Agent-Oriented Programming in Linear Logic

Abdullah-Al AMIN
Supervisor: Dr. James Harland
Technical Supervisor: Dr. Michael Winikoff

Honours Thesis

Department of Computer Science
RMIT University
Melbourne, Australia
amin@cs.rmit.edu.au

October 25, 1999

Abstract

This thesis investigates how a linear logic programming language, such as Lygon, can be used in the implementation of agent-oriented programs. Agent-oriented programming is a recent computational framework of interest to both academic and industrial researchers. Agent methodology is being successfully utilised in designing complex (distributed) applications that require concurrency, reasoning, communication, sharing and integration of knowledge, and, of course, intelligence. On the other hand, linear logic, a logic of resource-consumption, provides the possibility to construct efficient tools for modelling updates, reasoning about the environment and implementing concurrency. Linear logic has been used as a basis for creating a number of programming languages. One of these is the logic programming language Lygon. The aim of this thesis is to investigate the possibility of implementing agents with Lygon. A number of experiments have been carried out and results analysed, which will be useful to future agent researchers who would like to use Lygon as a linear logic agent tool.

Keywords: Agent, Agent-Oriented Programming, Linear Logic, Lygon, Logic Programming

Acknowledgements

First, I would like to thank my supervisor Dr. James Harland for his ongoing support, guidance, care, and understanding throughout the year. Dr. Michael Winikoff is the second person I am grateful to, because of his sincere help and patience in guiding me through the end of this project as a technical supervisor. Especially, I would like to thank him for his help in writing programs in Lygon and composing the thesis in LaTeX. I am really fortunate to have James and Michael as my supervisors. I acknowledge their thorough help with ideas, and of course, staying late at night in the UK (James) and sacrificing a weekend in Melbourne (Michael) to proof-read my draft thesis and add valuable comments.

Thanks go to the Department of Computer Science of RMIT University for funding my course and research as well as employing me as a staff. I would like to thank all the teachers and staff, especially Raja Ghosal, Dr. Lin Padgham, Dr. Hugh Williams, Dr. Vic Ciesielski and Phil Vines for encouraging me all the time. Personally, I am grateful to Kathleen Lynch, our International Student Advisor, for her unconditional mental support throughout my tertiary student life.

This thesis would have been impossible to complete without the support of my wife, Zoun, who had to put up with my strange schedules and stresses with the tremendous pressure of her own work. I would also like to thank my parents and friends here and back home for their encouragement and prayers from a distance.

Finally, I want to remember my sister and my dearest one, whom I lost last year – they are not here but I feel their presence and good wishes all the time.

Contents

1 Introduction

Agent methodology represents a new way of analysing, designing, and implementing complex software applications. It offers a powerful model of conceptualisation as well as implementation of many programming problems. This approach is being successfully utilised in complex applications with distributed components, which require concurrent behaviour, efficient reasoning, dependable communication, sharing and integration of knowledge, and of course, intelligence to some extent. Intelligent agents are currently being used for modelling simple rational behaviours in a wide variety of distributed and centralised applications – ranging from comparatively small systems such as personalised email filters to large, complex, mission critical systems such as air traffic control [JSW98]. Since its introduction, various approaches have been proposed to determine a suitable architecture for agent-based systems. There have been a number of attempts to model *Agent-Oriented Programming(AOP)* languages. AOP can be viewed as a specialisation of *Object-Oriented Programming(OOP)* [Sho93]. AOP languages define agents, their properties, behaviours, and states, and obviously, programs to control the agents.

On the other hand, linear logic, introduced by Girard in 1987 [Gir87], has become an area of interest of many researchers, as it provides an efficient methodology for constructing tools that can be successfully exploited in modelling updates, reasoning about the environment, and implementing concurrent behaviour. Linear logic was designed to reason about bounded resources. As a consequence of its resource-sensitive nature, linear logic has been the basis for a number of programming languages including Lygon [WH95, WH96], Lolli [HM94], Forum [Mil94], ACL [KY93], $\mathcal{LC}$ [Vol94], and LO [AP91]. These languages were used to solve problems requiring concurrency, updates, knowledge representation, logical interpretation of actions and graph search algorithms [Win97, HPW96, WH96]. Some of the features of linear logic have motivated us to use it in AOP. We have used Lygon programming language in this project.

This thesis focuses on the investigation of how a linear logic programming language, in this instance Lygon, can be utilised in AOP. The research goal is not devising a new agent-based system like JACK [BRHL98] or dMARS [AAI96]; nor is the goal writing a compiler or interpreter to extend the capabilities of Lygon. The main goal of the project is to investigate the possibilities of creating agents with linear logic. As a part of the goal, we created some agents in Lygon and observed their behaviours and complexity as well as efficiency both in runtime and programming time.

The contributions of this thesis are as follows:

1. We perform a literature review, which includes agent theory, AOP, linear logic, and Lygon.
2. We look at how agents are implemented in some well-known AOP languages. We investigate the relationship of linear logic with AOP framework.
3. Based on the literature survey and investigation we propose a simple framework of agents in linear logic.
4. We implement the framework in Lygon and use it to construct some agent based programs.

5. We analyse our experience in programming agents with Lygon and point out areas of future work.

The thesis starts with some background in section 2. This section explains the foundations of the three principal areas of the research – *agent theory, linear logic* and *Lygon*. Section 3 provides information about a well known agent architecture and some existing AOP languages. Section 4 describes our proposed linear logic framework of agents. Section 5 outlines the experiments carried out with Lygon and an analysis of the results. A brief overview of related work is included in section 6. Based on the experience in section 5.2, a number of areas of future work have been identified. Section 7 describes these and our conclusions.

2 Foundations

2.1 Agent-Theory

The objective of Agent Oriented (AO) technology is to build systems applicable to the real world that can observe and act on changes in the environment. Such systems must be able to behave rationally and autonomously in the completion of their designated tasks [WJ95].

AO technology is an approach for building complex real-time distributed systems. This technology is built on the belief that a computer system should be designed to exhibit rational, goal-directed behaviour similar to that of a human being. AO technology achieves this by building entities called agents which are purposeful, reactive, communication-based and sometimes team-oriented.

An agent emulates rational behaviour in that it has intentions, which it forms according to its beliefs and goals [RG91, BIP88]. According to the BDI model, an agent uses pre-defined plans, which are applicable to a situation, to fulfil its intentions. Agent-oriented systems consist of a collection of agents, possibly very large in number. Each agent in the system responds to events generated by other agents or the environment in which the agents are situated.

This section will give an overview of agent theory. First, we will look at different definitions of the term *agent* and then, discuss briefly different agent architectures, and finally, we will give some examples of agent-based applications.

2.1.1 Definition of an Agent

> The question *what is as agent?* is embarrassing for the agent-based computing community in just the same way that the question *what is intelligence?* is embarrassing for the mainstream AI community.

Carl Hewitt made this remark at the thirteenth international workshop on distributed AI. The problem is that although the term *agent* is widely used by many people working in closely related areas, it defies attempts to produce a single universally accepted definition [WJ95].

Generally, the term *agent* is used to denote a hardware or (more usually) software-based computer system with the following properties:

- *Autonomy*: The ability to operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.
- *Social Ability*: The ability to interact with other agents (and possibly humans) via some kind of *agent-communication language*.
- *Reactivity*: The ability to perceive the environment[1], and respond regularly to changes that occur in it.
- *Pro-activeness*: The ability to exhibit goal-directed behaviour by taking the initiative instead of just acting in response.

Wooldridge and Jennings [WJ95] conceptualise an agent as a kind of UNIX-like software process, that exhibits the properties listed above. From the viewpoint of AI researchers an agent is a computer system that, in addition to having the attributes mentioned above, is either conceptualised or implemented using concepts that are more usually applied to humans, such as *knowledge, belief, intention,* and *obligation* (and sometimes *emotion*) [Sho93]. Some other attributes of agency are:

- *Mobility*: The ability to move around an electronic network.
- *Veracity*: The assumption of not communicating false information knowingly.
- *Benevolence*: The assumption of not having conflicting goals.
- *Rationality*: The assumption of acting with a view to achieve its goals, instead of preventing them.

2.1.2 Agent Architectures

This section will try to give an overview of how agent theories have been approached in practice. Various methodologies have been suggested for building agents. According to Maes [Mae91], an agent architecture is a particular methodology which specifies how the construction of an agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interaction has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions and future internal state of the agent.

There are three classes of architectures:

1. *Deliberative Architectures* (or Classical Approach)
2. *Reactive Architectures* (or Alternative Approach)
3. *Hybrid Architectures*

[1]By environment we mean the physical world, a user via GUI or some sort of interface, a team or collection of other agents, the Internet, or perhaps some (possibly all) of these combined

Deliberative Architectures contain an explicitly represented, symbolic model of the world in which decisions (such as what actions to perform) are made via (logical) reasoning, based on pattern matching and symbolic manipulation [WJ95]. This approach builds agents as a type of knowledge-based system. The main problem with this type of architecture is performance. The architecture needs to translate the real world into a correct symbolic description and the agents need to reason with this information quickly enough for the results to be useful. This problem leads to work on vision, speech understanding, learning, knowledge representation, automated reasoning, planning, etc. Examples of deliberative agent architectures include Intelligent Resource-bounded Machine Architecture (IRMA) [BIP88], HOMER [VB90], and GRATE* [Jen93].

