will be sent as an addressed message directly to the robotic agent in response to a reserve request.

4.8 Memory Structure Architectures

When the sensor percepts are augmented by rule defined relations, as they are in the tower building program, there are two main ways in which the rules can be handled. One way is to use the rules in a forward inferring way to find and remember all the instances of the defined relations for the first batch of percepts, and to update the remembered instances using a truth maintenance system (TMS) when each time a new batch of percepts arrives. This involves remembering the dependencies of the inferred instances. In the second way, the rules are used as in QuLog in backward inferring fashion to test or to find instances of the defined relations each time a guard that uses them is evaluated. No dependencies need to be remembered but an instance may have to be re-inferred several times.

4.8.1 Triple tower memory architecture

In Nilsson’s triple tower agent memory structure architecture [63] depicted in Fig. 4.2 the first method is adopted. Each tower corresponds to a layering of progressively more abstract concepts. As we ascend the definitions tower we
find definitions of higher level beliefs defined in terms of lower level ones. As we ascend the beliefs tower we find the inferred instances of these higher level
beliefs. As we ascend the action tower we find higher level TR procedures with preconditions that query these higher level beliefs and which call lower level procedures that query the lower level beliefs. At the top of the action tower we have behavioural skills that require reasoning to determine how to act, where the actions are typically complex activities defined by TR procedures. At the bottom we have behavioural skills that directly respond to changing percept beliefs with primitive actions.

The Java applet simulation of [64] shows all the facts in the agent’s belief tower as the configuration of blocks changes. For example, if you start with two blocks labelled a and b on the table you will see the facts

\[
\begin{align*}
on(a,\text{table}) & \quad on(b,\text{table}) & \quad clear(a) & \quad clear(b) \\
stack([a]) & \quad stack([b]) & \quad tower([a]) & \quad tower([b])
\end{align*}
\]

The clear, stack and tower facts are all inferred using the Belief Store rules. The applet uses beliefs such as on(a,table) rather than on(a,table) which we have used.

If you move block a to be on top of block b you will see

\[
\begin{align*}
on(b,\text{table}) & \quad on(a,b) & \quad clear(a) \\
stack([b]) & \quad stack([a,b]) & \quad tower([a,b])
\end{align*}
\]

clear(a) and stack([b]) were the only inferred facts not to have their dependencies undermined by the change of on percepts. stack([a,b]) and tower([a,b]) are newly inferred.

### 4.8.2 Double tower memory architecture

The backward inferring method whereby instances of a defined relation are inferred each time a condition using the relation is evaluated in a guard evaluation, with no remembering of the inferred instances, corresponds to a two tower architecture as depicted in Fig. 4.3. A hybrid approach would use some rules to infer all instances of the relations they define when the percepts arrive, with the inferred instances being remembered. The TeleoR rule guards then query the remembered facts and do not use the rules. We then re-infer instances of these rule defined relations for each new batch of primitive percepts, forgetting previously inferred instances that are no longer inferable and remembering new inferred instances. Using this simple approach we do not need to remember dependencies.

This approach can be adopted for the relations next_to, close_to, near and see defined in terms of sensor percept relation see_patch in Section 2.5. In the TeleoR program we signal that their rules are to be used in this way after each percepts update by declaring them as percept relations even though they are rule defined. They are percepts in the sense that instances
for these relations are stored as facts refreshed when each new batch of sensor percept facts arrives at the agent.

To allow inferring of all the instances of such defined relations in any order we restrict their definitions so that there is no direct or indirect dependencies between the definitions. The definitions can depend upon other defined relations but these other relations must not also be declared as percept relations. For example, in our tower building program we could declare stack or tower as percept relations, but not both.

The one rule definitions of next.to, close.to, near and see given in Section 2.5 are independent of one another. Without going into detail, there is a considerable implementation benefit to inferring and remembering all instances of relations such as these that have simple definitions and are close to the sensor percepts.

4.8.3 Multiple Thread Implementation

For both memory architectures we can adopt a multi-threaded implementation. In the case of the Triple Tower, one thread handles the stream of sensor percept facts, the forward inferencing and the associated truth maintenance system. In the case of the Double Tower architecture, this thread handles the stream of sensor percept facts and the generation and updating of all instances of defined percept relations. A second thread can handle a stream of messages from other agents. Any number of other threads can used, as in the path following agent, to respond to significant updates of the Belief Store by doing further updates or by sending out messages. In both memory architectures there is a thread that handles the evaluation of a TeleoR procedure call.
Chapter 5
Operational Semantics of Core TelesoR
This chapter assumes familiarity with predicate logic and set expressions. For an introduction see [?], [76] or Chapter 1 and Appendix B1 of [66].

We begin by giving a control reading of a core TeleoR (hence T-R) procedure. When we introduce the new forms of rules in Chapter 6 we will give similar control readings for them.

We follow that with an informal operational semantics for a TeleoR procedure call illustrated by applying it to the call get_next_to(bottle) using the program of Chapter 2. We follow its evaluation through three Belief Store states. This leads to the formulation of an evaluation algorithm, which we give in Section 5.3.