Reactive Architectures are based on the assumption that intelligent agents behaviour can be generated *without* an explicit representation and abstract reasoning of the kind that symbolic AI proposes and is an *emergent* property of certain complex systems. Brooks [Bro91] identifies two key ideas:

1. Situatedness and embodiment: "Real" intelligence is situated in the world, not in disembodied systems such as theorem provers or expert systems.

2. Intelligence and emergence: "Intelligent" behaviour arises as a result of an agent's interaction with its environment.

Brooks' approach is known as subsumption architecture. Examples of reactive architectures include PENGI [AC87], situated automata [Kae91] and ANA [Mae91].

Hybrid Architectures are designed by the product of the marriage of the two approaches discussed so far. Both the classical and the alternative approach to agent architecture have their own disadvantages, hence neither a completely deliberative nor completely reactive approach is suitable for building agents.

Obviously, a *hybrid* agent is built out of two (or more) subsystems: a deliberative one, containing a symbolic world model, which develops plans and makes decisions in the way proposed by mainstream symbolic AI; and a reactive one, which is capable of reacting to events that occur in the environment without engaging in complex reasoning. Often, the reactive component is given some kind of precedence over the deliberative one, so that it can provide a rapid response to important environmental events. TOURINGMACHINES [Fer92], INTERRAP [MPT95] and PRS [GL87] are instances of hybrid agent architectures.

The details of the example architectures that have been mentioned in this section are beyond the scope of this thesis. However, a brief description of the PRS (Procedural Reasoning System) is included in section 3.1 on page 14.

2.1.3 Multi-Agency

Multi-Agent Systems (MAS) refer to all types of systems composed of multiple (semi-) autonomous components. A related area is Distributed Problem Solving (DPS), where a particular problem is solved by a number of modules (nodes), which cooperate in dividing and sharing

knowledge about the problem and its evolving solutions. In DPS systems, all interaction strategies are incorporated as an integral part of the system. On the other hand, research in MAS is concerned with the behaviour of a collection of possibly pre-existing autonomous agents aiming at solving a given problem. MAS can be defined as loosely coupled network of problem solvers that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver. These problem solvers agents are autonomous and may be heterogeneous in nature. According to [JSW98], the characteristics of MAS are:

1. Each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint;
2. There is no global system control;
3. Data is decentralised; and
4. Computation is asynchronous.

The success of MAS depends on a number of factors. Firstly, there must be a good way to formulate, describe, decompose, and allocate problems and synthesise results among a group of intelligent agents. Secondly, comes the communication between the agents that depends on the language and protocols used. Thirdly, coherence in making decisions or carrying out actions, avoidance of harmful interactions, and recognition and reconciliation of conflicting viewpoints. Finally, coordination plays an important role in multi-agent systems. All these criteria as well as limited resources makes the task of engineering a practical MAS system more complicated and challenging.

2.1.4 Agent Applications

Current and potential applications of agent technology include [JSW98]

Area	**Example**
Industrial	Manufacturing, Process Control, Telecommunications, Air Traffic Control, and Transport System
Commercial	Information Management, E-Commerce, and Business Process Management
Entertainment	Games, Interactive Theatre, and Cinema
Medical	Patient Monitoring, Health Care

2.2 Linear Logic

In 1987 Jean-Yves Girard introduced *Linear Logic (LL)* [Gir87]. From the start it was recognised as relevant to issues of computation, evidence of which is that the paper appeared not in a journal of logic, but in *Theoretical Computer Science* [Ale94]. Linear Logic is a new constructive logic, following the development of intuitionistic logic. It is a refinement of classical logic and sometimes is described as *resource sensitive* because it provides an intrinsic and natural accounting

of resources. In this section we will discuss the resource sensitivity of linear logic followed by a general overview of its connectives and applications to different areas of computation. In parallel we will endeavour to point out why linear logic would be beneficial to AOP.

2.2.1 Logic of Resources

Linear logic is a logic of resources, which makes it directly applicable to many computer science tasks, including concurrency, updates, and natural language processing [Ale94]. Linear logic inherits all the properties of classical logic and in addition it has some important characteristics that are not present in classical logic. The main point of difference is that in linear logic (usually) each formula must be used exactly once in a proof; a formula can be neither ignored nor copied. This property is achieved by the elimination of the *weakening* and *contraction* rules in linear logic. This special property of linear logic gives us a more realistic model of resources in the world.

For example, if a single ball is represented by the predicate `ball`, then the property of having two balls may be specified by the conjunction of `ball` with itself, i.e. `ball` $\otimes$ `ball` (pronounced as "ball **cross** ball"). Classical logic represents this by `ball` $\wedge$ `ball`, which is equivalent to and hence indistinguishable from one ball. This happens due to the presence of the weakening rule in classical logic, but in this kind of instance, this rule generates totally absurd results. On the other hand, in linear logic `ball` $\otimes$ `ball` is not equivalent to `ball`, which is more realistic. Different amounts of the same thing are considered to be different and that is why linear logic is often described as a *logic of resources* rather than a *logic of truth* (such as classical logic) [WH96]. However, linear logic is flexible enough to recapture the classical features by using special operators. We will discuss the use of operators in the next section.

Linear logic allows controlled duplication and deletion of resources that makes itself more versatile and rich. Here, predicates are resources and they are produced and consumed during reasoning. The two types of predicates in linear logic are *p* and $p^{\perp}$ (known as "p-cap" or "p-perp"), representing the consumption and supply of the resource *p* respectively. $(.)^{\perp}$ plays a similar role to $\neg$ in classical logic. For instance, $(x^{\perp})^{\perp} \equiv x$.

In linear logic the rules of general contraction and weakening are removed that make it different to classical logic. Classical logic's contraction rule denotes that if some goal is derivable from *x, x* then it can also be derivable from *x*. This rule allows uncontrolled duplication.

$$\frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta}$$

Contraction Rule : Left

On the other hand, the weakening rule demands that if some goal is derivable from *x*, then the same goal can be derived from *x, y*, which allows resources to disappear.

$$\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta}$$

Weakening Rule : Left

As the contraction rule is removed, resources may be consumed at most once. They removal of weakening rule forces the resources to be consumed at least once (not to disappear). Combining these two conditions, resources must be consumed exactly once. Therefore, resources in linear logic may only be *supplied* or *consumed* as they may not generally be copied or ignored. Linear logic is capable of counting clauses, which is more useful in finding solutions to some problems than other logics [Win97, HPW96].

2.2.2 Operators and Constants

As predicates are countable resources, the normal logical connectives are extended. The *additive* connectives require that the resources not be shared between the sub-proofs, while the *multiplicative* connectives require that the resources are shared [Sce93]. The table below briefly describes the connectives:

Connectives	**multiplicative**	**additive**	**exponential**
conjunction	$\otimes$ times	$\&$ with	! ofCourse
disjunction	$\parr$ par	$\oplus$ plus	? whyNot
implication	$\multimap$ lollipop		

1. $x \otimes y$ denotes that the available resources are split between *x* and *y* , and *x* and *y* then form separate sub-proofs. For instance,

   left-hand, right-hand $\vdash$ *press-string* $\otimes$ *strum*

   would mean that in playing guitar the left-hand could be used to press-string to the fret board while right-hand could be used strum a rhythm (or the other combination), but right-hand could not be used to both press string and strum. It is noticeable in the above example that, $\otimes$ permits concurrency in linear logic, as (*left-hand* $\vdash$ *press-string*) and (*right-hand* $\vdash$ *strum*) may be solved independently.

2. $x \,\&\, y$ duplicates the available resources, such that they must be consumed by *x,* and also be consumed by *y* . For example,

   thesis $\vdash$ *student1* $\&$ *student2* $\&$ ... $\&$ *studentN*

   could mean that all N number of students must complete (or consume) *thesis*. This operator also provides concurrency, as (*thesis* $\vdash$ *student1*), (*thesis* $\vdash$ *student2*),..., and (*thesis* $\vdash$ *studentN*) can be solved concurrently.

3. $x \oplus y$ is similar to $\vee$ in classical logic, requiring that all the resources be consumed by either by *x* or by *y*. For instance,

$$\textit{dollar} \vdash \textit{tea} \oplus \textit{coffee}$$

means that a dollar could be spent by buying (a cup of) tea or coffee, not both.

4. $x \parr y$ is the dual of $x \otimes y$. Where $x \otimes y$ splits the context between *x* and *y*, $x \parr y$ just adds *x* and *y* to the context. This implies that with $\parr$ the operands may share resources between them, whereas with $\otimes$ they may not. For example,

$$\textit{achieve-honours} \vdash \textit{find-proposal} \parr \textit{research} \parr \textit{write-thesis} \parr \textit{complete-course work}$$

indicates what is required to *achieve-honours*, but *write-thesis* is not possible without a proposal being found and researched, produced by *find-proposal* and *research* respectively.

Each connective has an accompanying unit. The Table below shows the name and symbol of the four constant units of the linear logic operators.

Constants	**Name**	**Connective**	**Context**
$\top$	Top	$\&$	Provable in any context
1	One	$\otimes$	Provable only in empty context
$\bot$	Bottom	$\parr$	Cannot be proved, but can be weakened away
0	Zero	$\oplus$	Not provable

Linear logic also contains a negation, which behaves similarly to classical negation. The negation of a formula *F* is written as $F^{\bot}$. The following de Morgan rules are valid in linear logic:

$$\begin{gathered}
(p \otimes q)^{\bot} \equiv p^{\bot} \parr q^{\bot} \\
(p \parr q)^{\bot} \equiv p^{\bot} \otimes q^{\bot} \\
(p \oplus q)^{\bot} \equiv p^{\bot} \,\&\, q^{\bot} \\
(p \,\&\, q)^{\bot} \equiv p^{\bot} \oplus q^{\bot} \\
(!F)^{\bot} \equiv ?(F)^{\bot} \\
(?F)^{\bot} \equiv !(F)^{\bot} \\
(\forall x F)^{\bot} \equiv \exists x (F)^{\bot} \\
(\boldsymbol{1})^{\bot} \equiv \bot \\
(\bot)^{\bot} \equiv \boldsymbol{1} \\
(\boldsymbol{0})^{\bot} \equiv \top \\
(\top)^{\bot} \equiv \boldsymbol{0}
\end{gathered}$$

These de Morgan rules allow $(.)^{\perp}$ to be pushed to the level of atoms, which simplifies the task of finding a linear logic proof by automatic means – only $atom^{\perp}$ needs to be handled, we don't have to worry about the negation of an arbitrary formula.

To recapture the contraction and weakening property of classical logic the exponential operators '!' and '?' are used. The '!' (*of course*) operator provides an infinite supply or consumption of resource while the '?' (*why not*) operator indicates the *possibility* of endless resources. For example, ?*candy* can be interpreted as someone who is content with any number of candy including zero. On the other hand, !*candy* represents someone who is only happy if given an infinite amount of candy. In addition, ?*candy*$^{\perp}$ means potentially infinite production of candies and !*candy*$^{\perp}$ represents the forced consumption of an infinite amount of candy.

The operator '$\multimap$' (*lollipop*) works as a linear implication operator. In linear logic, $p \multimap q$ is equivalent to $p^{\perp} \parr q$. The existential operators of classical logic: $\forall$ (for all) and $\exists$ (exists) are also included in linear logic.

More information on linear logic and its applications to computer science can be found in [Ale94, Sce95, Sce93].

2.2.3 Linear Logic Example

We know that having two dollars to spend means that it is possible to buy up to two dollars worth of goods but not more. Suppose, a pie and a can of soft drink cost a dollar each. If someone has two dollars, this can be expressed as:

$$\texttt{dollar} \otimes \texttt{dollar}$$

There can be four possibilities[2]:

$\texttt{dollar} \otimes \texttt{soft-drink}$: Buying a can of soft drink and keeping a dollar in hand
$\texttt{dollar} \otimes \texttt{pie}$: Buying a pie and keeping a dollar in hand
$\texttt{pie} \otimes \texttt{soft-drink}$: Buying a can of soft drink and a pie
$\texttt{dollar} \otimes \texttt{dollar}$: Not buying anything, keep all the money

It is straightforward to capture this scenario in linear logic. The conversion of a dollar into a soft drink or a pie can be represented as:

$$\texttt{dollar} \multimap \texttt{soft-drink}$$
$$\texttt{dollar} \multimap \texttt{pie}$$

And the rule that a person with a can and a pie is satisfied can be written as:

$$(\texttt{soft-drink} \otimes \texttt{pie}) \multimap \texttt{satisfied}$$

If we include a rule that a person is partially satisfied if he/she can buy a can of drink or a pie, it will look like:

[2]Since $\otimes$ is commutative $\texttt{pie} \otimes \texttt{dollar}$ is equivalent to $\texttt{dollar} \otimes \texttt{pie}$.

$$(\texttt{soft-drink} \oplus \texttt{pie}) \multimap \texttt{partially-satisfied}$$

From the rules mentioned above we can deduce that:

$$\vdash \texttt{dollar} \multimap \texttt{partially-satisfied}$$
$$\nvdash \texttt{dollar} \multimap \texttt{satisfied}$$
$$\vdash \texttt{dollar} \otimes \texttt{dollar} \multimap \texttt{satisfied}$$

Another complex but interesting example can be found in [HPW96]. It is the *restaurant menu* problem. Suppose, a restaurant has the following menu:

```
entree :  fruit or seafood (in season)
              main course
                desserts
      hot drink :  tea or coffee
```

From the point of view of a customer, the menu corresponds to the formula:

$$\texttt{entree} \otimes \texttt{main} \otimes \texttt{dessert} \otimes \texttt{hot drink}$$
or
$$(\texttt{fruit} \oplus \texttt{seafood}) \otimes \texttt{main} \otimes \texttt{dessert} \otimes (\texttt{tea} \,\&\, \texttt{coffee})$$

The decision about whether the entree is fruit or seafood will be made by the chef, depending on what is in season, price etc., and so for the customer this choice is external. On the other hand, the choice of hot drink is always an internal choice for the customer as a restaurant can provide both of them anytime.

From the restauranter viewpoint this would look like:

$$(\texttt{fruit}^{\perp} \,\&\, \texttt{seafood}^{\perp}) \parr \texttt{main}^{\perp} \parr \texttt{dessert}^{\perp} \parr (\texttt{tea}^{\perp} \oplus \texttt{coffee}^{\perp})$$

Each formula here is negated, according to de Morgan's rule, as the restauranter has to *supply* each of the named items, rather than acquire each one. That is why, the courses are joined together with $\parr$, rather than $\oplus$. The internal and external choices are also swapped.

2.3 Lygon: A linear logic programming language

Lygon[3] is a logic programming language based on linear logic [Win97, WH95, WH96], building on the theoretical framework described by Harland and Pym [PH94]. An interpreter and a debugger for Lygon were implemented by Michael Winikoff [Win97, Win96] and Yi Xiao Xu [Xu95] respectively.

[3]The name is taken from a street called "Lygon Street" in Melbourne which is known for its restaurants and cafes. There is also a "Lygon Road" in Edinburgh which is familiar to Harland and Pym.

2.3.1 Basics

In common with other linear logic programming languages, *Lygon* allows resources to be used exactly once in a computation, which allows the elegant specification of programs which are somewhat more awkward in a language like Prolog [CM84, SS86, SS94]. However, just as linear logic is a strict extension of classical logic, Lygon is a strict extension of (pure) Prolog. The design of Lygon was guided by the slogan *"Lygon = Prolog + Linear Logic"*. Therefore, all (pure) Prolog programs can be executed by the Lygon system. Hence all the features of pure logic programs are retained in Lygon, and extended with new ones based on linear logic. Such features include global variables, a notion of state, mutual exclusion operators and various constructors for manipulating clauses.

Lygon syntax is similar to Prolog with the main difference that goals and the bodies of clauses are a (subset) of linear logic formulae rather than a sequence of atoms. Program clauses are assumed to be reusable (non-linear) and it is possible to specify program clauses to be consumed exactly once in each query by prefixing the clause with the keyword `linear`. As the linear logic connective symbols are not available (usually) in computer keyboards, Lygon uses some special ASCII characters instead. The table below defines the mapping $\Longrightarrow$ between logical connectives and ASCII.

$\otimes \Longrightarrow$ `*`	$\& \Longrightarrow$ `&`	$\parr \Longrightarrow$ `#`
$\oplus \Longrightarrow$ `@`	$\multimap \Longrightarrow$ `<-`	$-^{\perp} \Longrightarrow$ `neg-`
$\mathbf{1} \Longrightarrow$ `one`	$\top \Longrightarrow$ `top`	$\perp \Longrightarrow$ `bot`

A Lygon program consists of a number of clauses, written $D \leftarrow G$ where D specifies the resources that must be present for this clause to be applicable. These resources are consumed and replaced by G.

The grammar for the Lygon language can be summerized as:

$$
\begin{aligned}
G ::= {} & G \otimes G \mid G \oplus G \mid G \,\&\, G \mid G \parr G \mid \,!G \\
& \mid \texttt{neg}\, D \mid \mathbf{1} \mid \perp \mid \top \mid A \mid \texttt{neg}\, A \\
D ::= {} & [\texttt{linear}](A_1 \parr A_2 \parr \ldots \parr A_n \texttt{ <- } G)
\end{aligned}
$$

The execution of a Lygon program consists of a repeated two step cycle [Win97, Win96]:

1. Select a formula to be reduced
2. Reduce the selected formula

This cycle iterates until there are no goals left. Lygon interpreter uses a top-down execution model.

As discussed in [Win97], a Lygon goal may consist of a number of formulae. Generally the choice of a formula to be reduced first may make a difference in finding a proof. The interpreter attempts to reduce this nondeterminism where possible using known properties of linear logic.

The rules below show the reduction process in Lygon. C denotes the linear context and "*Look for a proof of* . . ." means that the goals indicated *replace* the goals being reduced. The rules are:

- *A* * *B* : Split *C* into *C1* and *C2* and look for proofs of (*C1*, *A*) and (*C2*, *B*).
- *A* # *B* : Look for a proof of (*C*, *A*, *B*).
- *A* & *B* : Look for proofs of (*C*, *A*) and (*C*, *B*).
- *A* @ *B* : Look for either a proof of (*C*, *A*) or a proof of (*C*, *B*).
- `top` : The goal succeeds.
- `bot` : Look for a proof of (*C*).
- `one` : The goal succeeds if the context *C* is empty.
- *A* : *resolve*. For more information please see [Win97].
- ? `neg` *A* : Add *A* to the program and look for a proof of (*C*).
- !A : If *C* is empty then look for a proof of (*A*) otherwise fail.

An example can be found on page 161 of [Win97].

2.3.2 Example Lygon Programs

We are including some interesting Lygon programs in this section. The example programs will demonstrate Lygon's basics, concurrency, and state update capabilities. All the Lygon programs include the *Lygon Standard Library* which can be found in Appendix A. All these programs were written by Michael Winikoff [Win97] and are used here with his permission. It should be mentioned that these programs are being used merely as examples to introduce the Lygon language and are not intended to be an original contribution.