We then consider the issue of the ambiguity in the original very informal operation semantics given for T-R evaluation in [64]. It is whether or not a rule should be re-fired if another instance of its guard becomes the first inferable instance but the old instance is still inferable. For reasons we shall give, we decided on allowing a re-fire.

In Section 5.6 we formally define the state of a procedure call evaluation as a collection of logically expressed constraints that must be satisfied by a mathematical abstraction of the state. The transition rule that tells us when and how one static state transitions into another is then relatively simple. We illustrate the formal semantics by giving the mathematical descriptions of three states in the evaluation of the call collect_bottle() in Section 5.7. You might find it useful to dip into this section as you read Section 5.6.

5.1 Informal Reading of a Procedure

A core TeleoR procedure consists of a parameterised sequence of guarded rules of the form:

\[
p(X_1,...X_m)\{ \\
K_1 \Rightarrow A_1 \\
. \\
. \\
K_n \Rightarrow A_n \\
\}
\]

\(X_1,...,X_m\) are the global variables of the rules. Variables in \(K_i\) and \(A_i\) that are not global are the local variables of the rule. A local variable appearing in \(A_i\) must also be a local variable of \(K_i\).

Each \(A_i\) is either a call to a TeleoR procedure, or it is a tuple \((a_1,...,a_j), j \geq 0\) of robotic resource actions, which can be a mixture of durative actions and discrete actions. When fully instantiated by the evaluation of the guard \(K_i\) of its rule against the agent’s Belief Store, all the robotic actions are executed in parallel using a robotic resource outside the agent.
$K_i$ is implicitly existentially quantified with respect to its local variables not in $A_i$, these are the ‘don’t care’ variables of $K_i$. The whole rule is implicitly universally quantified with respect to the common local variables of $K_i$ and $A_i$ with $K_i \rightsquigarrow A_i$ read as \textit{when then while $K_i$ do $A_i$}.

We can think of the sequence of all the rules as implicitly universally quantified with respect to the procedure parameters with an implicit \textit{always-do-first-of} after this quantification.

\begin{verbatim}
approach(Th,FS,TS) {
    see(Th,centre) \rightsquigarrow (move(FS))
    see(Th,Dir) \rightsquigarrow (move(FS), turn(Dir,TS))
}
\end{verbatim}

can be read as

\begin{equation}
\forall (Th,FS,TS) \text{ while doing approach}(Th,FS,TS) \\
\text{always do first of} \\
\text{when then while see}(Th,centre) \text{ do } (move(FS)) \\
\forall Dir \text{ when then while see}(Th,Dir) \text{ do } (move(FS), \text{ turn}(Dir,TS))
\end{equation}

\begin{verbatim}
near(Th,Dir) \rightsquigarrow approach(Th,1.5,0.5)
\end{verbatim}

can be read as

\begin{equation}
\text{when then while } \exists Dir \text{ near}(Th,Dir) \text{ do approach}(Th,1.5,0.5)
\end{equation}

\textit{Dir} is universally quantified using $\forall$ for the rule of \textit{approach} as it appears in the action of the rule. It is existentially quantified using $\exists$ for the rule of \textit{get\_next\_to} as it does not appear in the action of the rule. All that matters is that some value exists.

\section{5.2 Informal Operational Semantics}

When a \texttt{Teleor} procedure is called as an action $P = p(v_1, \ldots, v_m)$, its given arguments $v_1, \ldots, v_m$ are substituted for its parameters $X_1, \ldots, X_m$ throughout the rules resulting in a sequence of partially instantiated rules with guards existentially quantified with respect to the variables of the guards not in the action of the rule.
\[ \begin{align*}
K_1 & \rightarrow A_1 \\
\vdots \\
K_n & \rightarrow A_n
\end{align*} \]

where the bolding indicates the partial instantiation and existential quantification.

**Example** The procedure call `get_next_to(bottle)` for the procedure

```prolog
get_next_to(Th){
    next_to(Th,centre) \( \rightarrow \) ()
    next_to(Th,Dir) \( \rightarrow \) (turn(Dir,0.3))
    close_to(Th_,) \( \rightarrow \) approach(Th,0.2,0.2)
    near(Th_,) \( \rightarrow \) approach(Th,0.5,0.2)
    see(Th_,) \( \rightarrow \) approach(Th,1.5,0.1)
    true \( \rightarrow \) (turn(left,0.5))
}
```

of Section 2.2 gives us the following partially instantiated rules

\[ \begin{align*}
\exists \text{Dir next\_to(bottle,Dir)} & \rightarrow () \\
\text{next\_to(bottle,Dir)} & \rightarrow (\text{turn(Dir,0.3)}) \\
\exists \text{Dir close\_to(bottle,Dir)} & \rightarrow \text{approach(bottle,0.2,0.2)} \\
\exists \text{Dir near(bottle,Dir)} & \rightarrow \text{approach(bottle,0.5,0.2)} \\
\exists \text{Dir see(bottle,Dir)} & \rightarrow \text{approach(bottle,1.5,0.1)} \\
\text{true} & \rightarrow (\text{turn(left,0.5)})
\end{align*} \]

Note all occurrences of the anonymous variable ‘\( \_ \)’ are replaced by an existentially quantified `Dir`.

The existentially quantified guards \( K_1 \) to \( K_n \) of the rules of \( P \) are then evaluated, one after the other. This is to determine the *earliest* rule \( i \) such that \( K_i \) has a successful evaluation against the current state of the agent’s **Belief Store**, i.e. which has a ground instance \( K_i \theta \) that can be inferred from the store.

\( \theta \) is a set of bindings for the free variables of \( K_i \), if any, which are the variables in \( A_i \). There may be more than one such set of bindings, but the \( \theta \) returned will be the *first* that the QuLog query evaluation method returns.

The \( i \)th rule instance \( K_i \theta \rightarrow A_i \theta \) becomes the *fired rule instance* of the procedure call, with \( A_i \theta \), the action to execute. If this is a **TeleoR** procedure call \( Q \) the rules of \( Q \) are then checked in the same way and will normally result in the firing of a rule of \( Q \), which may also have a procedure call action. This cascading of procedure calls will continue until a rule is fired with robotic resource actions.

100
If for some procedure call there is no guard query that succeeds, meaning that none has an inferable instance given the current state of Belief Store, this is a error condition and the whole TeleoR evaluation terminates.

There is also the outside possibility that this cascading of procedure calls would not terminate for the current state of the Belief Store. An example is a call to the silly procedure below where SomeTest fails for the current Belief Store:

```plaintext
silly()
    {
      SomeTest ~> ()
      true ~> silly()
    }
```

Recursion in TeleoR procedures is not common and in any case nearly always has a small depth bound determined by the size of some argument of the call as in the recursive makeTower of Section 4.1. We can therefore set an upper limit to the depth of TeleoR procedure calls, which can be TeleoR program specific and declared in the program file. In our implementation the default value for this call limit is 100. We therefore have another error which is that the call depth limit has been reached. It is an extremely unlikely error condition but also marks the end of a TeleoR task evaluation.

### 5.2.1 Controls

If the cascade of procedure calls does terminate it will end with the firing of a robotic resource actions rule of the very last procedure call. If this is the first firing of a robotic actions rule for a task, the last tuple of robotic actions are taken to be the empty tuple (). Otherwise, the last tuple of actions LastActs comprises the tuple of actions of the most recent prior firing of a TeleoR rule with robotic actions.

A set of control messages is generated using LastActs and Acts. Any durative action Dur in LastActs not in Acts is stopped - a control stop(Dur) is generated. Any durative action in both LastActs and Acts is allowed to continue. No control is generated. If a durative action Dur in Acts has the same action name but different arguments as an action LDur in LastActs, a control mod(LDur, Dur) is generated. For any durative action Dur in Acts not in LastActs a start(Dur) control is generated. Finally, for any discrete action Dis in Acts a control do(Dis) is generated. The controls are then sent.
**Example** Suppose that $\text{get\_next\_to(bottle)}$ is called for the first time as part of the evaluation of a task with start call $\text{get\_bottle}$. Suppose this start call has fired its 5$\text{th}$ rule because we have

$$\text{Belief Store} = \{\text{see(bottle, left)}, \text{gripper\_open()}\}$$

The 5$\text{th}$ of the partially instantiated rules for the call $\text{get\_next\_to(bottle)}$ becomes the fired rule of this procedure call with $\theta = \{\}$, since the binding for $\text{Dir}$ is not retained, with action $\text{approach(bottle, 2.5, 0.3)}$.

The $\text{approach}$ call has partially instantiated rules

$$\text{see(bottle, centre)} \leadsto (\text{move}(2.5))$$
$$\text{see(bottle, Dir)} \leadsto (\text{move}(2.5), \text{turn(Dir, 0.3)})$$

The second rule of this call is fired with $\theta = \{\text{Dir = left}\}$. This is a rule firing with an instantiated guard

$$\text{see(bottle, Dir)}\{\text{Dir = left}\} \text{ i.e. see(bottle, left)}$$

and an instantiated tuple of robotic actions

$$(\text{move}(2.5), \text{turn(left, 0.3)})$$

If there was any preceding tuple of actions it was $(\text{open\_gripper()})$ containing a discrete action. These two durative actions will therefore be started. The set of controls

$$\{\text{start(move(2.5)), start(turn(left, 0.3))}\}$$

are sent.

### 5.2.2 Response to a Belief Store update

When the Belief Store is next updated, the guards of the rules of the start procedure call are re-checked. The re-check is to see if the same rule should be allowed to continue as the fired rule of the call, and if not, if the same rule should be re-fired, or if a different rule should be fired. The same $i$th rule $K_i \leadsto A_i$, should be allowed to continue as the fired rule if it is still the first rule with an inferable guard and the inference returns the same set of bindings $\theta$ for $K_i$, as the first answer. The rule is re-fired if it is still the
first rule with an inferable guard but the inference returns a different set of bindings \( \psi \) for \( K_i \) as the first answer.