Basic: *Toggling State*
This very simple program uses the linear context to store some state information. Program 1 stores a single bit of information and toggles it. The effect of a call to *toggle* in a context containing the fact *off* is to consume the fact and add the fact *on*.

```
toggle <- (off * neg on) @ (on * neg off).

go <- neg off # toggle # show.

show <- off * print('off').
show <- on * print('on').
```

Program 1: Toggling State

Concurrency: *Communicating Processes*

Program 2 shows how simple concurrent programming specifying multiple processes can be implemented in Lygon. A goal of the form `P # Q` evolves `P` and `Q` concurrently. Communication between two processes can be achieved by adding and removing atoms.

```
go(N) <- produce(N) # consume(0) # ack.

produce(N) # ack
       <- lt(0,N) * is(N1,N-1) * (mesg(1)#produce(N1)).
produce(0) # ack <- finished.

consume(N) # mesg(X)
       <- is(N1,N+X) * (consume(N1) # ack).
consume(N) # finished <- print(N) * nl.
```

Program 2: Communicating Processes

A simple call *go(5)* in program 2, denotes that under appropriate conditions an *ack* and a *produce(5)* will evolve to *produce(4)* and a *mesg(1)*. The *consume(0)* and the *mesg(1)* will evolve to *consume(1)* and *ack*. Finally, when *produce(N)* is called with N = 0 with an *ack*, it produces the atom *finished*, which will collaborate with *consume(N)* and complete the program by printing N followed by a new line.

Modelling Actions and State Update: *Yale Shooting Problem*

The Yale shooting problem [HM87] involves action and of course, update of states. Loading a gun changes the gun's state from unloaded to loaded, and shooting updates its state from loaded to unloaded. Shooting at a live turkey changes the turkey's state from alive to dead.

```
shoot <- alive * loaded * (neg dead # neg unloaded).
load <- unloaded * neg loaded.
start <- neg alive # neg unloaded.
```

Program 3: Yale Shooting Problem

Program 3 models this by having predicates *alive, dead, loaded,* and *unloaded* as states and *load* and *shoot* as actions. Here, initially the state asserts turkey to be *alive* and gun to be *unloaded*. The actions follow the corresponding rules.

Lygon is suitable for writing programs to examine graphs as well. Finding paths, cycles could easily be modelled by Lygon, but we are not discussing them here as they are out of the scope of this thesis. A good overview of Lygon's applications can be found in [HPW96, WH96].

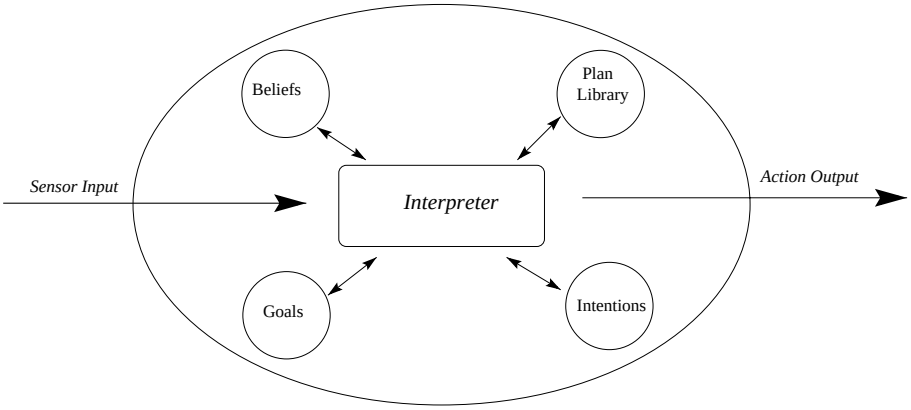

Figure 1: A BDI Agent Architecture: PRS

3 Agent-Oriented Programming

The term Agent-Oriented Programming (AOP) was first introduced by Yoav Shoham of Stanford University. This new computational framework was presented as a specialisation of object-oriented programming [Sho93]. Shoham proposed that agents consist of *mental states* such as beliefs, decisions, capabilities, and obligations and are controlled by agent programs, which provides communication primitives among the agents.

As agent technology becomes better known and established, an increasing number of software tools are available for the development of agent-oriented systems. Recently a number of agent languages have emerged, from *prototype* to *industrial* versions. In this section, we will have an overview of a popular agent-model architecture followed by brief descriptions of some existing agent languages.

3.1 PRS Architecture

The *Procedural Reasoning System (PRS)* is one of the best known agent architectures proposed so far. Originally introduced by Georgeff and Lansky [GL87] PRS is an example of the currently popular paradigm known as the *belief-desire-intention* (BDI) model of agents. The BDI model [BIP88] consists of four data structures: *beliefs*, *goals*, *intentions* and a *plan library* (Figure 1).

Belief is what an agent knows about the world, which may be incorrect or incomplete. Beliefs can be represented as a variable or as symbols (like Prolog facts). *Desires* or *goals* are the tasks allocated to an agent. Unlike human beings desires should be logically consistent. In the BDI model, an agent generally will not be able to achieve all its desires, even if they are consistent. As a result agent must fix upon some subset of available desires and commit resources to fulfil them. These subset chosen desires/goals are called *intentions*. Usually, an agent will continue to work towards achieving an intention until either it believes the intention is satisfied, or no longer possible [dKLW97]. Each agent in a BDI model has a *plan library*, which is a set of plans. Plans are like a recipes, specifying courses of action that may be undertaken by an agent in order to

achieve its intentions.

3.2 AOP Languages

3.2.1 AGENT-0: A simple logical approach

AGENT-0 [Sho93] is an extremely simple instance of the generic agent interpreter. The key idea is to directly program agents in terms of the mentalistic, intentional notions that have been developed to represent the properties of agents. This is Shoham's first attempt as an AOP language.

The logical component of AGENT-0 is a quantified multi-modal logic, allowing direct reference to time [WJ95]. The logic contains three modalities: belief, commitment and ability. An example of an acceptable formula of AGENT-0 may be:

$$CAN^{4}_{pele} pass(ball)^{9} \Longrightarrow B^{4}_{maradona} CAN^{10}_{maradona} score(goal)^{11}$$

This means, if at time 4 *pele* can ensure that he will pass the ball at time 9, then at time 4 *maradona* believes that at time 10 he (*maradona*) can ensure to score a goal at time 11.

In AGENT-0, an agent is specified in terms of a set of capabilities, a set of initial beliefs and commitments, and a set of commitment rules, which decide how the agent acts. Each commitment rule contains a *message condition*, a *mental condition*, and an *action*. When an agent recieves a message, it is matched against the message condition, mental condition is matched against the agent's beliefs and an action takes place by firing a corresponding commitment rule. Two types of action available: *private*, corresponding to an internally executed subroutine; and *communicative,* corresponding to exchange messages. Messages can be one of the four types:

- *Request*: to perform actions.
- *Unrequest*: to cancel a request.
- *Inform*: to pass on information.
- *Refrain*: to refrain from an action.

Each agent performs the following two tasks after certain interval (of time):

1. *Update Beliefs:* Read current messages, update mental states
2. *Update Commitments:* Execute commitments for the current time

For more information on AGENT-0 including a BNF description of the language, please refer to [Sho93].

3.2.2 dMARS: A BDI approach

dMARS (*Distributed Multi-Agent Reasoning System*) is a well-known multi-agent system. The current C++ implementation evolved from a LISP prototype of the PRS architecture.

dMARS agents monitor both the world and their internal state, and any events that are perceived are placed on an *event queue*. The basic execution model of dMARS is the following cycle [dKLW97]:

1. Observe the world and the agent's internal state, and update the event queue to reflect the events that have been observed.

2. Generate new possible desires (tasks), by finding plans whose trigger event matches an event in the event queue.

3. Select from this set of matching plans one for execution (an intended means).

4. Push the intended means onto an existing or new intention stack, according to whether or not the event is a subgoal.

5. Select an intention stack, take the topmost plan (intended means), and execute the next step of this current plan: if the step is an action, perform it; otherwise, if it is a subgoal, post this subgoal on the event queue.

dMARS has been used as the development platform for a number of applications, including simulations of tactical decision-making in air operations and air traffic management.

More information related to dMARS can be found in [dKLW97, AAI96].

3.2.3 JACK: A Java based light weight BDI approach

JACK Intelligent Agents^{TM} is a framework designed by Agent Oriented Software Pty. Ltd. This brings the concept of intelligent agents into the mainstream of commercial software engineering and Java. JACK was developed as a set of light-weight components with strong data typing.

JACK supports the BDI model of agents. According to [BRHL98] it contains three extensions to Java:

1. A set of syntactical additions to the host language have been made. The addition includes:

   - Few keywords (such as *agent*, *plan*, *event*, etc) for the identification of the main components of an agent.
   - A set of statements for the declaration of the attributes (which are strongly typed) and other characteristic of the components, for the definition of static relationship (which plan to adopt to react to a certain event), and for the manipulation of an agent's state (such as addition of new goals or subgoals, changes of beliefs, communication with other agents).

   Furthermore, Java statements can be used inside the components of an agent.

2. A compiler has been implemented that converts the syntactic additions described above into pure Java classes and statements that can be loaded with, and be called by, other Java code.

3. A set of classes known as the *kernel* provides the required run-time support to the generated code. This includes automatic concurrency management, default behaviour of the agent in reaction to events, failure of actions and tasks, and a native light-weight, high performance communications infrastructure for multi-agent applications.