It is a continuation and not a re-fire if the re-inference of \( K_i \) from the new Belief Store, which in practice will be a QuLog evaluation of the query \( K_i \), without the existential quantifiers, returns different values for just the existentially quantified variables of \( K_i \), the local variables of \( K_i \) not in \( A_i \).

We do not care about the values of these variables, all that matters is that values exist such that \( K_i \psi \) can be inferred.

In the above example, the fired rule of \( \text{get} \_\text{next} \_\text{to(bottle)} \) is its 5\textsuperscript{th} rule with existentially quantified guard \( \exists \text{Dir} \ \text{see(bottle, Dir)} \). This was inferable because \( \text{see(bottle, left)} \) was in the Belief Store. Dir would have been given the binding left but that binding was discarded as Dir does not appear in the rule action. Suppose that the Belief Store is updated again and now contains \( \text{see(bottle, centre)} \) but no \( \text{near(bottle, ...)} \) fact. \( \exists \text{Dir} \ \text{see(bottle, Dir)} \) is still inferable and the fact that Dir will be given a different binding is ignored. There is no rule re-firing and the call \( \text{approach(bottle, 2.5, 0.3)} \) will continue.

**Example continued** Suppose our Belief Store is updated when new percepts arrive to

\[
\{\text{see(bottle, centre)}, \text{near(bottle, centre)}, \text{gripper\_open()}\}
\]

The fifth rule of \( \text{collect\_bottle} \) will remain its fired rule. When the rules for the call \( \text{get} \_\text{next} \_\text{to(bottle)} \) are re-checked the first rule with an inferable guard is now rule 4. This switch of fired rule will cause the call to \( \text{approach(bottle, 2.5, 0.3)} \) to be terminated and a new call \( \text{approach(bottle, 1.5, 0.5)} \) to be invoked.

The partially instantiated rules for this new \textit{approach} call are

\[
\begin{align*}
\text{see(bottle, centre)} & \Rightarrow (\text{move(1.5)}) \\
\text{see(bottle, Dir)} & \Rightarrow (\text{move(1.5)}, \text{turn(Dir, 0.5)})
\end{align*}
\]

The first rule of \( \text{approach(bottle, 1.5, 0.5)} \) will be fired as the \textit{bottle} is in centre view. As its action tuple is \( (\text{move(1.5)}), \) and the last determined tuple of robotic actions was \( (\text{move(2.5)}, \text{turn(left, 0.3)}) \), the sent set of controls is therefore

\[
\{\text{mod(move(2.5), move(1.5)), stop(turn(left, 0.3))}\}
\]

Assume Belief Store is now updated to

\[
\{\text{see(bottle, right), near(bottle, right), gripper\_open()}\}
\]
The quantified guard $\exists \text{Dir} \ near(bottle, \text{Dir})$ of the 4th rule of 
$\text{get\_next\_to}(bottle)$ is still inferable. Evaluation of $\text{near}(bottle, \text{Dir})$ will 
return a different value for $\text{Dir}$, but this does not affect the rule action 
which is still the call $\text{approach}(bottle, 1.5, 0.5)$.

However, the second rule of $\text{approach}(bottle, 1.5, 0.5)$ will be now be 
fired with returned binding \{Dir=right\}. This results in a new tuple of 
actions $(\text{move}(1.5),\text{turn(right,0.5)})$. The previous $\text{move}(1.5)$ action is 
allowed to continue unchanged but the set of controls 

\{start(\text{turn(right,0.5)})\}

will be sent.

5.3 An Evaluation Algorithm

We can describe the evaluation cycle of a task executing a start $\text{TeleoR}$ 
procedure call $\text{SP}$ at time $\text{ST}$ with the following 8 step algorithm. 

$\text{Fired}$ is the sequence of active procedure calls. Each element of $\text{Fired}$ is 
a 4-tuple of the form $(\text{Call}, R, \theta, T)$ where $R$ is the number of the partially 
instantiated rule of procedure call $\text{Call}$ that was fired at time $T$, and $\theta$ is the 
set of generated bindings for all the variables of the action of that rule.

1. $\text{LastActs} := (); \text{Fired} := (); n := 1; \text{Call} := \text{SP}; T = \text{ST}$.
2. $n > \text{MaxDp}$: Signal a $\text{call-depth-reached}$ failure; 
   Compute and send $\text{Controls}$ using actions () and $\text{LastActs}$; 
   Terminate.
3. There is no rule for $\text{Call}$ with an inferable instance of its guard: 
   Signal a $\text{no-fireable-rule}$ failure; 
   Compute and send $\text{Controls}$ using actions () and $\text{LastActs}$; 
   Terminate.
4. The $R^{th}$ rule of $\text{Call}$ is its first rule with an inferable guard with 
   $\theta$ being the first returned substitution: 
   Rule $R$ with substitution $\theta$ is the $\text{fired rule instance}$ of $\text{Call}$. 
   Add $(\text{Call}, R, \theta, T)$ to the end of $\text{Fired}$.
5. The rule’s action $\text{A}$ is a procedure call: 
   $\text{Call} := \text{A}\theta; n := n + 1$; 
   Go to step 2.
The rule’s action $A$ is a tuple of robotic actions:
Compute and send $Controls$ using actions of $A\theta$ and $LastActs$;
$LastActs := A\theta$.

7. Wait for a $Belief Store$ update. $T := time\_of\_update$
8. There is a first tuple $(Call_j, R_j, \theta_j, T_j)$ at position $j$ in $Fired$
where rule $R_j$ with substitution $\theta_j$ should no longer
continue as the fired rule instance of $Call_j$ at time $T$:
$n = j; Call := Call_j$;
Remove tuples from position $j$ from $Fired$;
Go to step 3.

$MaxDp$ is the maximum number of allowed active calls. $LastActs$ is the
last tuple of determined actions for $SP$, initialised to $()$. Step 1, followed
by an iteration of steps 4 and 5, generate the initial $Fired$ sequence for $SP$
for the initial state of the $Belief Store$. This iterative generation terminates
successfully when a rule is fired with a tuple of robotic actions $A\theta$. $LastActs = ()$ and the tuple of actions of $A\theta$ are then used to generate the initial $Controls$
set that is sent in step 6. $LastActs$ is updated to the new tuple of actions.

The algorithm then suspends at step 7 until the $Belief Store$ is updated. If
the test of step 8 fails the algorithm waits for another $Belief Store$ update and
several updates may occur before a new tuple of actions is determined. During
this time any durative actions in $LastActs$ will be allowed to continue. When
the test of step 8 succeeds, the $Fired$ sequence is truncated to the $j - 1^{th}$
entry and is extended from that point on with a new sequence of calls. All
the entries of the truncated sequence record rule firings that continue as their
call’s fired rule instance. $Fired$ may be truncated to the empty sequence if a
new fireable rule for $SP$ must be sought.

The algorithm leaves unspecified the logical condition that determines
whether a rule instance should continue or not as a call’s fired rule instance,
the crucial test at step 8. This condition is defined in the formal semantics
and informally discussed below. Every such update of $Belief Store$ will be
deemed to determine a new state of evaluation even if $Fired$ remains un-
changed after the update - a situation handled by the jumps back to step 7
from step 9 of the algorithm.

5.4 Re-firing when Previous Guard Instance is Still
Inferable

At step 8 in the evaluation algorithm the minimal condition for a rule firing
$(Call, R, \theta, T)$ to be allowed to continue is that the guard of rule $R$ of $Call$
is still inferable from the new state of the Belief Store with substitution \( \theta \), and no guard of an earlier rule is inferable. If \( \theta \) is no longer an inferable substitution, and another substitution \( \psi \) of the guard is the first inferable substitution, the \( R^{th} \) rule is re-fired providing no earlier rule can be fired. Should we allow re-firing of the rule when the guard with substitution \( \theta \) is still inferable?

As an example of how this can happen consider the following Belief Store rules where bottles have different colours and the vision percepts are defined in terms of \( \text{see\_patch} \) as in Section 2.5. The bottle colour is also an argument to \textit{approach}.

\[
\exists \text{ Dir see(bottle, Col, Dir) } \rightarrow \text{ approach(bottle, Col, 2.5, 0.3)}
\]

\[
\text{rel see(thing, colour, dir)}
\]

\[
\text{see(bottle, blue, Dir) } \leftarrow \text{ see\_patch(blue, Size, Dir) } \& \text{ Size } > \text{ 50}
\]

\[
\text{see(bottle, Col, Dir) } \leftarrow \text{ see\_patch(Col, Size, Dir) } \& \text{ a\_bottle\_colour(Col) } \& \text{ Size } > \text{ 50}
\]

This is part of a bottle collecting program where there is a premium on collecting blue bottles. It is a little artificial but will illustrate the point.

Let us suppose that after a percepts update \( \text{see\_patch(green, 80, left)} \) is the only \( \text{see\_patch} \) fact in the Belief Store. The \texttt{TeleoR} rule will be fired with \textit{green} as the colour of \textit{bottle} to be approached. Suppose that while the green bottle is being approached a blue bottle is put down within sight of the robot and \( \text{see\_patch(blue, 55, right)} \) is added to the Belief Store on the next percepts update. \( \text{see\_patch(green, 90, left)} \) also replaces \( \text{see\_patch(green, 80, left)} \) as the green bottle, a little closer, is still in sight as well. \( \exists \text{ Dir see(bottle, blue, Dir)} \) will now be the first inferable instance of the guard of the rule

\[
\exists \text{ Dir see(bottle, Col, \_)} \rightarrow \text{ approach(bottle, Col, 2.5, 0.3)}
\]

giving a new action \( \text{approach(bottle, blue, 2.5, 0.3)} \) but the previous guard instance \( \exists \text{ Dir see(bottle, green, Dir)} \) will still be inferable.