JACK is called light-weight in the sense that its kernel supports multiple agents within a single process, this is beneficial to save system resources. In our research we programmed an agent in JACK and compared it with a Lygon version in section 5.1.2. Further information on JACK can be found in [BRHL98] or at `http://www.agent-software.com.au`.

4 A Linear Logic Framework of Agency

Based on our discussion in previous sections, it is clear that modelling agents and agent based systems requires efficient reasoning, concurrency, reactivity, and obviously, state updates. The *declarative* languages, such as Prolog, offer simple formal semantics that is easy to reason about and implement. However, Prolog has some defects, such as not allowing for concurrency or updates.

Linear logic solves some of the problems of the declarative languages. It is capable of modelling updates and it can model concurrent behaviour cleanly [Win97]. As reasoning and concurrency are vital for designing agents, we explored modelling agents with linear logic, which can offer these facilities. We are using a logic programming approach of linear logic via Lygon.

The program examples we have seen so far have shown how Lygon can model updates and handling concurrency. In the next section we will look at more Lygon programs that express agent-like properties.

According to the history of agent research, one of the early models of agents was the *actors* model [Agh90]. By definition, actors are self-contained, interactive, autonomous components of a computing system that communicate by asynchronous message passing. A system can *create* an actor from a behaviour description and a set of parameters, *send* a message to an actor, and *update* an actor's local state.

Although actors are a natural basis of many kinds of concurrent computation, they face a number of problems. One of these is the low-level granularity, which relates to the composition of actor behaviours in larger communities and achievement of higher level performance goals with only local knowledge [JSW98]. Our linear logic agent model has properties similar to the actors model.

Figure 2 shows us our proposed agent model. In linear logic, we can encode an Agent as *agent(ID,State)*, where *ID* is unique for each individual agent and *State* denotes the current state of the agent. A message can be expressed as *message(ID,Args)*, where *ID* is the ID of the recipient *agent* and *Args* is/are the argument(s). When an agent receives a message, it responds

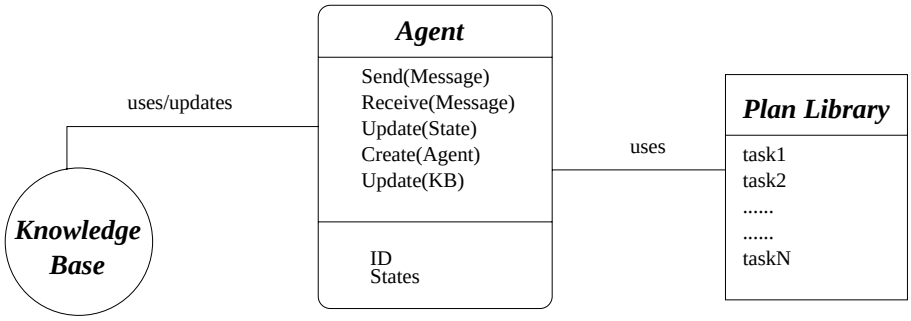

Figure 2: A Simple Agent Model

to it by carrying out some tasks, according to the *Args* and its *State*. Hence, a rule showing how an agent based on the *actors* model responds to a message or behaves can be written as[4]:

agent(ID,State) $\otimes$ message(ID,Args) $\leftarrow$
agent(NewID,SomeState) $\otimes \ldots \otimes$ *Create new agent(s)*
message(SomeID,SomeArgs)$\otimes \ldots \otimes$ *Create new message(s)*
agent(ID,NewState) $\otimes \ldots \otimes$ *Update local state*

For communication purpose *message* is used. A *message* to a certain *agent* fires a certain rule from the *plan library* (where all rules are stored). As a consequence of the firing of a rule, new *message(s)* might be generated (which would lead to firing more rules with the same or different agent), or new *agent(s)* might be created to carry out further tasks. After each *action* (firing a rule), the *agent* updates its knowledge about itself or/and about the world.

This actor-based *agent* model does not explicitly have a *knowledge-base* about the world. However, it is not difficult to add a knowledge-base, where information about the world the agent(s) exists can be stored as facts. In a multi-agent system, agents can share a common knowledge-base. In linear logic, a knowledge-base can be represented by a list and be supplied in the driver main rule in the form $KB(List)^{\perp} \otimes \ldots \otimes \ldots$ and can be consumed in the rule as:

$$agent(ID, State) \otimes message(ID, Args) \leftarrow KB(List) \otimes \ldots$$

In the next section we will perform some experiments with this simple linear logic *agent* model and observe how Lygon fits in with the specification.

5 Implementation

A number of experimental programs have been written in Lygon, using the agent model described in section 4. These programs require the Lygon Standard Library (see Appendix A). All these programs were executed successfully by the Lygon interpreter.

[4]This agent model is based on the help from [Win97] page 191

Basically, these programs were written to observe how agenthood can be implemented in Lygon. We will first describe the agents that have been implemented in section 5.1, which will be followed by a critical discussion of what we have learned from programming agents in Lygon in section 5.2.

5.1 Experiments

5.1.1 Bank Account Agent

The *Bank Account Agent* in program 4 is a simple implementation of the framework defined in section 4. There is only one agent. It has neither any knowledge-base to share nor any other agents to communicate with. It has only its own state, we call it *Balance*. The agent receives a message as a *request* and acts according to the rules defined. The agent has four actions: *Output Balance, Withdraw, Deposit* and *Exit*.

```
bankAgent(ID,Balance) # request(ID,balance(Balance))
                <- bankAgent(ID,Balance).
                %% Output Balance: processing of balance request

% Deposit: agent is requested to deposit N dollar
% N is added to the balance
bankAgent(ID,Balance) # request(ID,deposit(X))
                <- is(NewBalance, Balance + X) *
                        bankAgent(ID,NewBalance).

% Withdraw: agent is requested to withdraw X dollar
% handle situation where Balance >= X, approved
bankAgent(ID, Balance) # request(ID,withdraw(X,yes))
                <- le(X,Balance) *
                is(NewBalance,Balance-X) *
                (bankAgent(ID,NewBalance)).

% handle situation where Balance < X, not approved
bankAgent(ID, Balance) # request(ID,withdraw(X,no))
                <- gt(X,Balance) *
                bankAgent(ID,Balance).

% Exit: agent is requested to exit
bankAgent(ID, Balance) # request(ID,terminate(Balance))
                <- one.
```

Program 4: Bank Agent

In program 4 we define the bank account agent by *bankAgent(ID, Balance)*. Messages are in the form *request(ID, ..type..)*. The *Output Balance* rule simply outputs the *Balance* state of the agent. The *Deposit* rule takes the deposit value from the argument, adds the value to the value of the current balance and finally, updates the agent's *Balance* state with the new balance. We have two *Withdraw* rules. The first rule checks if the requested money is less than or equal to the current balance. If true, it updates the balance by deducting the requested amount from the current balance. The second rule checks if the requested money is greater than the current value. If true, it proves the goal which will eventually declare that the withdraw was not approved. The *Exit* rule just consumes all resources and terminates thre program.

The *Bank Account Agent* can be invoked by program 5. The *bankmain* goal creates a *bank-Agent* with an initial balance of `$0` and the agent receives requests to withdraw `$100`, deposit `$150`, output the balance, and terminate. One thing to mention is that the program does not implement any sequentiality, i.e., the agent can receive requests in any order. Hence, the output of program 5 includes all possible orderings.

```
bankmain <- (bankAgent(id1,0)
                # request(id1,withdraw(100,R)) #
                request(id1,deposit(150)) #
                request(id1,balance(B)) #
                request(id1,terminate(X))) *
        print('Balance before transaction: $') * print(B) * nl *
        print('Withdraw Approved: ') * print(R) * nl *
        print('Balance after transactions: $') * print(X)
                * nl * nl * fail.
        %% main driver program
```

Program 5: Bank Agent Main Program

```
Balance before transaction: $0
Withdraw Approved: yes
Balance after transactions: $50

Balance before transaction: $0
Withdraw Approved: no
Balance after transactions: $150

Balance before transaction: $150
Withdraw Approved: yes
Balance after transactions: $50

Balance before transaction: $50
Withdraw Approved: yes
Balance after transactions: $50
```

```
Balance before transaction: $0
Withdraw Approved: no
Balance after transactions: $150

Balance before transaction: $150
Withdraw Approved: no
Balance after transactions: $150
```

Output of the program 5

5.1.2 Fibonacci Agent

The generation of the Fibonacci series is a classical example of recursion. A recursive function in any programming language can compute the Fibonacci series. Another solution to this problem can be approached by using an agent. We have a JACK [BRHL98] implementation (please refer to Appendix B) of this problem. By choosing this problem we have an opportunity to compare the Lygon version with the JACK one. The agent overheads can be compared in each language on this conceptually simple but potentially inefficient calculation.

The Fibonacci agent has a database to store computed Fibonacci numbers. Instead of calculating Fibonacci from scratch for every input number, the agent computes using the following algorithm:

```
computeFibonacci(N)
          checks the database and
          if (the value exists)
             return the value
          else
             create a new goal: computeFibonacci(N-1)
             create a new goal: computeFibonacci(N-2)
             add the two results return the sum and
             update the database
          endif
```

The JACK implementation in Appendix B has an agent which creates the new goals for itself.

In the agent framework that was proposed in section 4, an agent cannot process a message sent by itself. It can create subgoals for itself, but cannot suspend its current goal processing and wait for messages to be sent from itself. For this reason the Lygon Fibonacci agent approaches this problem in a slightly different way in program 7. Instead of creating the goal for itself, it creates two new agents and requests them to compute the Fibonacci for (N-1) and (N-2). As a consequence, we will end up with a number of agents searching for different Fibonacci numbers. All the agents share a common knowledge-base, which is updated as soon as a Fibonacci value is found.