This is an example of where we have used Belief Store fact/rule ordering to implement a choice preference that can cause a change of behaviour when new percepts arrive, just as \texttt{TeleoR} rule ordering reflects action preference that can and often does affect behaviour when new percepts arrive.

In [63] \texttt{T-R} rule re-firing on Belief Store updates was not mentioned and so the issue of whether or not a rule should be re-fired when the guard of the previously fired rule instance was still inferable was left ambiguous. In the earlier papers [61],[60], which did not have Belief Store rules, the informal semantics for \texttt{TeleoR} procedures was based on the concept of analogue inputs
and outputs. Re-firing a rule when the previous guard instance still held was not possible. There could only be one analogue input at any time.

We made the choice to allow re-firing as the default, even when the previous guard instance was still inferable, so that Belief Store rules could be used to express behaviour preferences, as in this example. However, there are situations where one would like to stick with the previously determined action when it is still an option, especially when this is a procedure call that may have acquired robotic resources in a multi-tasking multi-resource using agent. We will return to the question in Section ?? when discussing future extensions of TeleoR.

Allowing rule re-firing when the previous guard instance in inferable is equivalent to having a continue test for a firing \((\text{Call}, R, \theta, T)\) that requires \(R\) to still be the first rule of \(\text{Call}\) with an inferable guard and \(\theta\) to still be the first inferable instance of the guard. Sticking with a previously determined action is equivalent to a continue test that requires \(R\) to still be the first rule of \(\text{Call}\) with an inferable guard and \(\theta\) to still be an inferable instance of the guard.

5.5 fire and continue Semantics of TeleoR Core Rules

To formalise the operational semantics the first thing we need to do is to more precisely define the conditions for when a rule instance is the fireable rule instance of the procedure call \(P\). We should also be more precise about the condition which allows a fired rule instance to continue being the fired rule instance of the call. Since we have decided to allow re-firing even when the guard instance of the previous firing is still inferable, this is equivalent to the rule’s remaining the fireable rule instance of its call. Had we decided to prohibit re-firing in this situation the condition under which a rule continued as the fired rule instance would be different. When we give the semantics of a TeleoR procedure call evaluation where we have while and until rules we shall see that a rule can be continued as the fired rule instance when it is no longer the fireable instance.

We assume a function \(BS\) such that \(BS_T\) is the Belief Store state at time \(T\). The state changes when new percepts arrive, which can happen as frequently as every few hundredth’s of a second, and not necessarily at fixed intervals. We will therefore assume that our evaluation algorithm includes the time of the most recent Belief Store update when it adds rule firings to the Fired sequence. We will define the following:

\[
\text{fire}(BS,(P,R,\theta,T)) - R \text{ is the first rule of procedure call } P \text{ with an inferable instance of its guard } K \text{ from } BS_T \text{ and } K\theta \text{ is the first such inferable instance. } P_R\theta \text{ is the fireable rule instance of call } P \text{ at time } T
\]
continue(BS,(P,R,θ,T),T') - P_Rθ is the fireable rule instance of call P at time T'.

The continue test will only be applied where fire(BS,(P,R,θ,T)) holds and T' > T.

Some notation

BS_T ⊢ f Cθ means the ground instance Cθ is inferable from BS_T with θ the first returned set of bindings for the free variables of C.

P_R is the Rth rule of a procedure call P.

guard(P_R) is the guard of the rule P_R

actions(A) A comprises a fully instantiated tuple (a_1,..,a_j), j ≥ 0 of robotic resource actions

procCall(A) A is a fully instantiated TeleoR procedure call.

≡ is read as is defined as.

If S is a sequence s_1,..,s_n, S_i is s_i and S_{i,j} is the sub-sequence s_i,..,s_j.

≺, read as immediately precedes, is the relation between successive different Belief Store states. It is defined by

\[
BS_T \prec BS_{T'} \equiv T < T' \land BS_T ≠ BS_{T'} \land ¬\exists t (T < t < T' \land BS_T ≠ BS_t)
\]

Finally we define a relation no_prior_fireable_rule(BS, P, R, T) which means that no rule before rule R in the sequence of rules of call P has an inferable guard at time T.

\[
\text{no\_prior\_fireable\_rule}(BS, P, R, T) \equiv \\
\forall r : 1 ≤ r < R \neg ¬\exists θ(BS_T \vdash \text{guard}(P_r)θ)
\]

5.5.1 The formal definitions of fire and continue

We can now give the formal definitions of fire, continue.
5.6 State Transition Semantics

We can now use these definitions to precisely define the state of a TeleoR evaluation that begins with some call $SP$ to a TeleoR procedure. As is usual when giving this type of semantic definition, we shall define the state as some tuple of values.

In the next section, 5.7, we give an example three step evaluation of an initial call to $collect\_bottle$ using the mathematical representation of the evaluation state. The fired rules used are those of the program of Chapter 2. You might find it useful to alternate between reading this quite mathematical section and the example evaluation.

5.6.1 Formal specification of an evaluation state

An evaluation state at time $T$ for some TeleoR procedure call $SP$ started at $ST \leq T$ with maximum call depth $MaxDp$ and Belief Store function $BS$, is a 4-tuple

$$(T, Fired, LastActs, Controls)$$

with $Fired$ the sequence of $n, 0 \leq n \leq MaxDp$, 4-tuples:

$$(P_1, R_1, \theta_1, T_1), (P_2, R_2, \theta_2, T_2), \ldots, (P_n, R_n, \theta_n, T_n)$$

where

$\forall i: 1 \leq i \leq n \ P_i R_i = (K_i \rightarrow A_i)$
$P_1 = SP \quad \forall i: 1 \leq i < n \ P_{i+1} = A_i \theta_i$ (Adjacency condition)
$\forall i: 1 \leq i \leq n \ fire(BS, Fire_i)$ (Firing condition)

and the state is either non-terminal or terminal as specified below.

We assume that $SP$ has a first rule $K_1 \rightarrow ()$ where $K_1$ is the goal of the procedure call. One state of the evaluation may therefore have $n = 1, R_1 = 1$
and \( \theta_1 = \{ \} \) when the goal of the call \( SP \) has been achieved. Another possible state has Fired the empty sequence when there is no fireable rule for \( SP \).

We have not specified the value of LastActs or Controls. LastActs will be given for the initial evaluation state and thereafter will be given for each new state by the state transition rule. The value of Controls depends upon whether the state is a non-terminal state or not.

### 5.6.2 Formal conditions specifying a non-terminal and a terminal evaluation state

#### Conditions characterising a non-terminal evaluation state

\[
\text{actions}(A_n \theta_n) \quad (5.1)
\]

\[
\text{Controls} = \text{update}(\text{LastActs}, A_n \theta_n) \quad (5.2)
\]

#### Conditions characterising a terminal evaluation state

\[
n = \text{MaxDp} \land \text{procCall}(A_n \theta_n) \quad (5.3)
\]

\[
\lor
\]

\[
0 < n < \text{MaxDp} \land \text{procCall}(A_n \theta_n) \land
\neg \exists (R, \theta) \text{fire}(BS, (A_n \theta_n, R, \theta, T)) \quad (5.4)
\]

\[
\lor
\]

\[
n = 0 \land \neg \exists (R, \theta) \text{fire}(BS, (SP, R, \theta, T))
\]

\[
\text{Controls} = \text{update}(\text{LastActs}, ()) \quad (5.5)
\]

\[
\text{update}(\text{LastActs}, \text{NewActs}) =
\{ \text{stop}(a) \mid \text{dur}(a) \land a \in \text{LastActs} \land a \notin \text{NewActs} \}
\lor
\{ \text{start}(a) \mid \text{dur}(a') \land a' \in \text{NewActs} \land a' \notin \text{LastActs} \land
\neg \exists a (a \in \text{LastActs} \land \text{modification}(a', a)) \}
\lor
\{ \text{mod}(a, a') \mid \text{dur}(a') \land a' \in \text{NewActs} \land a \in \text{LastActs} \land
\text{modification}(a', a) \}
\lor
\}
\]
\{ do(a') \mid \mathit{dis}(a') \land a' \in \text{NewActs} \}\]

When we have defined the initial state and the state transition rule we shall see that \( T_i \) is always the time that \((P_i, R_i, \theta_i, T_i)\) was added to \(\text{Fired} \). By induction on the number of times the transition rule is applied we shall see that:

\[
\forall i : 1 \leq i \leq n \forall t : T_i < t \leq T \text{ continue}(BS, Fired_i, t) \tag{5.6} \\
\land \\
\forall i : 1 < i < n T_i \leq T_{i+1} \leq T
\]

also holds of every evaluation state.

Condition 5.1 charactering a non-terminal evaluation state tells us that \( A_n\theta_n \) is a tuple of robotic resource actions. Condition 5.2 tells us that for such a state Controls comprises the control actions needed to update LastActs for the actions of \( A_n\theta_n \).

The first disjunct of condition 5.3 for a terminal state is the call-depth-reached error of the evaluation algorithm of Section 5.3. The last two are its no-fireable-rule error. The second condition 5.5 tells us that the Controls for a terminal state will stop any durative action in LastActs.

In the definition of the update function \(\mathit{dur}(a)\) means \( a \) is a durative action, \(\mathit{dis}(a)\) that \( a \) is a discrete action. The condition \(\mathit{modification}(a', a)\) means that the two actions are the same parameterised action, such as move, but have different arguments.