The Lygon Fibonacci agent code is listed in program 7, which uses some library functions defined in program 6. To program the agent we needed to preserve the sequence of goals. For example, one of the library functions, *updateKB(X,Cont)* implements this. The function updates the KB by the supplied X value and then calls the Cont, which stands for continuation. Hence,

we can pass another goal or a sequence of goals via the *Cont* variable. Thus the goals are forced to execute in order.

```
newconst(ID,Cont)
        <- constid(N) * is(N1,N+1) *
        eq(ID,const(N)) * (neg constid(N1) # call(Cont)).
        %% Create a new ID for the new agent

%% Prints the KB
printKB <- db(KB) * (print(KB)).

%% Update the KB by the value X
updateKB(X,Cont) <- db(KB) * (neg db([X|KB]) # call(Cont)).

%% This will first look up the Knowledge base for the result,
%% if it finds it returns the result, otherwise it returns -1.
%% This binds Res to the result, recreates the linear db fact
%% and calls the continuation.

result(N,Res,Cont)
        <- db(KB) *
                (!once((member(p(N,R1),KB) * eq(Res,R1))
                @ eq(Res,-1))) * (neg db(KB) # call(Cont)).
```

Program 6: Fibonacci Library

The Lygon interpreter does not provide any construct for IF-THEN-ELSE, which is necessary for checking for some condition. In Fibonacci we have simulated the IF-THEN-ELSE via the rule:

```
once((A * B) @ C))
```

which means `IF A THEN B ELSE C`. The library functions in program 6 provides some necessary tools for the agent to compute the Fibonacci series and output the result. The *newconst(ID,Cont)* generates an unique constant ID, which can be used to create a new agent. The *printKB* prints out the knowledge-base. The *updateKB(X,Cont)* updates the KB by the value X and calls the sequence of goals via *Cont*. The *result(N,Res,Cont)* consumes the KB, checks whether the *N*-th value is there or not. If the value exists, it returns the value in the variable *Res*, otherwise it sets the *Res* to -1 (an arbitrary default error value). Finally, the function recreates the KB to be used by other goals.

The agent in program 7 handles some situations. If the *exit(ID)* request received, the agent terminates the program by having all resources consumed. If the request is to calculate the 1^{st} and 2^{nd} Fibonacci value, it outputs 1 without looking at any database or performing any computation (this is the base case). If the request to calculate any other positive value, the agent first looks up the KB for that value via the *result* function. If the return value is -1, it calls the

computeresult(ID,N). This function creates two new agents to calculate two values of the lower indices. The values are achieved separately by following the same procedure (a recursive call) and they are added together to calculate the final value. The KB is updated and result is returned.

```
exit(ID) # fibagent(ID) <- bot.  %% Exit situations: need two
                         %% clauses to fix a bug in Lygon
fibagent(ID) # exit(ID) <- bot.

%% Base situations
fibagent(ID) # calc(ID,1)
   <- updateKB(p(1,1), (fibagent(ID) # neg msgresult(ID,1))).
fibagent(ID) # calc(ID,2)
   <- updateKB(p(2,1), (fibagent(ID) # neg msgresult(ID,1))).

%% Rules for calculating Fibonacci number greater than 2
%% First look up the data base, if not found, compute.
fibagent(ID) # calc(ID,N)
   <- gt(N,2) *
   result(N,Res,
          (eq(Res,-1) * computeresult(ID,N))
        @ (gt(Res,0) * (fibagent(ID) # neg msgresult(ID,Res)))).

%% Create two new agents to find the Fibonacci value
%% for N-1 and N-2, wait for messages carrying results
%% add results, update KB, terminate subgoals and return result
computeresult(ID,N)
   <- newconst(ID1, newconst(ID2,
        (is(N1,N-1) * is(N2,N-2) * (
                fibagent(ID1) # calc(ID1,N1) #
                fibagent(ID2) # calc(ID2,N2) #
        (msgresult(ID1,R1) * msgresult(ID2,R2) * is(R,R1+R2) *
        updateKB(p(N,R), (exit(ID1) # exit(ID2) # fibagent(ID) #
        neg msgresult(ID,R))))))))).
```

Program 7: Fibonacci Agent

A Fibonacci agent can be invoked to compute the tenth Fibonacci number, by calling *fibmain(10,Result)* in the program 8. The result will be stored in the variable *Result* (For this instance, the result will be 55). Note, that the Lygon Fibonacci agent can compute up to the 43^{rd} number in the Fibonacci series. It is worth mentioning that the agent uses two types of resources - constant ID (generated by *newconst(ID,Cont)*) and the KB (generated by *db()*). All constant Id's are consumed by *constid(_)* and the KB is consumed by the *printKB*, which does not recreate the KB.

```
fibmain(N,Res)
        <- (constid(_) * printKB) # (
           msgresult(const(0),Res) * exit(const(0))) #
           fibagent(const(0)) #
           calc(const(0),N) # neg db([]) # neg constid(1).
        %% The Main Driver Function
```

Program 8: Fibonacci Main Program

Comparison of Lygon and JACK Fibonacci agent:
In this section we have implemented the Fibonacci agent both in JACK and Lygon. Both agents compute the same thing (i.e., a Fibonacci series up to a given value), but their way of solving the problem differs in various ways, which gives us an idea of the relative merits of each approach. We can look at differences on a number of levels:

- **Agent model:** The JACK agent model has the capability of creating a new goal for itself and it can suspend its current goal and wait for the new request to be processed and notification of the result. On the other hand, the Lygon agent model that we have proposed in section 4, lacks the ability to send and process requests to itself. That is the reason, we have used a different approach in Lygon.

- **Language:** In Lygon we have developed the agent plans as a collection of clauses. No type checking has been implemented here. Everything is a *formula*, which includes *atoms*. On the other hand, JACK is developed over the Java language. Hence, it inherits all the strong data types Java can offer. Furthermore, JACK has implemented its own type checking, defined special key words to identify an agent, such as *event, database, plan* and *agent*, with associating modules, such as *Event, BDIGoalEvent, CloseWorld, Plan* and *Agent*. The Fibonacci program in Appendix B does not include many goals, but it is not hard to assume that these strong data types and modules would simplify the job in more complicated cases.

- **Performance:** It is not possible to make any comment on the performance of these two different agents just by comparing a single example. However, the factors influencing the performance include not only modelling issues, but also methodology issues (for example, JACK uses object-oriented programming while our model uses logic programming) and background language issues (for example, JACK is implemented in Java, while Lygon was implemented in Prolog).

However, JACK is an extension to Java and our proposed agent model is not an agent-oriented extension to Lygon. Future work could include the development of an AOP interpreter over Lygon with strong type checking and syntax that similar to an industrial implementation like JACK.

5.2 Analysis

By writing code for the agents in previous section, we went through some valuable experience which would be fruitful to implement agent-oriented programming in linear logic via Lygon. We

should mention that by the word *Lygon agent* we mean an agent developed using the framework proposed in section 4 via Lygon. Our observations are:

1. The Lygon agent is sufficient to model concurrent behaviour. The *Bank Account Agent* demonstrates the concurrent behaviour of a Lygon agent successfully.

2. It is a little complicated to apply sequentiality in a Lygon agent. A Lygon agent can be requested to do a number of actions via the format:

   $$\text{agent(ID,State)} \parr \textit{action1} \parr \textit{action2} \parr \ldots$$

   But the problematic point is using the "$\parr$" connective, which allows concurrency but cannot provide any control to its goals. The associated goals can be invoked in any order. This problem can be resolved by using a "Cont(inuation)" variable in requests, for example, the *updateKB(X,Cont)* request in program 6. Another way to solve this problem may be the use of non-commutative linear logic, which will signify the order of actions.

3. A Lygon agent cannot process a message until it has completed processing the current message. A rule can be written as:

   $$\begin{gathered}\text{agent A} \parr \text{message B} \leftarrow \\ \ldots \text{ send message to self} \\ \ldots \text{ wait for response from self} \\ \ldots \text{ continue } \ldots\end{gathered}$$

   Sending message to oneself is possible, but the message cannot be processed until the current plan (processing of the current message) finishes.

   This became clear when we first tried to implement the *Fibonacci Agent*, by firing new goals to compute Fibonacci values of lower indexes for itself. This is the reason, we adopted the approach of creating new Fibonacci agents to compute the lower Fibonacci values and making the previous ones to wait for a message sent by their descendants.

   Sending message to oneself can be regarded as a important feature of agents. To resolve this problem for Lygon agents future work could include modifying the agent framework described in section 4. For example, to send a message:

   $$\begin{gathered}\text{agent(ID)} \parr \text{message(ID,Args)} \leftarrow \\ \text{agent(ID)} \parr \text{agentProcess(ID,Args)}\end{gathered}$$

   the *agentProcess(ID,Args)* could send messages to agent(ID). In effect the agent spawns a thread to perform the computation associated with the incoming message.

   In order to reply to a message from oneself to oneself, rather than replying to an agent, replying to a thread can be modelled. It would not be possible to identify which thread to reply, if a single agent ID is used. A solution may be to introduce thread IDs(TID). Hence the rule becomes:

agent(ID) ⅋ message(ID,Args) ←
agent(ID) ⅋ agentProcess(TID,Args)

4. A knowledge-base shared by a number of agents can be implemented in Lygon. In the Fibonacci example, all the agents were sharing a common knowledge-base which was updated every time a new Fibonacci value was found. One thing to be careful of is that as all the actions are fired in a rule via the "⅋" connective, the knowledge-base is consumed when an action fires. The knowledge-base has to be recreated upon completing the action, otherwise the next action would not have any knowledge-base to use which results in a failure of the entire goal. This observation applies to all the common resources being used in Lygon.