5.6.3 Initial evaluation state

The initial state for the evaluation of a call \( SP \) started at \( ST \) is the 4-tuple

\[
(ST, Fired, (), Controls) \\
\text{where} \\
\forall i, 1 \leq i \leq n T_i = ST \\
\text{and} \\
\text{Controls} \text{ is determined by 5.2, or 5.5 if new state is terminal}
\]

\( Controls \) set sent at time \( ST \) if non-empty. Condition 5.6 is trivially satisfied for the initial state.
If it is a non-terminal state, the next state will materialise after $ST$ when the next update of the Belief Store occurs. Every other evaluation state is generated from the initial state by a sequence of applications of the state transition rule below.

### 5.6.4 State transition rule

For a non-terminal evaluation state $(T, Fired, LastActs, Controls)$ we have the transition rule

\[
BS_T \prec BS_{T'} \Rightarrow (T, Fired, LastActs, Controls')
\]

where

\[
\forall i : 1 \leq i \leq n \ (continue(BS, Fired_i, T')) \land Fired' = Fired
\]

\[
\lor
\exists j : 1 \leq j \leq n

(\forall i : 1 \leq i \leq j - 1 \ continue(BS, Fired_i) \land Fired'_i \land \neg continue(BS, Fired_j, T')) \land

\forall i : j \leq i \leq n' \ T_i = T'
\]

and

Controls' is determined by condition 5.2, or condition 5.5 if new state terminal.

$\Rightarrow$ can be read as transitions to. The new state records the time $T'$ of the Belief Store update that triggers the transition. LastActs' = $A_i \theta_n$, the robotic action tuple determined in the preceding state. The Controls' set is sent at time $T'$ if non-empty.

The adjacency condition for an evaluation state tells us that $A_{j-1} \theta_{j-1} = P'_j$, and that for all $i$ from $j$ to $n'$, $A_i' \theta_i' = P'_{i+1}$ This, and the firing condition, tell us that the new entries from $j$ on are generated at time $T'$ by a cascade of rule firings resulting from a new firing for procedure call $A_{j-1} \theta_{j-1}$ at time $T'$. It could be that $n' = j - 1$ because no rule of $A_{j-1} \theta_{j-1}$ can be fired at time $T'$. In which case the new state is terminal.

We leave the reader to check that if condition 5.6 applies to the transition-state it will also apply to the new state. Remember there is no change of $BS_T$ until $BS_{T'}$.

Note there is no specified connection in the state transition semantics between the Controls of the prior state and the new Belief Store state $BS_{T'}$ that triggers the transition. Indeed, the new Belief Store state may not be the result of the previously sent controls. Actions by other agents or nature may be the cause of the next state transition.
5.7 An Example Three Step Abstract Evaluation

The example we will use is an evaluation that starts with a call `collect.bottle()` at time `st`.

\[ BS_{st} = \{ \text{see(bottle, left), gripper.open()} \} \]

\[
\begin{align*}
(st, \\
( \text{collect.bottle()}, 5, \{ \}, st), & \quad \% \text{Fired sequence} \\
( \text{get.next.to(bottle)}, 5, \{ \}, st), & \\
( \text{approach(bottle, 2.5, 0.3)}, 2, \{ \text{Dir = left}, st \} & \\
), & \\
\{ \text{start(move(2.5))}, \text{start(turn(left, 0.3))} \} & \\
) \\
\% \text{Action tuple of last fired rule instance is (move(2.5), turn(left, 0.3))} &
\end{align*}
\]

Now suppose that two seconds later `BS_{st}` is updated.

\[ BS_{st+2} = \{ \text{see(bottle, centre), near(bottle, centre), gripper.open()} \} \]

\[
\begin{align*}
(st + 2, \\
( \text{collect.bottle()}, 5, \{ \}, st), \\
( \text{get.next.to(bottle)}, 4, \{ \}, st + 2), \\
( \text{approach(bottle, 1.5, 0.5)}, 1, \{ \}, st + 2) & \\
), & \\
( \text{move(2.5)}, \text{turn(left, 0.5)}), & \\
\{ \text{mod(move(2.5), move(1.5))}, \text{stop(turn(left, 0.1))} \} & \\
) \\
\% \text{Action tuple of last fired rule instance is (move(1.5))} &
\end{align*}
\]

The change point in the Fired sequence is \( j = 2 \). Finally, let us suppose that 3 seconds later there is another percepts update of the Belief Store.

\[ BS_{st+5} = \{ \text{see(bottle, right), close.to(bottle, right), gripper.open()} \} \]

\[
\begin{align*}
(st + 5, \\
( \text{collect.bottle()}, 5, \{ \}, st), \\
( \text{get.next.to(bottle)}, 3, \{ \}, st + 5), \\
( \text{approach(bottle, 0.5, 0.5)}, 2, \{ \text{Dir = right}, st + 5 \} & \\
), & \\
( \text{move(1.5)}), &
\end{align*}
\]

113
\{ \text{mod}(\text{move}(1.5), \text{move}(0.5)), \text{start}(\text{turn}(\text{right}, 0.5)) \} \\
% Action tuple of last fired rule instance is (\text{move}(0.5), \text{turn}(\text{right}, 0.5))

Again the change point is $j = 2$