   The problem of losing resources can be handled by recreating after execution. If an interpreter is to build for AOP in Lygon, we suggest some library functions should be defined to solve this frequent problem.

5. Lygon agents need to be killed after execution. When a number of agents exist, we have to take care that each of them terminates (in linear logic's term "are consumed") after the execution, otherwise, the entire goal fails.

   One way to solve this problem is to create a new connective for Lygon, which can consume all the atoms (resources) after the main execution. Lygon's `top` connective consumes all resources, but we cannot use it straight away because we will lose all resources regardless. Some way we have to make sure that the connective only consumes unused resources.

   Another suggestion might be to implement some change in the Lygon interpreter so that it might warn if there are resources that still exist after execution. A more wishful thinking may be to list all the non-consumed resources.

6. The current version of the Lygon debugger is too simple to debug complex programs as agents. While programming agents with Lygon, we had to use the debugger and found that it is very troublesome to locate the bug by the output that debugger makes. A future work might be to design a new user-friendly agent-oriented debugger, if a new AOP interpreter were to be designed in Lygon.

Based on our analysis in this section, some future works are discussed in the section 7.

6 Related Work

To build up an agent framework we have used a logic programming approach of linear logic. However, linear logic can be applied in functional programming (such as LINEAR ML, LILAC, Linear LISP, etc.), state oriented programming or object oriented programming (such as LO - Linear Objects) [Ale94]. As AOP can be taken as a specialisation of OOP, an object-oriented linear logic language might be a useful point to implement agent feature in linear logic.

A multi-theory meta-logic programming technique has been applied to the realisation of multi-agent systems in [MMZ97]. To solve complex problems in heterogeneous software environments, a special language named ACLPL (*Agent Constraint Logic programming Language*) has been developed via a constraint logic programming language *Eclipse*. By extending the standard constraint logic programming, ACLPL defines a world where the global knowledge is divided into theories or knowledge-bases which is used by agents. Theories are made up of facts and rules. ACLPL also provides primitives for communication between agents and update agents' attributes(states).

As an extension to the work described in [MMZ97], a multi-agent system development took place in [BDM+99]. As unfortunately there has been a lack of standard methods to build a desired MAS-based application, the researchers in [BDM+99] tried to present some features of **CaseLP**, which is an experimental, logic programming based, prototyping environment of MAS. Particularly, they outlined a methodology that combines traditional software engineering approaches with consideration from the agent-oriented field. First, a formal and abstract specification of the MAS was given using the linear logic programming language $\mathcal{E}_{hhf}$ and then ACLPL was used to build the prototype closer to a final implementation of the MAS.

$\mathcal{E}_{hhf}$ is an executable linear logic language for modelling concurrent and resource sensitive systems based on Forum [Mil94]. It contains features of logic programming languages like λProlog, such as goals with implication and universal quantification, with the notion of *formulas as resources*. In [BDM+99] $\mathcal{E}_{hhf}$ is used for the specification of the PRS agent. *Beliefs* were modelled by the form *belief(Fact)*. The set of beliefs is maintained in the current state Ω. *Goals* are terms of the form *achieve(Fact)* and *query(Fact)*. An *achieve goal* commits the agent to a sequence of actions to make the goal true. A *query goal* implies a test on the agent's beliefs to know whether the goal is true or not. An $\mathcal{E}_{hhf}$ agent has *internal actions* to update the *beliefs* and *external actions* to communicate and perform other tasks. It has a *plan library* which is a collection of *plans*. *Plan* is a sequence of *actions* under some *context*, invoked by an *event* via a *trigger*, maintained by a condition named *maintenance*, and followed by as sequence of actions. An event queue is associated with each agent which contains external and internal events.

This work is related to our work in the sense that it approached via linear logic to model agent based systems. The contrast is that we looked at the agent-behaviour in Lygon from a basic point of view. $\mathcal{E}_{hhf}$ is implemented to model the PRS agent, on the other hand, we explored to model our framework on the basis of the *actors* model. $\mathcal{E}_{hhf}$ is used for the development of a complex system, where our endeavour was to look at the possibilities of AOP within Lygon. Definitely, the $\mathcal{E}_{hhf}$ specification can work as a guideline for our further investigation of AOP in linear logic via Lygon.

In this thesis, our endeavour was to find the agent features already available in Lygon. Other than the programs included in the thesis, it is worth looking at programs 17, 22 and 24 of [Win97]. Mutual exclusion, which is a very important feature of concurrent behaviour, is implemented in program 17. Program 22 implements the *dining philosophers problem* which shows concurrent behaviour and message passing in a passive way. Finally, program 24, Blocks World, demonstrates some features of planning.

Reactivity is another feature of agents that has not been implemented in Lygon agents in this paper. Lygon, like most other logic programming languages, is implemented using standard top-

down resolution or *backward chaining*. But there are applications for which the use of *forward chaining* or bottom-up techniques within a dynamic environment are appropriate. A combination of forward-chaining and a dynamic environment allows logic programs to be reactive. MixLog [Smi97] demonstrates one approach to implementing reactivity in a logic programming language. Specifically, [HW98] gives a computational framework for making linear logic programs reactive. More research in this area might help the further implementation of agent-based programming in Lygon.

7 Future Work and Concluding Remarks

In this thesis we have explored three major areas. Firstly, we looked at agents and agent-based systems, briefly outlined their definitions, architectures and applications. Secondly, we discussed linear logic; we talked about its resource sensitivity, connectives and constants, and explained its applicability to model the environment through some examples. Finally, we briefly introduced the linear logic programming language Lygon and familiarised ourselves with its semantics and interpreter. We explored a number of Lygon programs that helped us to understand the language and some of its characteristics.

We have discussed some current AOP languages ranging from theoretical logical approaches to industrial implementation. We briefly described a well-known architecture that influenced the modelling of our proposed linear logic framework.

We have investigated existing Lygon programs that showed agent behaviour and implemented our agent framework in Lygon. We also created some agent programs using the framework and analysed the results, as well as comparing one of them with an implementation in an industrial agent framework (JACK). Our analysis has uncovered some underlying truth about Lygon which leads to some future work and possibilities. The future work includes:

- Examination of more agent programs under the current proposed framework outlined in section 4. We implemented two agent programs that explored some behaviour of agent-based system. Careful choice of various agent based problems will enable us to test the efficiency and expressiveness of the current framework.

- Investigation of the modification of the proposed framework to handle the problem raised by using the "⅋" connective in section 5.2. In section 5.2 we sketched a possible solution to this problem regarding processing a message to oneself.

- Modification of the current version of Lygon interpreter to directly support AOP. This includes:

  - Handling the underlying *non-consumed* resources. This can be done by making the interpreter warn about the non-consumed resources during the program execution. Another approach is the introduction of a new operator, which consumes all the unused resources and prints them.

  - Implementation of IF-THEN-ELSE construct. The current Lygon version does not have this, but in AOP, checking for conditions is very frequent. Currently, we simulate IF-statement by `once`, which is complicated.
  - Solving of the `bot` vs `one` dilemma. In $\mathcal{LC}$ [Vol94] this has been resolved by using $\bot \oplus \mathbf{1}$.
  - Regeneration of consumed resources. Some library functions should be defined to recreate resources after consumption when necessary.
  - Implementation of the sequentiality of goals. Some library functions need to be defined to take a series of goals and execute them sequentially.

- Development of a new debugger to support debugging agent systems written in Lygon. This is important if Lygon is selected as the base language for linear logic AOP. The existing simple debugger will not be able to serve the huge amount of complicated task, demanded by agent-systems.

- Design and development of a linear logic agent-oriented programming language with strong type checking, data structures and modules. Other than logic programming, linear logic is now implemented in functional programming, state and object-oriented programming. As AOP can be considered as a specialisation of OOP, there is a strong possibility that OO linear logic might be fruitful in designing and implementing agent-based systems. We have investigated a logic programming approach via Lygon and come up with some information. Therefore, it would be worth looking at other available linear logic languages and investigate where agent programs fit best. This future work would make it possible either to find the best suitable language or to build up a precise specification to design a new AOP language based on linear logic.

- Investigation of a reactive linear logic agent framework. The word *reactivity* denotes the ability to adapt to the environment without explicit instruction. This may be interpreted as responding to an input without any processing or planning. In the context of a programming language, the word reactivity is more related to autonomy. It is about waiting for the input and making a decision as soon as the input is supplied or sensed. In a broader perspective, an agent can be called *reactive* if a change in the environment triggers an action without the agent needing to be explicitly told to do something. This could be something simple like picking up a can (taken from an example in [Bro91]) or something compound like setting off a long sequence of actions. For example, noticing that a fire has just broken out, turning on the sprinkler system, alerting the fire brigade, making public announcements, etc. This sense of reactivity requires to look at some significant machinery, such as bottom-up execution.

Linear logic is considered to be an efficient tool for prototyping MAS. The Lygon programming language was not originally developed in order to support agent programming. We investigated the agents possibility and came up with some decisions. However, it is clear that this is not the end of the research. We just started with an idea and explored some possibility with an

available linear logic implementation. We had some results; we have analysed them and been able to enrich our knowledge as well as clearly point out some future tasks. This is an ongoing project, and we will learn more about it as we proceed. The goal from a broader perspective is to establish a better framework and tool for the agent-oriented technology, which has the potential to become a prominent technology in the next millennium.

References

[AAI96] AAII. dMARS Technical Overview. *The dMARS V1.6.11 System Overview*, 1996.

[AC87] P. Agre and D. Chapman. An Implementation of a Theory of Activity. *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, 1987.

[Agh90] Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[Ale94] Vladimir Alexiev. Applications of linear logic to computation: An overview. *Bulletin of the IGPL*, 2(1):77–107, March 1994.

[AP91] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[BDM+99] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. In G. Gupta, editor, *Proc. of First International Workshop on Practical Aspects of Declarative Languages*, San Antonio, Texas, January 1999. Springer Verlag. LNCS 1551.

[BIP88] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

[BRHL98] Paolo Busetta, Ralph Ronnquist, Andrew Hodgson, and Andrew Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998.

[Bro91] Rodney A. Brooks. Intelligence Without Reason. *Computers and Thought, IJCAI-91*, 1991.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, second edition, 1984. ISBN - 0-387-15011-0.

[dKLW97] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A Formal Specification of dMARS. Technical Note 72, Australian Artificial Intelligence Institute, November 1997.

[Fer92] I. A. Ferguson. TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents. Technical Report TR273, University of Cambridge, 1992.

[Gir87] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GL87] Michael P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, 1987.

[HM87] S. Hanks and D. MacDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 10(2):327–365, 1994.

[HPW96] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, pages 391–405. Springer, July 1996.

[HW98] James Harland and Michael Winikoff. Making Logic Programs Reactive. Technical report, RMIT University, 1998.

[Jen93] Nicholas R. Jennings. Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information Systems*, 2(3):289–318, 1993.

[JSW98] N.R. Jennings, K.P. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–36, July 1998.

[Kae91] L. P. Kaelbling. A situated automata approach to the design of embedded agents. *SIGART Bulletin*, 2(4):85–88, 1991.

[KY93] Naoki Kobayashi and Akinori Yonezawa. ACL – a concurrent linear logic programming paradigm. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 279–294, Vancouver, Canada, 1993. The MIT Press.

[Mae91] Pattie Maes. The Agent Network Architecture (ANA). *SIGART Bulletin*, 2(4):115–120, 1991.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In *Logic in Computer Science*, pages 272–281, 1994.

[MMZ97] M. Martelli, V. Mascardi, and F. Zini. Applying Logic Programming to the Specification of Complex Applications. *Mathematical Modelling and Scientific Computing*, 8, 1997. Proc. of *11th International Conference on Mathematical and Computer Modelling and Scientific Computing*.

[MPT95] J. P. Muller, M. Pischel, and M. Thiel. Modelling reactive behaviour in vertically layered agent architectures. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents*, pages 261–276. Springer Verlag, LNAI 890, 1995.

[PH94] David Pym and James Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[RG91] Anand S. Rao and Michael P. Georgeff. Modelling rational agents within a BDI-Architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Principles of Knowledge Representation and Reasoning, Proceedings of the Second International Conference*, pages 473–484, April 1991.

[Sce93] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Co., 1993.

[Sce95] A. Scedrov. Linear logic and computation: A survey. In H. Schwichtenberg, editor, *Proof and Computation, Proceedings Marktoberdorf Summer School 1993*, pages 379–395. NATO Advanced Science Institutes, Series F, Springer-Verlag, Berlin, 1995.

[Sho93] Yoav Shoham. Agent-oriented Programming. *Artificial Intelligence 60*, 60(1):51–92, 1993.

[Smi97] D. Smith. MixLog: A Generalized Rule-Based Language. *University of Waikato*, 1997.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog (second edition)*. MIT Press, 1994.

[VB90] S. Vere and T. Bickmore. A basic Agent. *Computational Intelligence*, 6:41–60, 1990.

[Vol94] Paolo Volpe. Concurrent logic programming as uniform linear proofs. In Giorgio Levi and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming*, pages 133–149. Springer-Verlag LNCS 850, September 1994.

[WH95] Michael Winikoff and James Harland. Implementing the linear logic programming language Lygon. In John Lloyd, editor, *International Logic Programming Symposium*, pages 66–80, Portland, Oregon, December 1995. MIT Press.

[WH96] Michael Winikoff and James Harland. Some applications of the linear logic programing language Lygon. In Kotagiri Ramamohanarao, editor, *Australasian Computer Science Conference*, pages 262–271, February 1996.

[Win96] Michael Winikoff. Hitch hiker's guide to Lygon 0.7. Technical Report 96/36, Melbourne University, October 1996.

[Win97] Michael Winikoff. *Logic Programming With Linear Logic*. PhD thesis, University of Melbourne, 1997.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[Xu95] Yixiao Xu. Debugging environment design for the logic programming language Lygon. Master's thesis, Department of Computer Science, RMIT University, February 1995.

A Lygon Standard Library

```
not(X) <- once((call(X) *  eq(Y,succeed)) @
        eq(Y,fail)) * eq(Y,fail).

var(X) <- not(not(eq(X,1))) * not(not(eq(X,2))).

le(X,Y) <- lt(X,Y) @ eq(X,Y).
gt(X,Y) <- lt(Y,X).
ge(X,Y) <- lt(Y,X) @ eq(X,Y).

output(X) <- (!(print(X) * nl)).

repeat <- one @ repeat.

sort([],[]).
sort([X],[X]).
sort([X,Y|Xs],R) <- halve([X,Y|Xs],A,B) * sort(A,R1) *
                sort(B,R2) * merge(R1,R2,R).

merge([],X,X).
merge([X|Xs],[],[X|Xs]).
merge([X|Xs],[Y|Ys],[X|R]) <- le(X,Y) * merge(Xs,[Y|Ys],R).
merge([X|Xs],[Y|Ys],[Y|R]) <- gt(X,Y) * merge([X|Xs],Ys,R).

halve([],[],[]).
halve([X],[X],[]).
halve([X,Y|R],[X|A],[Y|B]) <- halve(R,A,B).

append([],X,X).
append([X|Xs],Y,[X|Z]) <- append(Xs,Y,Z).

reverse(X,Y) <- var(Y) * rev(X,[],Y).
reverse(X,Y) <- not(var(Y)) * rev(Y,[],X).

rev([],X,X).
rev([X|Xs],Y,R) <- rev(Xs,[X|Y],R).

member(X,[_|Y]) <- member(X,Y).
member(X,[X|_]).
```

B Fibonacci Agent Code in JACK

```
//Written by Abdullah-Al AMIN

//THE EVENTS
//a JACK Event: The FibonacciRequest event which is
//used for the initial triggering of the computation
event FibonacciRequest extends Event {
   int  n;
   long result;

   #posted as
   fibonacci(int n){
      this.n = n;
   }
}

//another JACK event: a FibonacciGoal event which is
//raised for invoking computation
event FibonacciGoal extends BDIGoalEvent {
   int n;
   logical long result;

   #posted as
   fibonacci(int n,logical long r){
      this.n = n;
      result = r;
   }
}

//DATABASE
//a JACK database which is to hold knowledge
//about fibonacci numbers
database FibonacciDb extends ClosedWorld {
   #key field int value;
   #value field long fibonacci;

   #indexed query get(int v, logical long f);
   #indexed query get(logical int v, logical long f);
}

//a JACK plan handling the FibonacciRequest event
plan ServerPlan extends Plan {
   #handles event FibonacciRequest ev;
```

```
   #reads database FibonacciDb  known;
   #posts event FibonacciGoal compute;

   body(){
      logical long result;
      @achieve(known.get(ev.n, result),
         compute.fibonacci(ev.n, result));
      System.out.println(ev.n+"! = "+result.as_long());
      ev.result = result.as_long();
   }
}

//Another JACK plan handling the FibonacciGoal event

plan FibonacciPlan extends Plan {
   #handles event FibonacciGoal ev;

   #modifies database FibonacciDb known;
   #posts event FibonacciGoal compute;

   body(){
      System.out.println("  [computing "+ev.n+"!]");
      if (ev.n <= 2) {
         ev.result.unify(1);
      }
      else {
         logical long result1;
         logical long result2;

         @achieve(known.get(ev.n-1, result1),
             compute.fibonacci(ev.n-1, result1));
         @achieve(known.get(ev.n-2, result2),
             compute.fibonacci(ev.n-2, result2));
         ev.result.unify(result1.as_long()
             + result2.as_long());
      }
   }

   #reasoning method
   pass(){
      known.assert(ev.n, ev.result.as_long());
   }
}

//a JACK agent which has the outlined capabilities,
```

```
//and in particular an interface method that Java code
//may call to demand the fibonacci computation
agent FibonacciAgent extends Agent {
   #private database FibonacciDb known();

   #posts event FibonacciRequest compute;

   public
   FibonacciAgent(String s){
      super(s);
   }

   public long
   fibonacci(int i){
      FibonacciRequest r = compute.fibonacci(i);

      if (postEventAndWait(r))
         return r.result;
      return -1;
   }

   #handles event FibonacciGoal;
   #handles event FibonacciRequest;
   #uses plan ServerPlan;
   #uses plan FibonacciPlan;

}

//a Test class that contains the application main() function,
//which creates an instance of the JACK agent,
//and then demands fibonacci computation
public class Test {
   public static void main(String [] arg){
      FibonacciAgent fib = new FibonacciAgent("Leonardo");

      for (int i=1; i<9; i++)
      System.out.println("The "+i+"th Fibonacci number is "
                +fib.fibonacci(i));

      System.exit(0);
   }
}
```